Sage Reference Manual: Polynomial Rings

Release 6.6

The Sage Development Team

April 18, 2015

CONTENTS

CHAPTER

ONE

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).

`sage.rings.polynomial.polynomial_ring_constructor.`**`BooleanPolynomialRing_constructor`**(*n=None, names=N or- der='lex'*

Construct a boolean polynomial ring with the following parameters:

INPUT:

- `n` – number of variables (an integer > 1)
- `names` – names of ring variables, may be a string or list/tuple of strings
- `order` – term order (default: lex)

EXAMPLES:

```
sage: R.<x, y, z> = BooleanPolynomialRing() # indirect doctest
sage: R
Boolean PolynomialRing in x, y, z

sage: p = x*y + x*z + y*z
sage: x*p
x*y*z + x*y + x*z

sage: R.term_order()
Lexicographic term order

sage: R = BooleanPolynomialRing(5,'x',order='deglex(3),deglex(2)')
sage: R.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='degneglex')
sage: R.term_order()
Degree negative lexicographic term order

sage: BooleanPolynomialRing(names=('x','y'))
Boolean PolynomialRing in x, y
```

```
sage: BooleanPolynomialRing(names='x,y')
Boolean PolynomialRing in x, y
```

TESTS:

```
sage: P.<x,y> = BooleanPolynomialRing(2,order='deglex')
sage: x > y
True

sage: P.<x0, x1, x2, x3> = BooleanPolynomialRing(4,order='deglex(2),deglex(2)')
sage: x0 > x1
True
sage: x2 > x3
True
```

`sage.rings.polynomial.polynomial_ring_constructor.`**`PolynomialRing`**`(`*base_ring, arg1=None, arg2=None, sparse=False, order='degrevlex', names=None, name=None, var_array=None, implementation=None*`)`

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

There are five ways to call the polynomial ring constructor:

1. `PolynomialRing(base_ring, name, sparse=False)`
2. `PolynomialRing(base_ring, names, order='degrevlex')`
3. `PolynomialRing(base_ring, name, n, order='degrevlex')`
4. `PolynomialRing(base_ring, n, name, order='degrevlex')`
5. `PolynomialRing(base_ring, n, var_array=var_array, order='degrevlex')`

The optional arguments sparse and order *must* be explicitly named, and the other arguments must be given positionally.

INPUT:

- `base_ring` – a ring
- `name` – a string
- `names` – a list or tuple of names, or a comma separated string
- `var_array` – a list or tuple of names, or a comma separated string
- `n` – an integer
- `sparse` – bool (default: False), whether or not elements are sparse
- `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 $\mathbf{Z}[x]$ can be implemented with 'NTL' or 'FLINT'; default is 'FLINT')

Note: The following rules were introduced in trac ticket #9944, in order to preserve the "unique parent assumption" in Sage (i.e., if two parents evaluate equal then they should actually be identical).

- In the multivariate case, a dense representation is not supported. Hence, the argument `sparse=False` is silently ignored in that case.
- 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 of VARIABLE NAME CONTEXT:

```
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
```

SQUARE BRACKETS NOTATION: You can alternatively create a single or multivariate polynomial ring over a ring R by writing `R['varname']` or `R['var1,var2,var3,...']`. This square brackets notation doesn't allow for setting any of the optional arguments.

EXAMPLES:

1. `PolynomialRing(base_ring, name, sparse=False)`

```
sage: PolynomialRing(QQ, '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:

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

You must specify a name:

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

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

sage: R.<w> = PolynomialRing(PolynomialRing(GF(7),'k')); R
Univariate Polynomial Ring in w over Univariate Polynomial Ring in k over Finite Field of si
```

The square bracket notation:

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

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

```
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 * x^1000000 in FLINT may be unwise.

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

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

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

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

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

2.`PolynomialRing(base_ring, names, order='degrevlex')`

```
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.

```
sage: (R is S) and  (S is T)
True
```

There is a unique polynomial ring with each term order:

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

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

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

3. `PolynomialRing(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.

```
sage: PolynomialRing(QQ, 'x', 10)
Multivariate Polynomial Ring in x0, x1, x2, x3, x4, x5, x6, x7, x8, x9 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 first 100 primes:

```
sage: R = PolynomialRing(ZZ, ['x%s'%p for p in primes(100)]); R
Multivariate Polynomial Ring in x2, x3, x5, x7, x11, x13, x17, x19, x23, x29, x31, x37, x41,
```

By calling the `inject_variables()` method, all those variable names are available for interactive use:

```
sage: R.inject_variables()
Defining x2, x3, x5, x7, x11, x13, x17, x19, x23, x29, x31, x37, x41, x43, x47, x53, x59, x6
```

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

5. `PolynomialRing(base_ring, n, m, var_array=var_array, order='degrevlex')`

   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

   If `var_array` is a single string, this creates an $m \times n$ array of variables:

   ```
   sage: PolynomialRing(ZZ, 2, 3, var_array='m')
   Multivariate Polynomial Ring in m00, m01, m02, m10, m11, m12 over Integer Ring
   ```

   If `var_array` is a single string and m is not specified, this creates an $n \times n$ array of variables:

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

TESTS:

We test here some changes introduced in trac ticket #9944.

If there is no dense implementation for the given number of variables, then requesting a dense ring results yields the corresponding sparse ring:

```
sage: R.<x,y> = QQ[]
sage: S.<x,y> = PolynomialRing(QQ, sparse=False)
sage: R is S
True
```

If the requested implementation is not known or not supported for the given number of variables and the given sparsity, then an error results:

```
sage: R.<x> = PolynomialRing(ZZ, implementation='Foo')
Traceback (most recent call last):
...
ValueError: Unknown implementation Foo for ZZ[x]
sage: R.<x,y> = PolynomialRing(ZZ, implementation='FLINT')
Traceback (most recent call last):
...
ValueError: The FLINT implementation is not known for multivariate polynomial rings
```

The following corner case used to result in a warning message from `libSingular`, and the generators of the resulting polynomial ring were not zero:

```
sage: R = Integers(1)['x','y']
sage: R.0 == 0
True
```

We verify that trac ticket #13187 is fixed:

```
sage: var('t')
t
sage: PolynomialRing(ZZ, name=t) == PolynomialRing(ZZ, name='t')
True
```

We verify that polynomials with interval coefficients from trac ticket #7712 and trac ticket #13760 are fixed:

```
sage: P.<y,z> = PolynomialRing(RealIntervalField(2))
sage: Q.<x> = PolynomialRing(P)
sage: C = (y-x)^3
sage: C(y/2)
1.?*y^3
sage: R.<x,y> = PolynomialRing(RIF,2)
sage: RIF(-2,1)*x
0.?e1*x
```

`sage.rings.polynomial.polynomial_ring_constructor.`**`polynomial_default_category`**(*base_ring_categor mul- ti- vari- ate*)

Choose an appropriate category for a polynomial ring.

INPUT:

- `base_ring_category`: The category of ring over which the polynomial ring shall be defined.
- `multivariate`: Will the polynomial ring be multivariate?

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_ring_constructor import polynomial_default_category
sage: polynomial_default_category(Rings(), False) is Algebras(Rings())
True
sage: polynomial_default_category(Rings().Commutative(),False) is Algebras(Rings().Commutative()
True
sage: polynomial_default_category(Fields(),False) is EuclideanDomains() & Algebras(Fields())
True
sage: polynomial_default_category(Fields(),True) is UniqueFactorizationDomains() & CommutativeAl
True

sage: QQ['t'].category() is EuclideanDomains() & CommutativeAlgebras(QQ.category())
True
sage: QQ['s','t'].category() is UniqueFactorizationDomains() & CommutativeAlgebras(QQ.category()
True
sage: QQ['s']['t'].category() is UniqueFactorizationDomains() & CommutativeAlgebras(QQ['s'].cate
True
```

CHAPTER

TWO

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 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.
- Simon King (2011-10): Choice of categories for polynomial rings.

EXAMPLES:

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

Saving and loading of polynomial rings works:

```
sage: loads(dumps(QQ['x'])) == QQ['x']
True
sage: k = PolynomialRing(QQ['x'],'y'); loads(dumps(k))==k
True
sage: k = PolynomialRing(ZZ,'y'); loads(dumps(k)) == k
True
sage: k = PolynomialRing(ZZ,'y', sparse=True); loads(dumps(k))
Sparse Univariate Polynomial Ring in y over Integer Ring
```

Rings with different variable names are not equal; in fact, by trac ticket #9944, polynomial rings are equal if and only if they are identical (which should be the case for all parent structures in Sage):

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

We create a polynomial ring over a quaternion algebra:

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

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

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

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

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

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

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

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

TESTS:

```
sage: K.<x>=FractionField(QQ['x'])
sage: V.<z> = K[]
sage: x+z
z + x
```

Check that trac ticket #5562 has been fixed:

```
sage: R.<u> = PolynomialRing(RDF, 1, 'u')
sage: v1 = vector([u])
sage: v2 = vector([CDF(2)])
sage: v1 * v2
2.0*u
```

These may change over time:

```
sage: x = var('x')
sage: type(ZZ['x'].0)
<type 'sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint'>
sage: type(QQ['x'].0)
<type 'sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint'>
sage: type(RR['x'].0)
<type 'sage.rings.polynomial.polynomial_real_mpfr_dense.PolynomialRealDense'>
sage: type(Integers(4)['x'].0)
<type 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>
sage: type(Integers(5*2^100)['x'].0)
<type 'sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_ZZ'>
sage: type(CC['x'].0)
<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_field'>
sage: type(CC['t']['x'].0)
<type 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
sage: type(NumberField(x^2+1,'I')['x'].0)
<class 'sage.rings.polynomial.polynomial_number_field.Polynomial_absolute_number_field_dense'>
sage: type(NumberField(x^2+1,'I')['x'])
<class 'sage.rings.polynomial.polynomial_ring.PolynomialRing_field_with_category'>
sage: type(NumberField([x^2-2,x^2-3],'a')['x'].0)
<class 'sage.rings.polynomial.polynomial_number_field.Polynomial_relative_number_field_dense'>
sage: type(NumberField([x^2-2,x^2-3],'a')['x'])
<class 'sage.rings.polynomial.polynomial_ring.PolynomialRing_field_with_category'>
```

class `sage.rings.polynomial.polynomial_ring.`**`PolynomialRing_commutative`**(*base_ring*, *name=None*, *sparse=False*, *element_class=None*, *category=None*)

Bases: `sage.rings.polynomial.polynomial_ring.PolynomialRing_general`, `sage.rings.ring.CommutativeAlgebra`

Univariate polynomial ring over a commutative ring.

quotient_by_principal_ideal(*f*, *names=None*)

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.

EXAMPLES:

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

The same example, using the polynomial instead of the ideal, and customizing the variable name:

```
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
```

TESTS:

Quotienting by the zero ideal returns self (trac ticket #5978):

```
sage: R = QQ['x']
sage: R.quotient_by_principal_ideal(R.zero_ideal()) is R
True
sage: R.quotient_by_principal_ideal(0) is R
True
```

weyl_algebra()

Return the Weyl algebra generated from self.

EXAMPLES:

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

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

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_field

Univariate polynomial ring over a finite field.

EXAMPLE:

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

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

Construct a monic irreducible polynomial of degree n.

INPUT:

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

OUTPUT:

A monic irreducible polynomial of degree n in `self`.

EXAMPLES:

```
sage: GF(5^3, 'a')['x'].irreducible_element(2)
x^2 + (4*a^2 + a + 4)*x + 2*a^2 + 2
sage: GF(19)['x'].irreducible_element(21, algorithm="first_lexicographic")
x^21 + x + 5
sage: GF(5**2, 'a')['x'].irreducible_element(17, algorithm="first_lexicographic")
x^17 + a*x + 4*a + 3
```

AUTHORS:

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

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

Bases: `sage.rings.polynomial.polynomial_ring.PolynomialRing_commutative`

TESTS:

```
sage: from sage.rings.polynomial.polynomial_ring import PolynomialRing_dense_mod_n as PRing
sage: R = PRing(Zmod(15), 'x'); R
Univariate Polynomial Ring in x over Ring of integers modulo 15
sage: type(R.gen())
<type 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>

sage: R = PRing(Zmod(15), 'x', implementation='NTL'); R
Univariate Polynomial Ring in x over Ring of integers modulo 15 (using NTL)
sage: type(R.gen())
<type 'sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_zz'>

sage: R = PRing(Zmod(2**63*3), 'x', implementation='NTL'); R
Univariate Polynomial Ring in x over Ring of integers modulo 27670116110564327424 (using NTL)
sage: type(R.gen())
<type 'sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_ZZ'>

sage: R = PRing(Zmod(2**63*3), 'x', implementation='FLINT')
Traceback (most recent call last):
...
ValueError: FLINT does not support modulus 27670116110564327424
```

```
sage: R = PRing(Zmod(2**63*3), 'x'); R
Univariate Polynomial Ring in x over Ring of integers modulo 27670116110564327424 (using NTL)
sage: type(R.gen())
<type 'sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_ZZ'>
```

modulus()

EXAMPLES:

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

residue_field(*ideal*, *names=None*)

Return the residue finite field at the given ideal.

EXAMPLES:

```
sage: R.<t> = GF(2)[]
sage: k.<a> = R.residue_field(t^3+t+1); k
Residue field in a of Principal ideal (t^3 + t + 1) of Univariate Polynomial Ring in t over
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
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 F
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 R
sage: k.cardinality()
256
```

Non-maximal ideals are not accepted:

```
sage: R.residue_field(t^2 + 1)
Traceback (most recent call last):
...
ArithmeticError: ideal is not maximal
sage: R.residue_field(0)
Traceback (most recent call last):
...
ArithmeticError: ideal is not maximal
sage: R.residue_field(1)
Traceback (most recent call last):
...
ArithmeticError: ideal is not maximal
```

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

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

TESTS:

```
sage: P = GF(2)['x']; P
Univariate Polynomial Ring in x over Finite Field of size 2 (using NTL)
sage: type(P.gen())
<type 'sage.rings.polynomial.polynomial_gf2x.Polynomial_GF2X'>

sage: from sage.rings.polynomial.polynomial_ring import PolynomialRing_dense_mod_p
sage: P = PolynomialRing_dense_mod_p(GF(5), 'x'); P
Univariate Polynomial Ring in x over Finite Field of size 5
sage: type(P.gen())
<type 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>

sage: P = PolynomialRing_dense_mod_p(GF(5), 'x', implementation='NTL'); P
Univariate Polynomial Ring in x over Finite Field of size 5 (using NTL)
sage: type(P.gen())
<type 'sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_p'>

sage: P = PolynomialRing_dense_mod_p(GF(9223372036854775837), 'x')
sage: P
Univariate Polynomial Ring in x over Finite Field of size 9223372036854775837 (using NTL)
sage: type(P.gen())
<type 'sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_p'>
```

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

Construct a monic irreducible polynomial of degree n.

INPUT:

- `n` – integer: the degree of the polynomial to construct
- `algorithm` – string: algorithm to use, or `None`. Currently available options are:
  - **'adleman-lenstra': a variant of the Adleman–Lenstra** algorithm as implemented in PARI.
  - `'conway'`: look up the Conway polynomial of degree n over the field of p elements in the database; raise a `RuntimeError` if it is not found.
  - `'ffprimroot'`: use the `ffprimroot()` function from PARI.
  - `'first_lexicographic'`: return the lexicographically smallest irreducible polynomial of 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 `algorithm` is `None`, use $x - 1$ in degree 1. In degree > 1, the Conway polynomial is used if it is found in the database. Otherwise, the algorithm `minimal_weight` is used if $p = 2$, and the algorithm `adleman-lenstra` if $p > 2$.

OUTPUT:

A monic irreducible polynomial of degree n in `self`.

EXAMPLES:

```
sage: GF(5)['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 + ...
```

In characteristic 2:

```
sage: GF(2)['x'].irreducible_element(33)
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")
x^33 + x^10 + 1
```

In degree 1:

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

AUTHORS:

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

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

Bases: `sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_field_generic`

TESTS:

```
sage: from sage.rings.polynomial.polynomial_ring import PolynomialRing_dense_padic_field_capped_
sage: R = PRing(Qp(13), name='t'); R
Univariate Polynomial Ring in t over 13-adic Field with capped relative precision 20
sage: type(R.gen())
<class 'sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_cap
```

class `sage.rings.polynomial.polynomial_ring.`**`PolynomialRing_dense_padic_field_generic`**(*base_ring*, *name='x'*, *sparse=Fal* *el-* *e-* *ment_class*

Bases: `sage.rings.polynomial.polynomial_ring.PolynomialRing_field`

TESTS:

```
sage: from sage.rings.polynomial.polynomial_ring import PolynomialRing_field as PRing
sage: R = PRing(QQ, 'x'); R
Univariate Polynomial Ring in x over Rational Field
sage: type(R.gen())
<type 'sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint'>
sage: R = PRing(QQ, 'x', sparse=True); R
Sparse Univariate Polynomial Ring in x over Rational Field
sage: type(R.gen())
<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_field'>
sage: R = PRing(CC, 'x'); R
Univariate Polynomial Ring in x over Complex Field with 53 bits of precision
sage: type(R.gen())
<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_field'>
```

Demonstrate that trac ticket #8762 is fixed:

```
sage: R.<x> = PolynomialRing(GF(next_prime(10^20)), sparse=True)
sage: x^(10^20) # this should be fast
x^100000000000000000000
```

class `sage.rings.polynomial.polynomial_ring.`**`PolynomialRing_dense_padic_field_lazy`**(*base_ring*, *name=None*, *el- e- ment_class=No*

Bases: `sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_field_generic`

TESTS:

```
sage: from sage.rings.polynomial.polynomial_ring import PolynomialRing_dense_padic_field_lazy as
sage: R = PRing(Qp(13, type='lazy'), name='t')
Traceback (most recent call last):
...
NotImplementedError: lazy p-adics need more work.  Sorry.

#sage: type(R.gen())
```

class `sage.rings.polynomial.polynomial_ring.`**`PolynomialRing_dense_padic_ring_capped_absolute`**(*b n e e n*

Bases: `sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_generic`

TESTS:

```
sage: from sage.rings.polynomial.polynomial_ring import PolynomialRing_dense_padic_ring_capped_a
sage: R = PRing(Zp(13, type='capped-abs'), name='t'); R
Univariate Polynomial Ring in t over 13-adic Ring with capped absolute precision 20
sage: type(R.gen())
<class 'sage.rings.polynomial.padics.polynomial_padic_flat.Polynomial_padic_flat'>
```

class `sage.rings.polynomial.polynomial_ring.`**`PolynomialRing_dense_padic_ring_capped_relative`**(*b n e e n*

Bases: `sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_generic`

TESTS:

```
sage: from sage.rings.polynomial.polynomial_ring import PolynomialRing_dense_padic_ring_capped_r
sage: R = PRing(Zp(13), name='t'); R
Univariate Polynomial Ring in t over 13-adic Ring with capped relative precision 20
sage: type(R.gen())
<class 'sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_cap
```

class sage.rings.polynomial.polynomial_ring.**PolynomialRing_dense_padic_ring_fixed_mod**(*base_ring*, *name=No* *el-e-ment_clas*

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_generic

TESTS:

```
sage: from sage.rings.polynomial.polynomial_ring import PolynomialRing_dense_padic_ring_fixed_mo
sage: R = PRing(Zp(13, type='fixed-mod'), name='t'); R
Univariate Polynomial Ring in t over 13-adic Ring of fixed modulus 13^20

sage: type(R.gen())
<class 'sage.rings.polynomial.padics.polynomial_padic_flat.Polynomial_padic_flat'>
```

class sage.rings.polynomial.polynomial_ring.**PolynomialRing_dense_padic_ring_generic**(*base_ring*, *name='x'*, *sparse=False*, *im-ple-men-ta-tion=None*, *el-e-ment_class=*

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_integral_domain

TESTS:

```
sage: from sage.rings.polynomial.polynomial_ring import PolynomialRing_integral_domain as PRing
sage: R = PRing(ZZ, 'x'); R
Univariate Polynomial Ring in x over Integer Ring
sage: type(R.gen())
<type 'sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint'>

sage: R = PRing(ZZ, 'x', implementation='NTL'); R
Univariate Polynomial Ring in x over Integer Ring (using NTL)
sage: type(R.gen())
<type 'sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl'>
```

class sage.rings.polynomial.polynomial_ring.**PolynomialRing_dense_padic_ring_lazy**(*base_ring*, *name=None*, *el-e-ment_class=None*

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_generic

TESTS:

```
sage: from sage.rings.polynomial.polynomial_ring import PolynomialRing_dense_padic_ring_lazy as
sage: R = PRing(Zp(13, type='lazy'), name='t')
```

```
Traceback (most recent call last):
...
NotImplementedError: lazy p-adics need more work.  Sorry.

#sage: type(R.gen())
```

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

Bases: `sage.rings.polynomial.polynomial_ring.PolynomialRing_integral_domain`, `sage.rings.polynomial.polynomial_singular_interface.PolynomialRing_singular_repr`, `sage.rings.ring.PrincipalIdealDomain`

TESTS:

```
sage: from sage.rings.polynomial.polynomial_ring import PolynomialRing_field as PRing
sage: R = PRing(QQ, 'x'); R
Univariate Polynomial Ring in x over Rational Field
sage: type(R.gen())
<type 'sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint'>
sage: R = PRing(QQ, 'x', sparse=True); R
Sparse Univariate Polynomial Ring in x over Rational Field
sage: type(R.gen())
<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_field'>
sage: R = PRing(CC, 'x'); R
Univariate Polynomial Ring in x over Complex Field with 53 bits of precision
sage: type(R.gen())
<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_field'>
```

Demonstrate that trac ticket #8762 is fixed:

```
sage: R.<x> = PolynomialRing(GF(next_prime(10^20)), sparse=True)
sage: x^(10^20) # this should be fast
x^100000000000000000000
```

`divided_difference`(*points*, *full_table=False*)

Return the Newton divided-difference coefficients of the n-th Lagrange interpolation polynomial of `points`.

If `points` are $n+1$ distinct points $(x_0, f(x_0)), (x_1, f(x_1)), \dots, (x_n, f(x_n))$, then $P_n(x)$ is the n-th Lagrange interpolation polynomial of $f(x)$ that passes through the points $(x_i, f(x_i))$. This method returns the coefficients $F_{i,i}$ such that

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

INPUT:

- `points` – a list of tuples $(x_0, f(x_0)), (x_1, f(x_1)), \dots, (x_n, f(x_n))$ where each $x_i \neq x_j$ for $i \neq j$
- `full_table` – (default: `False`) if `True` then return the full divided-difference table; if `False` then only return entries along the main diagonal. The entries along the main diagonal are the Newton divided-difference coefficients $F_{i,i}$.

OUTPUT:

- The Newton divided-difference coefficients of the n-th Lagrange interpolation polynomial that passes through the points in `points`.

EXAMPLES:

Only return the divided-difference coefficients $F_{i,i}$. This example is taken from Example 1, p.121 of [BF05]:

```
sage: points = [(1.0, 0.7651977), (1.3, 0.6200860), (1.6, 0.4554022), (1.9, 0.2818186), (2.2
sage: R = PolynomialRing(QQ, "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
sage: R = PolynomialRing(QQ, "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,
-0.0494433333333339,
0.0658783950617283],
[0.110362300000000,
-0.571520999999999,
0.0118183333333349,
0.0680685185185209,
0.00182510288066044]]
```

The following example is taken from Example 4.12, p.225 of [MF99]:

```
sage: points = [(1, -3), (2, 0), (3, 15), (4, 48), (5, 105), (6, 192)]
sage: R = PolynomialRing(RR, "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],
[105, 57, 12, 1, 0],
[192, 87, 15, 1, 0, 0]]
```

REFERENCES:

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 in `self` associated to the given list of points.

Given a list of points, i.e. tuples of elements of `self`'s base ring, this function returns the interpolation polynomial in the Lagrange form.

INPUT:

- `points` – a list of tuples representing points through which the polynomial returned by this function must pass.
- `algorithm` – (default: `'divided_difference'`) the available values for this option are `'divided_difference'` and `neville`.
  - If `algorithm='divided_difference'` then use the method of divided difference.
  - If `algorithm='neville'` then adapt Neville's method as described on page 144 of [BF05] 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 together 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 in, 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 which contains a better Lagrange interpolation polynomial.

EXAMPLES:

By default, we use the method of divided-difference:

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

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

```
sage: R = PolynomialRing(QQ, 'x')
sage: R.lagrange_polynomial([(0,1),(2,2),(3,-2),(-4,9)], algorithm="neville")

[9,
-11/7*x + 19/7,
-17/42*x^2 - 83/42*x + 53/7,
-23/84*x^3 - 11/84*x^2 + 13/7*x + 1]
```

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

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

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

TESTS:

The value for `algorithm` must be either `'divided_difference'` (by default it is), or `'neville'`:

```
sage: R = PolynomialRing(QQ, "x")
sage: R.lagrange_polynomial([(0,1),(2,2),(3,-2),(-4,9)], algorithm="abc")
Traceback (most recent call last):
...
ValueError: algorithm must be one of 'divided_difference' or 'neville'
sage: R.lagrange_polynomial([(0,1),(2,2),(3,-2),(-4,9)], algorithm="divided difference")
Traceback (most recent call last):
...
ValueError: algorithm must be one of 'divided_difference' or 'neville'
sage: R.lagrange_polynomial([(0,1),(2,2),(3,-2),(-4,9)], algorithm="")
Traceback (most recent call last):
...
ValueError: algorithm must be one of 'divided_difference' or 'neville'
```

Make sure that ticket #10304 is fixed. The return value should always be an element of `self` in the case of `divided_difference`, or a list of elements of `self` in the case of `neville`.

```
sage: R = PolynomialRing(QQ, "x")
sage: R.lagrange_polynomial([]).parent() == R
True
sage: R.lagrange_polynomial([(2, 3)]).parent() == R
True
sage: row = R.lagrange_polynomial([], algorithm='neville')
sage: all(poly.parent() == R for poly in row)
True
sage: row = R.lagrange_polynomial([(2, 3)], algorithm='neville')
sage: all(poly.parent() == R for poly in row)
True
```

REFERENCES:

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

Bases: `sage.rings.ring.Algebra`

Univariate polynomial ring over a ring.

`base_extend`(*R*)

Return the base extension of this polynomial ring to R.

EXAMPLES:

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

`change_ring`(*R*)

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

EXAMPLES:

```
sage: R.<ZZZ> = RealIntervalField() []; R
Univariate Polynomial Ring in ZZZ over Real Interval Field with 53 bits of precision
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()
19
```

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

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

EXAMPLES:

```
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: 1/PP(f)
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
```

construction()

x.__init__(...) initializes x; see help(type(x)) for signature

cyclotomic_polynomial(*n*)

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

EXAMPLES:

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

TESTS:

Make sure it agrees with other systems for the trivial case:

```
sage: ZZ['x'].cyclotomic_polynomial(1)
x - 1
sage: gp('polcyclo(1)')
x - 1
```

extend_variables(*added_names*, *order='degrevlex'*)

Returns a multivariate polynomial ring with the same base ring but with added_names as additional variables.

EXAMPLES:

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

```
sage: R.extend_variables(('y', 'z'))
Multivariate Polynomial Ring in x, y, z over Integer Ring
```

gen(*n=0*)

Return the indeterminate generator of this polynomial ring.

EXAMPLES:

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

An identical generator is always returned.

```
sage: t is R.gen()
True
```

gens_dict()

Returns a dictionary whose keys are the variable names of this ring as strings and whose values are the corresponding generators.

EXAMPLES:

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

is_exact()

x.__init__(...) initializes x; see help(type(x)) for signature

is_field(*proof=True*)

Return False, since polynomial rings are never fields.

EXAMPLES:

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

is_finite()

Return False since polynomial rings are not finite (unless the base ring is 0.)

EXAMPLES:

```
sage: R = Integers(1)['x']
sage: R.is_finite()
True
sage: R = GF(7)['x']
sage: R.is_finite()
False
sage: R['x']['y'].is_finite()
False
```

is_integral_domain(*proof=True*)

EXAMPLES:

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

is_noetherian()

x.__init__(...) initializes x; see help(type(x)) for signature

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
sage: R.is_sparse()
True
```

is_unique_factorization_domain(*proof=True*)

EXAMPLES:

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

karatsuba_threshold()

Return the Karatsuba threshold used for this ring by the method _mul_karatsuba to fall back to the school-book 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: R.<z> = GF(9,'a')[]; R
Univariate Polynomial Ring in z over Finite Field in a of size 3^2
sage: R.krull_dimension()
1
sage: S.<t> = R[]
sage: S.krull_dimension()
2
sage: for n in range(10):
...     S = PolynomialRing(S,'w')
```

```
sage: S.krull_dimension()
12
```

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

Return an iterator over the monic polynomials of specified degree.

INPUT: Pass exactly one of:

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

OUTPUT: an iterator

EXAMPLES:

```
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 + (a + 1)*y + 1
y^2 + y
y^2 + y + a
y^2 + y + a + 1
y^2 + y + 1
sage: for p in P.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

ngens ()

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

EXAMPLES:

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

parameter()

Return the generator of this polynomial ring.

This is the same as `self.gen()`.

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

Return an iterator over the polynomials of specified degree.

INPUT: Pass exactly one of:

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

OUTPUT: an iterator

EXAMPLES:

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

random_element(*degree=(-1, 2)*, **args*, ***kwds*)

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

INPUT:

- `degree` - optional integer for fixing the degree or or a tuple of minimum and maximum degrees. By default set to `(-1,2)`.
- `*args, **kwds` - Passed on to the `random_element` method for the base ring

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: R.random_element(10, 5,10)
9*x^10 + 8*x^9 + 6*x^8 + 8*x^7 + 8*x^6 + 9*x^5 + 8*x^4 + 8*x^3 + 6*x^2 + 8*x + 8
sage: R.random_element(6)
x^6 - 3*x^5 - x^4 + x^3 - x^2 + x + 1
sage: R.random_element(6)
-2*x^6 - 2*x^5 + 2*x^4 - 3*x^3 + 1
sage: R.random_element(6)
-x^6 + x^5 - x^4 + 4*x^3 - x^2 + x
```

If a tuple of two integers is given for the degree argument, a polynomial of degree in between the bound is given:

```
sage: R.random_element(degree=(0,8))
x^8 + 4*x^7 + 2*x^6 - x^4 + 4*x^3 - 5*x^2 + x + 14
sage: R.random_element(degree=(0,8))
-5*x^7 + x^6 - 3*x^5 + 4*x^4 - x^2 - 2*x + 1
```

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

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

TESTS:

```
sage: R.random_element(degree=[5])
Traceback (most recent call last):
...
ValueError: degree argument must be an integer or a tuple of 2 integers (min_degree, max_deg

sage: R.random_element(degree=(5,4))
Traceback (most recent call last):
...
ValueError: minimum degree must be less or equal than maximum degree
```

Check that trac ticket #16682 is fixed:

```
sage: R = PolynomialRing(GF(2), 'z')
sage: for _ in xrange(100):
....:     d = randint(-1,20)
....:     P = R.random_element(degree=d)
....:     assert P.degree() == d, "problem with {} which has not degree {}".format(P,d)

sage: R.random_element(degree=-2)
Traceback (most recent call last):
...
ValueError: degree should be an integer greater or equal than -1
```

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:

```
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:

```
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*)

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

EXAMPLES:

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

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

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_commutative, sage.rings.ring.IntegralDomain

TESTS:

```
sage: from sage.rings.polynomial.polynomial_ring import PolynomialRing_integral_domain as PRing
sage: R = PRing(ZZ, 'x'); R
Univariate Polynomial Ring in x over Integer Ring
sage: type(R.gen())
<type 'sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint'>

sage: R = PRing(ZZ, 'x', implementation='NTL'); R
Univariate Polynomial Ring in x over Integer Ring (using NTL)
sage: type(R.gen())
<type 'sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl'>
```

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
```

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['w'])
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: is_PolynomialRing(PolynomialRing(ZZ,1,'w'))
False
sage: R = PolynomialRing(ZZ,1,'w'); R
Multivariate Polynomial Ring in w over Integer Ring
sage: is_PolynomialRing(R)
False
sage: type(R)
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular'>
```

sage.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 polygens.

sage.rings.polynomial.polynomial_ring.**polygens**(*base_ring*, *names='x'*)

Return indeterminates over the given base ring with the given names.

EXAMPLES:

```
sage: x,y,z = polygens(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 = polygens(QQ,['x','yz','abc'])
sage: t
(x, yz, abc)
```

2.2 Univariate Polynomial Base Class

AUTHORS:

- William Stein: first version.
- Martin Albrecht: Added singular coercion.
- Robert Bradshaw: Move Polynomial_generic_dense to Cython.
- Miguel Marco: Implemented resultant in the case where PARI fails.
- Simon King: Use a faster way of conversion from the base ring.
- Julian Rueth (2012-05-25,2014-05-09): Fixed is_squarefree() for imperfect fields, fixed division without remainder over QQbar; added `_cache_key` for polynomials with unhashable coefficients
- Simon King (2013-10): Implement copying of `PolynomialBaseringInjection`.

TESTS:

```
sage: R.<x> = ZZ[]
sage: f = x^5 + 2*x^2 + (-1)
sage: f == loads(dumps(f))
True
```

class `sage.rings.polynomial.polynomial_element.`**`ConstantPolynomialSection`**

Bases: `sage.categories.map.Map`

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

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

EXAMPLES:

```
sage: P0.<y_1> = GF(3)[]
sage: P1.<y_2,y_1,y_0> = GF(3)[]
sage: P0(-y_1)    # indirect doctest
2*y_1

sage: phi = GF(3).convert_map_from(P0); phi
Generic map:
  From: Univariate Polynomial Ring in y_1 over Finite Field of size 3
  To:   Finite Field of size 3
sage: type(phi)
<type 'sage.rings.polynomial.polynomial_element.ConstantPolynomialSection'>
sage: phi(P0.one())
1
sage: phi(y_1)
Traceback (most recent call last):
...
TypeError: not a constant polynomial
```

class `sage.rings.polynomial.polynomial_element.`**`Polynomial`**

Bases: `sage.structure.element.CommutativeAlgebraElement`

A polynomial.

EXAMPLE:

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

`add_bigoh`(*prec*)

Returns 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
```

`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. Returns a root of degree `abs(degree)` over the ground field. If negative, also assumes that all factors of this polynomial are of degree `abs(degree)`. If None, returns a root of minimal degree contained within the given ring.
- `assume_squarefree` (bool) – Used for polynomials over finite fields. If True, this polynomial is assumed to be squarefree.

EXAMPLES:

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

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

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

TESTS:

```
sage: R.<x> = GF(5)[]
sage: K.<a> = GF(5^12)
sage: for _ in range(40):
....:     f = R.random_element(degree=4)
....:     assert f(f.any_root(K)) == 0
```

Check that our Cantor-Zassenhaus implementation does not loop over finite fields of even characteristic (see trac ticket #16162):

```
sage: K.<a> = GF(2**8)
sage: x = polygen(K)
sage: (x**2+x+1).any_root() # used to loop
Traceback (most recent call last):
...
ValueError: no roots A 1
sage: (x**2+a+1).any_root()
a^7 + a^2
```

Also check that such computations can be interrupted:

```
sage: K.<a> = GF(2^8)
sage: x = polygen(K)
sage: pol = x^1000000 + x + a
sage: alarm(0.5); pol.any_root()
Traceback (most recent call last):
...
AlarmInterrupt
```

Check root computation over large finite fields:

```
sage: K.<a> = GF(2**50)
sage: x = polygen(K)
sage: (x**10+x+a).any_root()
a^49 + a^47 + a^44 + a^42 + a^41 + a^39 + a^38 + a^37 + a^36 + a^34 + a^33 + a^29 + a^27 + a
sage: K.<a> = GF(2**150)
sage: x = polygen(K)
sage: (x**10+x+a).any_root()
a^149 + a^148 + a^146 + a^144 + a^143 + a^140 + a^138 + a^136 + a^134 + a^132 + a^131 + a^13
```

args()

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

EXAMPLES:

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

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

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

base_extend(*R*)

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

EXAMPLES:

```
sage: R.<x> = QQ[]
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.

EXAMPLES:

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

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]
```

coeffs()

Using `coeffs()` is now deprecated (trac ticket #17518). Returns `self.list()`.

(It is potentially slightly faster to use `self.list()` directly.)

EXAMPLES:

```
sage: x = QQ['x'].0
sage: f = 10*x^3 + 5*x + 2/17
sage: f.coeffs()
doctest:...: DeprecationWarning: The use of coeffs() is now deprecated in favor of coefficie
See http://trac.sagemath.org/17518 for details.
[2/17, 5, 0, 10]
```

complex_roots()

Return the complex roots of this polynomial, without multiplicities.

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

EXAMPLES:

```
sage: x = polygen(ZZ)
sage: (x^3 - 1).complex_roots()   # note: low order bits slightly different on ppc.
[1.00000000000000, -0.500000000000000 - 0.86602540378443...*I, -0.500000000000000 + 0.866025
```

TESTS:

```
sage: x = polygen(RR)
sage: (x^3 - 1).complex_roots()[0].parent()
Complex Field with 53 bits of precision
sage: x = polygen(RDF)
sage: (x^3 - 1).complex_roots()[0].parent()
Complex Double Field
sage: x = polygen(RealField(200))
sage: (x^3 - 1).complex_roots()[0].parent()
Complex Field with 200 bits of precision
sage: x = polygen(CDF)
sage: (x^3 - 1).complex_roots()[0].parent()
Complex Double Field
sage: x = polygen(ComplexField(200))
sage: (x^3 - 1).complex_roots()[0].parent()
Complex Field with 200 bits of precision
sage: x=polygen(ZZ,'x'); v=(x^2-x-1).complex_roots()
sage: v[0].parent() is CC
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()

Return the content of `self`, which is the ideal generated by the coefficients of `self`.

EXAMPLES:

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

degree(*gen=None*)

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

EXAMPLES:

```
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()
-1
sage: f = x + 33
sage: f.degree()
1
```

AUTHORS:

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

denominator()

Return a denominator of self.

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

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

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

EXAMPLES:

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

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

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

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

```
sage: f.denominator()
51
```

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

```
sage: R.<x> = RR[]
sage: f = x + RR('0.3'); f
x + 0.300000000000000
sage: f.denominator()
1.00000000000000
```

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

```
sage: R.<a> = GF(5)[]
sage: x = R(0)
sage: x.denominator()
1
sage: type(x.denominator())
<type 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
sage: isinstance(x.numerator() / x.denominator(), Polynomial)
True
sage: isinstance(x.numerator() / R(1), Polynomial)
False
```

derivative(**args*)

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

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

See also:

`_derivative()`

EXAMPLES:

```
sage: 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
```

discriminant()

Returns 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 $r_1, \ldots, r_n$.

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

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

EXAMPLES:

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

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

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

We can also compute discriminants over univariate and multivariate polynomial rings, provided that PARI's variable ordering requirements are respected. Usually, your discriminants will work if you always ask for them in the variable `x`:

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

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

```
sage: d.parent() is R
True
```

Unfortunately Sage does not handle PARI's variable ordering requirements gracefully, so the following has to be done through Singular:

```
sage: R.<x, y> = QQ[]
sage: S.<a> = R[]
sage: f = x^2 + a
sage: f.discriminant()
1
```

Check that trac ticket #13672 is fixed:

```
sage: R.<t> = GF(5)[]
sage: S.<x> = R[]
sage: f = x^10 + 2*x^6 + 2*x^5 + x + 2
sage: (f-t).discriminant()
4*t^5
```

The following examples show that trac ticket #11782 has been fixed:

```
sage: ZZ.quo(81)[x](3*x^2 + 3*x + 3).discriminant()
54
sage: ZZ.quo(9)[x](2*x^3 + x^2 + x).discriminant()
2
```

This was fixed by trac ticket #15422:

```
sage: R.<s> = PolynomialRing(Qp(2))
sage: (s^2).discriminant()
0
```

TESTS:

This was fixed by trac ticket #16014:

```
sage: PR.<b,t1,t2,x1,y1,x2,y2> = QQ[]
sage: PRmu.<mu> = PR[]
sage: E1 = diagonal_matrix(PR, [1, b^2, -b^2])
sage: M = matrix(PR, [[1,-t1,x1-t1*y1],[t1,1,y1+t1*x1],[0,0,1]])
sage: E1 = M.transpose()*E1*M
sage: E2 = E1.subs(t1=t2, x1=x2, y1=y2)
sage: det(mu*E1 + E2).discriminant().degrees()
(24, 12, 12, 8, 8, 8, 8)
```

This addresses an issue raised by trac ticket #15061:

```
sage: R.<T> = PowerSeriesRing(QQ)
sage: F = R([1,1],2)
sage: RP.<x> = PolynomialRing(R)
sage: P = x^2 - F
sage: P.discriminant()
4 + 4*T + O(T^2)
```

euclidean_degree()

Return the degree of this element as an element of a euclidean domain.

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

EXAMPLES:

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

exponents()

Return the exponents of the monomials appearing in self.

EXAMPLES:

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

factor(***kwargs*)

Return the factorization of `self` over its base ring.

INPUT:

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

OUTPUT:

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

EXAMPLES:

Factorization is implemented over various rings. Over **Q**:

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

Since **Q** is a field, the irreducible factors are monic:

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

Over **Z** the irreducible factors need not be monic:

```
sage: x = ZZ['x'].0
sage: f = 10*x^5 - 1
sage: f.factor()
10*x^5 - 1
```

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

```
sage: k.<a> = GF(25)
sage: R.<x> = k[]
sage: f = 2*x^10 + 2*x + 2*a
```

```
sage: F = f.factor(); F
(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
```

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

```
sage: expand(F)
2*x^10 + 2*x + 2*a
```

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

```
sage: R.<x> = PolynomialRing(IntegerModRing(4),implementation="NTL")
sage: (x^2).factor()
Traceback (most recent call last):
...
NotImplementedError: factorization of polynomials over rings with composite characteristic i
sage: R.base_ring()._factor_univariate_polynomial = lambda f: f.change_ring(ZZ).factor()
sage: (x^2).factor()
x^2
sage: del R.base_ring()._factor_univariate_polynomial # clean up
```

Arbitrary precision real and complex factorization:

```
sage: R.<x> = RealField(100)[]
sage: F = factor(x^2-3); F
(x - 1.7320508075688772935274463415) * (x + 1.7320508075688772935274463415)
sage: expand(F)
x^2 - 3.0000000000000000000000000000
sage: factor(x^2 + 1)
x^2 + 1.0000000000000000000000000000

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

Over a relative number field:

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

Over the real double field:

```
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 - 1.0000065719436413) * (x^2 - 1.9999934280563585*x + 0.9999934280995487)
```

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

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

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

```
sage: R.<x> = CDF[]
sage: f = (x^2 + 2*R(I))^3
sage: F = f.factor()
sage: F  # abs tol 3e-5
(x - 1.0000138879287663 + 1.0000013435286879*I) * (x - 0.9999942196864997 + 0.99998730098039
sage: [f(t[0][0]).abs() for t in F] # abs tol 1e-13
[1.979365054e-14, 1.97936298566e-14, 1.97936990747e-14, 3.6812407475e-14, 3.65211563729e-14,
```

Factoring polynomials over $\mathbf{Z}/n\mathbf{Z}$ for composite n is not implemented:

```
sage: R.<x> = PolynomialRing(Integers(35))
sage: f = (x^2+2*x+2)*(x^2+3*x+9)
sage: f.factor()
Traceback (most recent call last):
...
NotImplementedError: factorization of polynomials over rings with composite characteristic i
```

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

```
sage: R.<x> = QQbar[]
sage: (x^8-1).factor()
(x - 1) * (x - 0.7071067811865475? - 0.7071067811865475?*I) * (x - 0.7071067811865475? + 0.7
```

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

```
sage: R.<x> = AA[]
sage: (x^8+1).factor()
(x^2 - 1.847759065022574?*x + 1.000000000000000?) * (x^2 - 0.7653668647301795?*x + 1.0000000
```

TESTS:

This came up in ticket #7088:

```
sage: R.<x>=PolynomialRing(ZZ)
sage: f = 12*x^10 + x^9 + 432*x^3 + 9011
sage: g = 13*x^11 + 89*x^3 + 1
sage: F = f^2 * g^3
sage: F = f^2 * g^3; F.factor()
(12*x^10 + x^9 + 432*x^3 + 9011)^2 * (13*x^11 + 89*x^3 + 1)^3
sage: F = f^2 * g^3 * 7; F.factor()
7 * (12*x^10 + x^9 + 432*x^3 + 9011)^2 * (13*x^11 + 89*x^3 + 1)^3
```

This example came up in ticket #7097:

```
sage: x = polygen(QQ)
sage: f = 8*x^9 + 42*x^6 + 6*x^3 - 1
sage: g = x^24 - 12*x^23 + 72*x^22 - 286*x^21 + 849*x^20 - 2022*x^19 + 4034*x^18 - 6894*x^17
sage: assert g.is_irreducible()
sage: K.<a> = NumberField(g)
sage: len(f.roots(K))
9
sage: f.factor()
(8) * (x^3 + 1/4) * (x^6 + 5*x^3 - 1/2)
sage: f.change_ring(K).factor()
(8) * (x - 3260097/3158212*a^22 + 35861067/3158212*a^21 - 197810817/3158212*a^20 + 722970825
sage: f = QQbar['x'](1)
sage: f.factor()
1
```

Factorization also works even if the variable of the finite field is nefariously labeled “x”:

```
sage: R.<x> = GF(3^2, 'x')[]
sage: f = x^10 +7*x -13
sage: G = f.factor(); G
(x + x) * (x + 2*x + 1) * (x^4 + (x + 2)*x^3 + (2*x + 2)*x + 2) * (x^4 + 2*x*x^3 + (x + 1)*x
sage: prod(G) == f
True

sage: R.<x0> = GF(9,'x')[]  # purposely calling it x to test robustness
sage: f = x0^3 + x0 + 1
sage: f.factor()
(x0 + 2) * (x0 + x) * (x0 + 2*x + 1)
sage: f = 0*x0
sage: f.factor()
Traceback (most recent call last):
...
ValueError: factorization of 0 not defined

sage: f = x0^0
sage: f.factor()
1
```

Over a complicated number field:

```
sage: x = polygen(QQ, 'x')
sage: f = x^6 + 10/7*x^5 - 867/49*x^4 - 76/245*x^3 + 3148/35*x^2 - 25944/245*x + 48771/1225
sage: K.<a> = NumberField(f)
sage: S.<T> = K[]
sage: ff = S(f); ff
T^6 + 10/7*T^5 - 867/49*T^4 - 76/245*T^3 + 3148/35*T^2 - 25944/245*T + 48771/1225
sage: F = ff.factor()
sage: len(F)
4
```

```
sage: F[:2]
[(T - a, 1), (T - 40085763200/924556084127*a^5 - 145475769880/924556084127*a^4 + 52761709648
sage: expand(F)
T^6 + 10/7*T^5 - 867/49*T^4 - 76/245*T^3 + 3148/35*T^2 - 25944/245*T + 48771/1225

sage: f = x^2 - 1/3
sage: K.<a> = NumberField(f)
sage: A.<T> = K[]
sage: A(x^2 - 1).factor()
(T - 1) * (T + 1)

sage: A(3*x^2 - 1).factor()
(3) * (T - a) * (T + a)

sage: A(x^2 - 1/3).factor()
(T - a) * (T + a)
```

Test that ticket #10279 is fixed:

```
sage: R.<t> = PolynomialRing(QQ)
sage: K.<a> = NumberField(t^4 - t^2 + 1)
sage: pol = t^3 + (-4*a^3 + 2*a)*t^2 - 11/3*a^2*t + 2/3*a^3 - 4/3*a
sage: pol.factor()
(t - 2*a^3 + a) * (t - 4/3*a^3 + 2/3*a) * (t - 2/3*a^3 + 1/3*a)
```

Test that this factorization really uses `nffactor()` internally:

```
sage: pari.default("debug", 3)
sage: F = pol.factor()

Entering nffactor:
...
sage: pari.default("debug", 0)
```

Test that ticket #10369 is fixed:

```
sage: x = polygen(QQ)
sage: K.<a> = NumberField(x^6 + x^5 + x^4 + x^3 + x^2 + x + 1)
sage: R.<t> = PolynomialRing(K)

sage: pol = (-1/7*a^5 - 1/7*a^4 - 1/7*a^3 - 1/7*a^2 - 2/7*a - 1/7)*t^10 + (4/7*a^5 - 2/7*a^4
sage: pol.factor()
(-1/7*a^5 - 1/7*a^4 - 1/7*a^3 - 1/7*a^2 - 2/7*a - 1/7) * t * (t - a^5 - a^4 - a^3 - a^2 - a

sage: pol = (1/7*a^2 - 1/7*a)*t^10 + (4/7*a - 6/7)*t^9 + (102/49*a^5 + 99/49*a^4 + 96/49*a^3
sage: pol.factor()
(1/7*a^2 - 1/7*a) * t^5 * (t^5 + (-40/7*a^5 - 38/7*a^4 - 36/7*a^3 - 34/7*a^2 - 32/7*a - 30/7

sage: pol = x^10 + (4/7*a - 6/7)*x^9 + (9/49*a^2 - 3/7*a + 15/49)*x^8 + (8/343*a^3 - 32/343*
sage: pol.factor()
x^5 * (x^5 + (4/7*a - 6/7)*x^4 + (9/49*a^2 - 3/7*a + 15/49)*x^3 + (8/343*a^3 - 32/343*a^2 +
```

Factoring over a number field over which we cannot factor the discriminant by trial division:

```
sage: x = polygen(QQ)
sage: K.<a> = NumberField(x^16 - x - 6)
sage: R.<x> = PolynomialRing(K)
sage: f = (x+a)^50 - (a-1)^50
sage: len(factor(f))
6
```

```
sage: pari(K.discriminant()).factor(limit=0)
[-1, 1; 3, 15; 23, 1; 887, 1; 12583, 1; 2354691439917211, 1]
sage: factor(K.discriminant())
-1 * 3^15 * 23 * 887 * 12583 * 6335047 * 371692813
```

Factoring over a number field over which we cannot factor the discriminant and over which $nffactor()$ fails:

```
sage: p = next_prime(10^50); q = next_prime(10^51); n = p*q;
sage: K.<a> = QuadraticField(p*q)
sage: R.<x> = PolynomialRing(K)
sage: K.pari_polynomial('a').nffactor("x^2+1")
Mat([x^2 + 1, 1])
sage: factor(x^2 + 1)
x^2 + 1
sage: factor( (x - a) * (x + 2*a) )
(x - a) * (x + 2*a)
```

A test where nffactor used to fail without a nf structure:

```
sage: x = polygen(QQ)
sage: K = NumberField([x^2-1099511627777, x^3-3],'a')
sage: x = polygen(K)
sage: f = x^3 - 3
sage: factor(f)
(x - a1) * (x^2 + a1*x + a1^2)
```

We check that trac ticket #7554 is fixed:

```
sage: L.<q> = LaurentPolynomialRing(QQ)
sage: F = L.fraction_field()
sage: R.<x> = PolynomialRing(F)
sage: factor(x)
x
sage: factor(x^2 - q^2)
(-1) * (-x + q) * (x + q)
sage: factor(x^2 - q^-2)
(1/q^2) * (q*x - 1) * (q*x + 1)

sage: P.<a,b,c> = PolynomialRing(ZZ)
sage: R.<x> = PolynomialRing(FractionField(P))
sage: p = (x - a)*(b*x + c)*(a*b*x + a*c) / (a + 2)
sage: factor(p)
(a/(a + 2)) * (x - a) * (b*x + c)^2
```

gcd(*other*)

Compute 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. Over a field, the return value will be 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: R.<x> = QQ[]
sage: S.<y> = R[]
sage: h1 = y*x
sage: h2 = y^2*x^2
sage: h1.gcd(h2)
Traceback (most recent call last):
...
NotImplementedError: Univariate Polynomial Ring in x over Rational Field does not provide a
sage: T.<x,y> = QQ[]
sage: R._gcd_univariate_polynomial = lambda f,g: S(T(f).gcd(g))
sage: h1.gcd(h2)
x*y
sage: del R._gcd_univariate_polynomial
```

hamming_weight()

Returns the number of non-zero coefficients of self.

EXAMPLES:

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

homogenize(*var='h'*)

Return the homogenization of this polynomial.

The polynomial itself is returned if it 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:

```
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:

```
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:

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

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

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

In positive characterstic, 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 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.integral(); g
1/4*x^4 + 1/2*x^2 - 2*x
sage: g.parent()
Univariate Polynomial Ring in x over Rational Field
```

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

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

sage: P.<x,z> = PolynomialRing(GF(next_prime(2147483647)))
sage: Q.<y> = PolynomialRing(P)
sage: p=x+y+z
sage: p.integral()
1073741830*y^2 + (x + z)*y
```

A truly convoluted example:

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

Integration with respect to a variable in the base ring:

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

inverse_mod(*a*, *m*)

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

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: R.<x> = RDF[]
sage: epsilon = RDF(1).ulp()*50   # Allow an error of up to 50 ulp
sage: f = inverse_mod(x^2 + 1, x^5 + x + 1); f  # abs tol 1e-14
0.4*x^4 - 0.2*x^3 - 0.4*x^2 + 0.2*x + 0.8
sage: poly = f * (x^2 + 1) % (x^5 + x + 1)
sage: # Remove noisy zero terms:
sage: parent(poly)([ 0.0 if abs(c)<=epsilon else c for c in poly.coefficients(sparse=False)
1.0
sage: f = inverse_mod(x^3 - x + 1, x - 2); f
0.14285714285714285
sage: f * (x^3 - x + 1) % (x - 2)
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 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:

•Robert Bradshaw (2007-05-31)

inverse_of_unit()

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x - 90283
sage: f.inverse_of_unit()
```

```
Traceback (most recent call last):
...
ValueError: self is not a unit.
sage: f = R(-90283); g = f.inverse_of_unit(); g
-1/90283
sage: parent(g)
Univariate Polynomial Ring in x over Rational Field
```

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*)

Return `True` if `self` is a cyclotomic polynomial. If `certificate` is `True`, the result is `0` if `self` is not cyclotomic, and `n` if `self` is the n-th cyclotomic polynomial.

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

Todo

Calling `poliscyclo()` from libpari would be much faster. See trac ticket #17730.

ALGORITHM:

The first cyclotomic polynomial `x-1` is treated apart, otherwise the first algorithm of [BD89] is used.

If `certificate` is `True`, the function `poliscyclo` of GP is called.

EXAMPLES:

Quick tests:

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

Test first 100 cyclotomic polynomials:

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

Some more tests:

```
sage: (x^16 + x^14 - x^10 + x^8 - x^6 + x^2 + 1).is_cyclotomic()
False
sage: (x^16 + x^14 - x^10 - x^8 - x^6 + x^2 + 1).is_cyclotomic()
True
sage: y = polygen(QQ)
sage: (y/2 - 1/2).is_cyclotomic()
False
sage: (2*(y/2 - 1/2)).is_cyclotomic()
True
```

Test using other rings:

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

REFERENCES:

is_gen()

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

EXAMPLES:

```
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.

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

is_homogeneous()

Return `True` if this polynomial is homogeneous.

EXAMPLES:

```
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 True precisely if this polynomial is irreducible over its base ring.

Testing irreducibility over $\mathbf{Z}/n\mathbf{Z}$ for composite n is not implemented.

EXAMPLES:

```
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
```

See trac ticket #5140,

```
sage: R(1).is_irreducible()
False
sage: R(4).is_irreducible()
False
sage: R(5).is_irreducible()
True
```

The base ring does matter: for example, 2x is irreducible as a polynomial in QQ[x], but not in ZZ[x],

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

TESTS:

```
sage: F.<t> = NumberField(x^2-5)
sage: Fx.<xF> = PolynomialRing(F)
sage: f = Fx([2*t - 5, 5*t - 10, 3*t - 6, -t, -t + 2, 1])
sage: f.is_irreducible()
False
sage: f = Fx([2*t - 3, 5*t - 10, 3*t - 6, -t, -t + 2, 1])
sage: f.is_irreducible()
True
```

is_monic()

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

EXAMPLES:

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

AUTHORS:

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

is_monomial()

Returns 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
sage: f^2
0
```

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

is_primitive(*n=None*, *n_prime_divs=None*)

Returns `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 $\mathbf{F}_q$ is primitive if it is irreducible and its root in $\mathbf{F}_{q^m}$ generates the multiplicative group $\mathbf{F}_{q^m}^*$.
- (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).

INPUTS:

- `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.

::

  sage: R.<x> = GF(2)['x']
  sage: f = x^4+x^3+x^2+x+1
  sage: f.is_irreducible(), f.is_primitive()
  (True, False)
  sage: f = x^3+x+1
  sage: f.is_irreducible(), f.is_primitive()
  (True, True)
  sage: R.<x> = GF(3)[]
  sage: f = x^3-x+1
  sage: f.is_irreducible(), f.is_primitive()
  (True, True)
  sage: f = x^2+1
  sage: f.is_irreducible(), f.is_primitive()
  (True, False)
  sage: R.<x> = GF(5)[]
  sage: f = x^2+x+1
  sage: f.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: is_primitive() not defined for polynomials over infinite fields.

Ring semantics examples.

::

  sage: x=polygen(ZZ)
  sage: f = 5*x^2+2
  sage: f.is_primitive()
  True
  sage: f = 5*x^2+5
  sage: f.is_primitive()
  False

  sage: K=NumberField(x^2+5,'a')
```

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

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

TESTS:

```
sage: R.<x> = GF(2)['x']
sage: f = x^4+x^3+x^2+x+1
sage: f.is_primitive(15)
False
sage: f.is_primitive(15, [3,5])
False
sage: f.is_primitive(n_prime_divs=[3,5])
False
sage: f = x^3+x+1
sage: f.is_primitive(7, [7])
True
sage: R.<x> = GF(3)[]
sage: f = x^3-x+1
sage: f.is_primitive(26, [2,13])
True
sage: f = x^2+1
sage: f.is_primitive(8, [2])
False
sage: R.<x> = GF(5)[]
sage: f = x^2+x+1
sage: f.is_primitive(24, [2,3])
False
sage: f = x^2-x+2
sage: f.is_primitive(24, [2,3])
True
sage: x=polygen(Integers(103)); f=x^2+1
sage: f.is_primitive()
False
```

is_square(*root=False*)

Returns 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:

```
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

sage: g.is_square()
True
```

TESTS:

Make sure trac ticket #9093 is fixed:

```
sage: R(1).is_square()
True
sage: R(4/9).is_square(root=True)
(True, 2/3)
sage: R(-1/3).is_square()
False
sage: R(0).is_square()
True
```

is_squarefree()

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

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

EXAMPLES:

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

A generic implementation is available for polynomials defined over principal ideal domains of characteristic 0; the algorithm relies on gcd computations:

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

sage: S.<y> = QQ[]
sage: R.<x> = S[]
sage: (2*x*y).is_squarefree() # R does not provide a gcd implementation
Traceback (most recent call last):
...
NotImplementedError: Univariate Polynomial Ring in y over Rational Field does not provide a
sage: (2*x*y^2).is_squarefree()
False
```

Over principal ideal domains of positive characteristic, we compute the square-free decomposition or a full factorization depending on which is available:

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

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()
False
sage: f.squarefree_decomposition()
(4) * x
```

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

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

is_term()

Return True if self is an element of the base ring times a power of the generator.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: x.is_term()
True
```

```
sage: R(1).is_term()
True
sage: (3*x^5).is_term()
True
sage: (1+3*x^5).is_term()
False
```

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

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

is_unit()

Return True if this polynomial is a unit.

EXAMPLES:

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

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

lcm(*other*)

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

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()

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)
<type '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)
<type 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
sage: v = f.list(); v
[-3*x, x, 0, 1]
sage: v[0] = 10
sage: f.list()
[-3*x, x, 0, 1]
```

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

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

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

INPUT:

•`f` – a callable that will be applied to the coefficients of self.

•`new_base_ring` (optional) – if given, the resulting polynomial will be defined over this ring.

EXAMPLES:

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

Examples with different base ring:

```
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 NTL)
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 NTL)
```

mod(*other*)

Remainder of division of self by other.

EXAMPLES:

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

monic()

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

EXAMPLES:

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

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

```
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

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 x0.

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

If one of the iterates is a critical point of f then a ZeroDivisionError exception is raised.

EXAMPLES:

```
sage: x = PolynomialRing(RealField(), 'x').gen()
sage: f = x^2 - 2
sage: f.newton_raphson(4, 1)
[1.50000000000000, 1.41666666666667, 1.41421568627451, 1.41421356237469]
```

AUTHORS:

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

newton_slopes(*p*)

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

OUTPUT: list of rational numbers

EXAMPLES:

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

ALGORITHM: Uses PARI.

norm(*p*)

Return the p-norm of this polynomial.

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

INPUT:

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

EXAMPLES:

```
sage: R.<x> = RR[]
sage: f = x^6 + x^2 + -x^4 - 2*x^3
sage: f.norm(2)
2.64575131106459
sage: (sqrt(1^2 + 1^2 + (-1)^2 + (-2)^2)).n()
2.64575131106459

sage: f.norm(1)
5.00000000000000
sage: f.norm(infinity)
2.00000000000000

sage: f.norm(-1)
Traceback (most recent call last):
...
ValueError: The degree of the norm must be positive
```

TESTS:

```
sage: R.<x> = RR[]
sage: f = x^6 + x^2 + -x^4 -x^3
sage: f.norm(int(2))
2.00000000000000
```

AUTHORS:

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

numerator()

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

Note that some subclases 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.

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

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

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

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

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

We check that the computation the numerator and denominator are valid

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

ord(*p=None*)

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

EXAMPLES:

```
sage: P,x=PolynomialRing(ZZ,'x').objgen()
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:

```
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:

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

polynomial(*var*)

Let 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:

```
sage: R.<x> = QQ[]
sage: (x+1).polynomial(x)
x + 1
```

TESTS:

```
sage: x.polynomial(1)
Traceback (most recent call last):
...
ValueError: given variable is not the generator of parent.
```

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:

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

radical()

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

EXAMPLES:

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

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

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

real_roots()

Return the real roots of this polynomial, without multiplicities.

Calls self.roots(ring=RR), unless this is a polynomial with floating-point real coefficients, in which case it calls self.roots().

EXAMPLES:

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

TESTS:

```
sage: x = polygen(RealField(100))
sage: (x^2 - x - 1).real_roots()[0].parent()
    Real Field with 100 bits of precision
sage: x = polygen(RDF)
sage: (x^2 - x - 1).real_roots()[0].parent()
Real Double Field

sage: x=polygen(ZZ,'x'); v=(x^2-x-1).real_roots()
sage: v[0].parent() is RR
True
```

resultant (*other*)

Returns the resultant of self and other.

INPUT:

- `other` - a polynomial

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

Note: Implemented using PARI's `polresultant` function.

EXAMPLES:

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

We can also compute resultants over univariate and multivariate polynomial rings, provided that PARI's variable ordering requirements are respected. Usually, your resultants will work if you always ask for them in the variable `x`:

```
sage: R.<a> = QQ[]
sage: S.<x> = R[]
sage: f = x^2 + a; g = x^3 + a
sage: r = f.resultant(g); r
a^3 + a^2
sage: r.parent() is R
True

sage: R.<a, b> = QQ[]
sage: S.<x> = R[]
sage: f = x^2 + a; g = x^3 + b
sage: r = f.resultant(g); r
a^3 + b^2
sage: r.parent() is R
True
```

Unfortunately Sage does not handle PARI's variable ordering requirements gracefully, so the following has to be done through Singular:

```
sage: R.<x, y> = QQ[]
sage: S.<a> = R[]
sage: f = x^2 + a; g = y^3 + a
sage: h = f.resultant(g); h
y^3 - x^2
sage: h.parent() is R
```

```
True
```

Check that trac ticket #13672 is fixed:

```
sage: R.<t> = GF(2)[]
sage: S.<x> = R[]
sage: f = (t^2 + t)*x + t^2 + t
sage: g = (t + 1)*x + t^2
sage: f.resultant(g)
t^4 + t
```

Check that trac ticket #15061 is fixed:

```
sage: R.<T> = PowerSeriesRing(QQ)
sage: F = R([1,1],2)
sage: RP.<x> = PolynomialRing(R)
sage: P = x^2 - F
sage: P.resultant(P.derivative())
-4 - 4*T + O(T^2)
```

Check that trac ticket #16360 is fixed:

```
sage: K.<x> = FunctionField(QQ)
sage: R.<y> = K[]
sage: y.resultant(y+x)
x

sage: K.<a> = FunctionField(QQ)
sage: R.<b> = K[]
sage: L.<b> = K.extension(b^2-a)
sage: R.<x> = L[]
sage: f=x^2-a
sage: g=x-b
sage: f.resultant(g)
0
```

reverse(*degree=None*)

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 and the reverse polynomial will have the specified degree.

EXAMPLES:

```
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
```

TESTS:

```
sage: f.reverse(degree=1.5r)
Traceback (most recent call last):
...
ValueError: degree argument must be a non-negative integer, got 1.5
```

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

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

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

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

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

sage: R.<x> = QQ[]
sage: f = x^2
sage: K.<alpha> = f.root_field()
Traceback (most recent call last):
...
ValueError: polynomial must be irreducible
```

TESTS:

```
sage: (PolynomialRing(Integers(31),name='x').0+5).root_field('a')
Ring of integers modulo 31
```

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

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

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

EXAMPLES:

```
sage: x = QQ['x'].0
sage: f = x^3 - 1
sage: f.roots()
[(1, 1)]
sage: f.roots(ring=CC)   # note -- low order bits slightly different on ppc.
[(1.00000000000000, 1), (-0.500000000000000 - 0.86602540378443...*I, 1), (-0.500000000000000
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: 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()
[]
sage: g = lambda f, *args, **kwds: f.change_ring(CDF).roots()
```

```
sage: QQ._roots_univariate_polynomial = g
sage: (x^2 + 1).roots()  # abs tol 1e-14
[(2.7755575615628914e-17 - 1.0*I, 1), (0.9999999999999997*I, 1)]
sage: del QQ._roots_univariate_polynomial
```

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

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

sage: x = CC['x'].0
sage: f = x^3 -2
sage: f.roots()
[(1.25992104989487, 1), (-0.62996052494743... - 1.09112363597172*I, 1), (-0.62996052494743..
sage: f.roots(algorithm='pari')
[(1.25992104989487, 1), (-0.629960524947437 - 1.09112363597172*I, 1), (-0.629960524947437 +
```

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

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

An example involving large numbers:

```
sage: x = RR['x'].0
sage: f = x^2 - 1e100
sage: f.roots()
[(-1.00000000000000e50, 1), (1.00000000000000e50, 1)]
sage: f = x^10 - 2*(5*x-1)^2
sage: f.roots(multiplicities=False)
[-1.6772670339941..., 0.19995479628..., 0.20004530611..., 1.5763035161844...]

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:

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

```
[-I*sqrt(2), I*sqrt(2)]
```

The roots of some polynomials can’t be described using radical expressions:

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

For some other polynomials, no roots can be found at the moment due to the way roots are computed. trac ticket #17516 addresses these defecits. 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()
[]
```

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
sage: f.roots()
[(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 -
sage: f.roots()
[(I, 3), (-2^(1/4), 1), (2^(1/4), 1), (1, 1)]
sage: f.roots(multiplicities=False)
[I, -2^(1/4), 2^(1/4), 1]
```

A couple of examples where the base ring doesn’t have a factorization algorithm (yet). Note that this is currently done via 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 (t
sage: p.roots(multiplicities=False)
[1, 5]
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)
[0, 1, 3, 6]
```

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

```
sage: R.<x> = CDF[]
sage: f = R.cyclotomic_polynomial(5); f
x^4 + x^3 + x^2 + x + 1.0
sage: f.roots(multiplicities=False)  # abs tol 1e-9
[-0.8090169943749469 - 0.5877852522924724*I, -0.8090169943749473 + 0.5877852522924724*I, 0.3
sage: [z^5 for z in f.roots(multiplicities=False)]  # abs tol 1e-14
[0.9999999999999957 - 1.2864981197413038e-15*I, 0.9999999999999976 + 3.062854959141552e-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
[2.574630599127759e-15, 1.457101633618084e-15, 1.1443916996305594e-15]
```

Another example over RDF:

```
sage: x = RDF['x'].0
sage: ((x^3 -1)).roots()  # abs tol 4e-16
[(1.0000000000000002, 1)]
sage: ((x^3 -1)).roots(multiplicities=False)  # abs tol 4e-16
[1.0000000000000002]
```

More examples involving the complex double field:

```
sage: x = CDF['x'].0
sage: i = CDF.0
sage: f = x^3 + 2*i; f
x^3 + 2.0*I
sage: f.roots()  # abs tol 1e-14
[(-1.0911236359717227 - 0.6299605249474374*I, 1), (3.885780586188048e-16 + 1.259921049894873
sage: f.roots(multiplicities=False)  # abs tol 1e-14
[-1.0911236359717227 - 0.6299605249474374*I, 3.885780586188048e-16 + 1.2599210498948734*I, 1
sage: [abs(f(z)) for z in f.roots(multiplicities=False)]  # abs tol 1e-14
[8.95090418262362e-16, 8.728374398092689e-16, 1.0235750533041806e-15]
sage: f = i*x^3 + 2; f
I*x^3 + 2.0
sage: f.roots()  # abs tol 1e-14
[(-1.0911236359717227 + 0.6299605249474374*I, 1), (3.885780586188048e-16 - 1.259921049894873
sage: f(f.roots()[0][0])  # abs tol 1e-13
3.3306690738754696e-15 + 1.3704315460216776e-15*I
```

Examples using real root isolation:

```
sage: x = polygen(ZZ)
sage: f = x^2 - x - 1
sage: f.roots()
[]
sage: f.roots(ring=RIF)
[(-0.6180339887498948482045868343657?, 1), (1.6180339887498948482045868343657?, 1)]
sage: f.roots(ring=RIF, multiplicities=False)
[-0.6180339887498948482045868343657?, 1.6180339887498948482045868343657?]
sage: f.roots(ring=RealIntervalField(150))
[(-0.6180339887498948482045868343656381177203091798057628621354486227?, 1), (1.6180339887498
sage: f.roots(ring=AA)
[(-0.618033988749895?, 1), (1.618033988749895?, 1)]
sage: f = f^2 * (x - 1)
sage: f.roots(ring=RIF)
[(-0.6180339887498948482045868343657?, 2), (1.0000000000000000000000000000000?, 1), (1.61803
sage: f.roots(ring=RIF, multiplicities=False)
[-0.6180339887498948482045868343657?, 1.0000000000000000000000000000000?, 1.6180339887498948
```

Examples using complex root isolation:

```
sage: x = polygen(ZZ)
sage: p = x^5 - x - 1
sage: p.roots()
[]
sage: p.roots(ring=CIF)
```

```
[(1.167303978261419?, 1), (-0.764884433600585? - 0.352471546031727?*I, 1), (-0.7648844336005
sage: p.roots(ring=ComplexIntervalField(200))
[(1.167303978261418684256045899854842180720560371525489039140082?, 1), (-0.76488443360058472
sage: rts = p.roots(ring=QQbar); rts
[(1.167303978261419?, 1), (-0.7648844336005847? - 0.3524715460317263?*I, 1), (-0.76488443360
sage: p.roots(ring=AA)
[(1.167303978261419?, 1)]
sage: p = (x - rts[4][0])^2 * (3*x^2 + x + 1)
sage: p.roots(ring=QQbar)
[(-0.1666666666666667? - 0.552770798392567?*I, 1), (-0.1666666666666667? + 0.552770798392567
sage: p.roots(ring=CIF)
[(-0.1666666666666667? - 0.552770798392567?*I, 1), (-0.1666666666666667? + 0.552770798392567
```

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

```
sage: K.<im> = NumberField(x^2 + 1)
sage: y = polygen(K)
sage: p = y^4 - 2 - im
sage: p.roots(ring=CC)
[(-1.2146389322441... - 0.14142505258239...*I, 1), (-0.14142505258239... + 1.2146389322441..
sage: p = p^2 * (y^2 - 2)
sage: p.roots(ring=CIF)
[(-1.414213562373095?, 1), (1.414213562373095?, 1), (-1.214638932244183? - 0.141425052582394
```

Note that one should not use NumPy when wanting high precision output as it does not support any of the high precision types:

```
sage: R.<x> = RealField(200)[]
sage: f = x^2 - R(pi)
sage: f.roots()
[(-1.7724538509055160272981674833411451827975494561223871282138, 1), (1.77245385090551602729
sage: f.roots(algorithm='numpy')
doctest... UserWarning: NumPy does not support arbitrary precision arithmetic.  The roots fo
[(-1.77245385090551..., 1), (1.77245385090551..., 1)]
```

We can also find roots over number fields:

```
sage: K.<z> = CyclotomicField(15)
sage: R.<x> = PolynomialRing(K)
sage: (x^2 + x + 1).roots()
[(z^5, 1), (-z^5 - 1, 1)]
```

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

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

```
....:             print 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 [1.25992104989...]
Real Field with 100 bits of precision [1.2599210498948731647672106073]
Complex Field with 53 bits of precision [1.25992104989487, -0.62996052494743... - 1.09112363
Complex Double Field [1.25992104989..., -0.629960524947... - 1.0911236359717...*I, -0.629960
Complex Field with 100 bits of precision [1.2599210498948731647672106073, -0.629960524947436
```

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

```
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
[(1.414213562373095?, 1), (1.732050807568878?, 1)]
sage: rts[0][0] == rt2
True
sage: f.roots(ring=RealIntervalField(150))
[(1.414213562373095048801688724209698078569671875376948073176679738?, 1), (1.732050807568877
```

We can handle polynomials with huge coefficients.

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

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

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

```
sage: x = polygen(QQ)
sage: p = x + bigc
sage: p.roots(ring=RR, algorithm='numpy')
Traceback (most recent call last):
...
LinAlgError: Array must not contain infs or NaNs
sage: p.roots(ring=RR, algorithm='pari')
[(-3.50746621104340e451, 1)]
sage: p.roots(ring=AA)
[(-3.5074662110434039?e451, 1)]
sage: p.roots(ring=QQbar)
[(-3.5074662110434039?e451, 1)]
sage: p = bigc*x + 1
sage: p.roots(ring=RR)
[(0.000000000000000, 1)]
sage: p.roots(ring=AA)
[(-2.8510609648967059?e-452, 1)]
sage: p.roots(ring=QQbar)
[(-2.8510609648967059?e-452, 1)]
sage: p = x^2 - bigc
sage: p.roots(ring=RR)
[(-5.92238652153286e225, 1), (5.92238652153286e225, 1)]
```

```
sage: p.roots(ring=QQbar)
[(-5.9223865215328558?e225, 1), (5.9223865215328558?e225, 1)]
```

Algorithms used:

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

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

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

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

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

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

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

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

For all other cases where K is different than L, we just use .change_ring(L) and proceed as below.

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

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

Note: We mentioned above that polynomials with multiple roots are always ill-conditioned; if your input is given to n bits of precision, you should not expect more than n/k good bits for a k-fold root. (You can get solutions that make the polynomial evaluate to a number very close to zero; basically the problem is that with a multiple root, there are many such numbers, and it's difficult to choose between them.)

To see why this is true, consider the naive floating-point error analysis model where you just pretend that all floating-point numbers are somewhat imprecise - a little 'fuzzy', if you will. Then the graph of a floating-point polynomial will be a fuzzy line. Consider the graph of $(x-1)^3$; this will be a fuzzy line with a horizontal tangent at $x = 1$, $y = 0$. If the fuzziness extends up and down by about j, then it will extend left and right by about cube_root(j).

TESTS:

```
sage: K.<zeta> = CyclotomicField(2)
sage: R.<x> = K[]
sage: factor(x^3-1)
(x - 1) * (x^2 + x + 1)
```

This shows that the issue from trac ticket #6237 is fixed:

```
sage: R.<u> = QQ[]
sage: g = -27*u^14 - 32*u^9
sage: g.roots(CDF, multiplicities=False)  # abs tol 2e-15
[-1.0345637159435719, 0.0, -0.3196977699902601 - 0.9839285635706636*I, -0.3196977699902601 +
sage: g.roots(CDF)  # abs tol 2e-15
[(-1.0345637159435719, 1), (0.0, 9), (-0.3196977699902601 - 0.9839285635706636*I, 1), (-0.31
```

This shows that the issue at trac ticket #2418 is fixed:

```
sage: x = polygen(QQ)
sage: p = (x^50/2^100 + x^10 + x + 1).change_ring(ComplexField(106))
sage: rts = (p/2^100).roots(multiplicities=False)
sage: eps = 2^(-50)   # we test the roots numerically
sage: [abs(p(rt)) < eps for rt in rts] == [True]*50
True
```

This shows that the issue at trac ticket #10901 is fixed:

```
sage: a = var('a'); R.<x> = SR[]
sage: f = x - a
sage: f.roots(RR)
Traceback (most recent call last):
...
TypeError: Cannot evaluate symbolic expression to a numeric value.
sage: f.roots(CC)
Traceback (most recent call last):
...
TypeError: Cannot evaluate symbolic expression to a numeric value.
```

We can find roots of polynomials defined over $\mathbf{Z}$ or $\mathbf{Q}$ over the p-adics, see trac ticket #15422:

```
sage: R.<x> = ZZ[]
sage: pol = (x - 1)^2
sage: pol.roots(Qp(3,5))
[(1 + O(3^5), 2)]
```

This doesn't work if we first change coefficients to $\mathbf{Q}_p$:

```
sage: pol.change_ring(Qp(3,5)).roots()
Traceback (most recent call last):
...
PrecisionError: p-adic factorization not well-defined since the discriminant is zero up to t

sage: (pol - 3^6).roots(Qp(3,5))
[(1 + 2*3^3 + 2*3^4 + O(3^5), 1), (1 + 3^3 + O(3^5), 1)]
sage: r = pol.roots(Zp(3,5), multiplicities=False); r
[1 + O(3^5)]
sage: parent(r[0])
3-adic Ring with capped relative precision 5
```

Spurious crash with pari-2.5.5, see trac ticket #16165:

```
sage: f=(1+x+x^2)^3
sage: f.roots(ring=CC)
[(-0.500000000000000 - 0.866025403784439*I, 3),
 (-0.500000000000000 + 0.866025403784439*I, 3)]
```

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 (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
```

TESTS:

```
sage: p = R(0)
sage: p.shift(3).is_zero()
True
sage: p.shift(-3).is_zero()
True
```

AUTHORS:

- David Harvey (2006-08-06)
- Robert Bradshaw (2007-04-18): Added support for infix operator.

splitting_field(*names*, *map=False*, ***kwds*)

Compute the absolute splitting field of a given polynomial.

INPUT:

- `names` – 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
sage: K.<a> = (x^3 - 3*x + 1).splitting_field(); K
Number Field in a with defining polynomial x^3 - 3*x + 1
```

Relative situation:

```
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)
sage: phi
Ring morphism:
  From: Number Field in a with defining polynomial x^3 + 2
  To:   Number Field in b with defining polynomial t^6 + 2
  Defn: a |--> b^2
```

An example over a finite field:

```
sage: P.<x> = PolynomialRing(GF(7))
sage: t = x^2 + 1
sage: t.splitting_field('b')
Finite Field in b of size 7^2

sage: P.<x> = PolynomialRing(GF(7^3, 'a'))
sage: t = x^2 + 1
sage: t.splitting_field('b', map=True)
(Finite Field in b of size 7^6,
 Ring morphism:
   From: Finite Field in a of size 7^3
   To:   Finite Field in b of size 7^6
   Defn: a |--> 2*b^4 + 6*b^3 + 2*b^2 + 3*b + 2)
```

If the extension is trivial and the generators have the same name, the map will be the identity:

```
sage: t = 24*x^13 + 2*x^12 + 14
sage: t.splitting_field('a', map=True)
(Finite Field in a of size 7^3,
 Identity endomorphism of Finite Field in a of size 7^3)

sage: t = x^56 - 14*x^3
sage: t.splitting_field('b', map=True)
(Finite Field in b of size 7^3,
 Ring morphism:
 From: Finite Field in a of size 7^3
   To:   Finite Field in b of size 7^3
   Defn: a |--> b)
```

See also:

`sage.rings.number_field.splitting_field.splitting_field()` for more examples over number fields

TESTS:

```
sage: K.<a,b> = x.splitting_field()
Traceback (most recent call last):
...
IndexError: the number of names must equal the number of generators
sage: polygen(RR).splitting_field('x')
Traceback (most recent call last):
...
NotImplementedError: splitting_field() is only implemented over number fields and finite fie

sage: P.<x> = PolynomialRing(GF(11^5, 'a'))
sage: t = x^2 + 1
sage: t.splitting_field('b')
Finite Field in b of size 11^10
sage: t = 24*x^13 + 2*x^12 + 14
sage: t.splitting_field('b')
Finite Field in b of size 11^30
sage: t = x^56 - 14*x^3
sage: t.splitting_field('b')
Finite Field in b of size 11^130

sage: P.<x> = PolynomialRing(GF(19^6, 'a'))
sage: t = -x^6 + x^2 + 1
sage: t.splitting_field('b')
Finite Field in b of size 19^6
sage: t = 24*x^13 + 2*x^12 + 14
sage: t.splitting_field('b')
Finite Field in b of size 19^18
sage: t = x^56 - 14*x^3
sage: t.splitting_field('b')
Finite Field in b of size 19^156

sage: P.<x> = PolynomialRing(GF(83^6, 'a'))
sage: t = 2*x^14 - 5 + 6*x
sage: t.splitting_field('b')
Finite Field in b of size 83^84
sage: t = 24*x^13 + 2*x^12 + 14
sage: t.splitting_field('b')
Finite Field in b of size 83^78
sage: t = x^56 - 14*x^3
sage: t.splitting_field('b')
Finite Field in b of size 83^12

sage: P.<x> = PolynomialRing(GF(401^13, 'a'))
sage: t = 2*x^14 - 5 + 6*x
sage: t.splitting_field('b')
Finite Field in b of size 401^104
sage: t = 24*x^13 + 2*x^12 + 14
sage: t.splitting_field('b')
Finite Field in b of size 401^156
sage: t = x^56 - 14*x^3
sage: t.splitting_field('b')
Finite Field in b of size 401^52
```

square()

Returns 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? ring_element or something?

AUTHORS:

- David Harvey (2006-09-09)

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^3 + 1
sage: f.square()
x^6 + 2*x^3 + 1
sage: f*f
x^6 + 2*x^3 + 1
```

squarefree_decomposition()

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

EXAMPLES:

```
sage: x = polygen(QQ)
sage: p = 37 * (x-1)^3 * (x-2)^3 * (x-1/3)^7 * (x-3/7)
sage: p.squarefree_decomposition()
(37*x - 111/7) * (x^2 - 3*x + 2)^3 * (x - 1/3)^7
sage: p = 37 * (x-2/3)^2
sage: p.squarefree_decomposition()
(37) * (x - 2/3)^2
sage: x = polygen(GF(3))
sage: x.squarefree_decomposition()
x
sage: f = QQbar['x'](1)
sage: f.squarefree_decomposition()
1
```

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

Identical to self(*x).

See the docstring for `self.__call__`.

EXAMPLES:

```
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 `self.__call__`.

EXAMPLES:

```
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*)

Returns 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)
sage: M
[  42  317  134 1786    0    0    0]
[   0   42  317  134 1786    0    0]
[   0    0   42  317  134 1786    0]
[   0    0    0   42  317  134 1786]
[  18  141   12  100   47    0    0]
[   0   18  141   12  100   47    0]
[   0    0   18  141   12  100   47]
```

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

```
sage: M.determinant()
0
```

If self and right are polynomials of positive degree, the determinant of the Sylvester matrix is the resultant of the polynomials.:

```
sage: h1 = R.random_element()
sage: h2 = R.random_element()
sage: M1 = h1.sylvester_matrix(h2)
sage: M1.determinant() == h1.resultant(h2)
True
```

The rank of the Sylvester matrix is related to the degree of the gcd of self and right:

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

TESTS:

The variable is optional, but must be the same in both rings:

```
sage: K.<x> = QQ['x']
sage: f = x+1
sage: g = QQ['y']([1, 0, 1])
sage: f.sylvester_matrix(f, x)
[1 1]
[1 1]
sage: f.sylvester_matrix(g, x)
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents: 'Univariate Polynomial Ring
```

Polynomials must be defined over compatible base rings:

```
sage: f = QQ['x']([1, 0, 1])
sage: g = ZZ['x']([1, 0, 1])
sage: h = GF(25, 'a')['x']([1, 0, 1])
sage: f.sylvester_matrix(g)
[1 0 1 0]
[0 1 0 1]
[1 0 1 0]
[0 1 0 1]
sage: g.sylvester_matrix(h)
[1 0 1 0]
[0 1 0 1]
[1 0 1 0]
[0 1 0 1]
sage: f.sylvester_matrix(h)
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents: 'Univariate Polynomial Ring
```

We can compute the sylvester matrix of a univariate and multivariate polynomial:

```
sage: K.<x,y> = QQ['x,y']
sage: g = K.random_element()
sage: f.sylvester_matrix(g) == K(f).sylvester_matrix(g,x)
True
```

Corner cases:

```
sage: K.<x>=QQ[]
sage: f = x^2+1
sage: g = K(0)
sage: f.sylvester_matrix(g)
Traceback (most recent call last):
...
ValueError: The Sylvester matrix is not defined for zero polynomials
sage: g.sylvester_matrix(f)
```

```
Traceback (most recent call last):
...
ValueError: The Sylvester matrix is not defined for zero polynomials
sage: g.sylvester_matrix(g)
Traceback (most recent call last):
...
ValueError: The Sylvester matrix is not defined for zero polynomials
sage: K(3).sylvester_matrix(x^2)
[3 0]
[0 3]
sage: K(3).sylvester_matrix(K(4))
[]
```

truncate(*n*)

Returns the polynomial of degree ' < n' which is equivalent to self modulo x^n.

EXAMPLES:

```
sage: R.<x> = ZZ[]; S.<y> = PolynomialRing(R, sparse=True)
sage: f = y^3 + x*y -3*x; f
y^3 + x*y - 3*x
sage: f.truncate(2)
x*y - 3*x
sage: f.truncate(1)
-3*x
sage: f.truncate(0)
0
```

valuation(*p=None*)

If $f = a_r x^r + a_{r+1}x^{r+1} + \cdots$, with a_r nonzero, then the valuation of f is r. The valuation of the zero polynomial is ∞.

If a prime (or non-prime) p is given, then the valuation is the largest power of p which divides self.

The valuation at ∞ is -self.degree().

EXAMPLES:

```
sage: P,x=PolynomialRing(ZZ,'x').objgen()
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()

Returns 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()
()
```

class sage.rings.polynomial.polynomial_element.**PolynomialBaseringInjection**

Bases: sage.categories.morphism.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 polynomial type.

Technically, it should be a method of the polynomial ring, but few polynomial rings are cython classes, and so, as a method of a cython polynomial class, it is faster.

EXAMPLES:

We demonstrate that most polynomial ring classes use polynomial base injection maps for coercion. They are supposed to be the fastest maps for that purpose. See trac ticket #9944.

```
sage: R.<x> = Qp(3)[]
sage: R.coerce_map_from(R.base_ring())
Polynomial base injection morphism:
  From: 3-adic Field with capped relative precision 20
  To:   Univariate Polynomial Ring in x over 3-adic Field with capped relative precision 20
sage: R.<x,y> = Qp(3)[]
sage: R.coerce_map_from(R.base_ring())
Polynomial base injection morphism:
  From: 3-adic Field with capped relative precision 20
  To:   Multivariate Polynomial Ring in x, y over 3-adic Field with capped relative precision 20
sage: R.<x,y> = QQ[]
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 trac ticket #9944, there are now only very few exceptions:

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

section()

TESTS:

```
sage: from sage.rings.polynomial.polynomial_element import PolynomialBaseringInjection
sage: m = PolynomialBaseringInjection(RDF, RDF['x'])
sage: m.section()
Generic map:
  From: Univariate Polynomial Ring in x over Real Double Field
  To:   Real Double Field
sage: type(m.section())
<type 'sage.rings.polynomial.polynomial_element.ConstantPolynomialSection'>
```

class sage.rings.polynomial.polynomial_element.**Polynomial_generic_dense**

Bases: sage.rings.polynomial.polynomial_element.Polynomial

A generic dense polynomial.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(PolynomialRing(QQ,'y'))
sage: f = x^3 - x + 17
sage: type(f)
<type 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
sage: loads(f.dumps()) == f
True
```

constant_coefficient()

Return the constant coefficient of this polynomial.

OUTPUT: element of base ring

EXAMPLES: sage: R.<t> = QQ[] sage: S.<x> = R[] sage: f = x*t + x + t sage: f.constant_coefficient() t

degree(*gen=None*)

EXAMPLES:

```
sage: R.<x> = RDF[]
sage: f = (1+2*x^7)^5
sage: f.degree()
35
```

TESTS:

Check that trac ticket #12552 is fixed:

```
sage: type(f.degree())
<type 'sage.rings.integer.Integer'>
```

list(*copy=True*)

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

EXAMPLES:

```
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*)

Returns the quotient and remainder of the Euclidean division of `self` and `other`.

Raises ZerodivisionError if `other` is zero. Raises ArithmeticError if `other` has a nonunit leading coefficient.

EXAMPLES:

```
sage: P.<x> = QQ[]
sage: R.<y> = P[]
sage: f = R.random_element(10)
sage: g = y^5+R.random_element(4)
sage: q,r = f.quo_rem(g)
sage: f == q*g + r
True
sage: g = x*y^5
sage: f.quo_rem(g)
Traceback (most recent call last):
...
ArithmeticError: Nonunit leading coefficient
sage: g = 0
sage: f.quo_rem(g)
Traceback (most recent call last):
...
ZeroDivisionError: Division by zero polynomial
```

shift (n)

Returns this polynomial multiplied by the power x^n. If n is negative, terms below x^n will be discarded. Does not change this polynomial.

EXAMPLES:

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

TESTS:

```
sage: p = R(0)
sage: p.shift(3).is_zero()
True
sage: p.shift(-3).is_zero()
True
```

AUTHORS:

- David Harvey (2006-08-06)

truncate (n)

Returns 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
q^10 + 1
sage: f.degree()
10
sage: f = (1+q^10+q^100).truncate(500); f
```

```
q^100 + q^10 + 1
```

TESTS:

Make sure we're not actually testing a specialized implementation.

```
sage: type(f)
<type 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
```

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

Return True if f is of type univariate polynomial.

INPUT:

- `f` - an object

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_element import is_Polynomial
sage: R.<x> = ZZ[]
sage: is_Polynomial(x^3 + x + 1)
True
sage: S.<y> = R[]
sage: f = y^3 + x*y -3*x; f
y^3 + x*y - 3*x
sage: is_Polynomial(f)
True
```

However this function does not return True for genuine multivariate polynomial type objects or symbolic polynomials, since those are not of the same data type as univariate polynomials:

```
sage: R.<x,y> = QQ[]
sage: f = y^3 + x*y -3*x; f
y^3 + x*y - 3*x
sage: is_Polynomial(f)
False
sage: var('x,y')
(x, y)
sage: f = y^3 + x*y -3*x; f
y^3 + x*y - 3*x
sage: is_Polynomial(f)
False
```

`sage.rings.polynomial.polynomial_element.`**`make_generic_polynomial`**`(`*parent*, *coeffs*`)`

`sage.rings.polynomial.polynomial_element.`**`universal_discriminant`**`(`*n*`)`

Return the discriminant of the 'universal' univariate polynomial $a_n x^n + \cdots + a_1 x + a_0$ in $\mathbf{Z}[a_0, \ldots, a_n][x]$.

INPUT:

- `n` - degree of the polynomial

OUTPUT:

The discriminant as a polynomial in $n + 1$ variables over $\mathbf{Z}$. The result will be cached, so subsequent computations of discriminants of the same degree will be faster.

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_element import universal_discriminant
sage: universal_discriminant(1)
1
```

```
sage: universal_discriminant(2)
a1^2 - 4*a0*a2
sage: universal_discriminant(3)
a1^2*a2^2 - 4*a0*a2^3 - 4*a1^3*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()`

2.3 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)

TESTS:

We test coercion in a particularly complicated situation:

```
sage: W.<w>=QQ['w']
sage: WZ.<z>=W['z']
sage: m = matrix(WZ,2,2,[1,z,z,z^2])
sage: a = m.charpoly()
sage: R.<x> = WZ[]
sage: R(a)
x^2 + (-z^2 - 1)*x
```

class `sage.rings.polynomial.polynomial_element_generic.`**`Polynomial_generic_dense_field`**(*parent, x=None, check=Tr is_gen=Fa con- struct=Fa*

Bases: `sage.rings.polynomial.polynomial_element.Polynomial_generic_dense`, `sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_field`

class `sage.rings.polynomial.polynomial_element_generic.`**`Polynomial_generic_domain`**(*parent, is_gen=False, con- struct=False*)

Bases: `sage.rings.polynomial.polynomial_element.Polynomial`, `sage.structure.element.IntegralDomainElement`

`is_unit`()

Return True if this polynomial is a unit.

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

EXAMPLES:

```
sage: R.<z> = PolynomialRing(ZZ, sparse=True)
sage: (2 + z^3).is_unit()
False
sage: f = -1 + 3*z^3; f
3*z^3 - 1
sage: f.is_unit()
False
sage: R(-3).is_unit()
False
sage: R(-1).is_unit()
True
sage: R(0).is_unit()
False
```

class `sage.rings.polynomial.polynomial_element_generic.`**`Polynomial_generic_field`**(*parent*, *is_gen=False*, *construct=False*)

Bases: `sage.rings.polynomial.polynomial_singular_interface.Polynomial_singular_repr`, `sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_domain`, `sage.structure.element.EuclideanDomainElement`

`gcd`(*other*)

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:

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

sage: zero = R.zero()
sage: zero.gcd(2*x)
x
sage: (2*x).gcd(zero)
x
sage: zero.gcd(zero)
0
```

`quo_rem`(*other*)

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

EXAMPLES:

```
sage: R.<y> = PolynomialRing(QQ)
sage: K.<t> = NumberField(y^2 - 2)
sage: P.<x> = PolynomialRing(K)
sage: x.quo_rem(K(1))
(x, 0)
sage: x.xgcd(K(1))
(1, 0, 1)
```

`xgcd`(*other*)

Extended gcd of `self` and polynomial `other`.

INPUT:

•`other` – a polynomial defined over the same ring as `self`

OUTPUT:

Polynomials `g`, `u`, and `v` such that `g = u * self + v * other`.

EXAMPLES:

```
sage: P.<x> = QQ[]
sage: F = (x^2 + 2)*x^3; G = (x^2+2)*(x-3)
sage: g, u, v = F.xgcd(G)
sage: g, u, v
(x^2 + 2, 1/27, -1/27*x^2 - 1/9*x - 1/3)
sage: u*F + v*G
x^2 + 2

sage: g, u, v = x.xgcd(P(0)); g, u, v
(x, 1, 0)
sage: g == u*x + v*P(0)
True
sage: g, u, v = P(0).xgcd(x); g, u, v
(x, 0, 1)
sage: g == u*P(0) + v*x
True
```

TESTS:

We check that the behavior of xgcd with zero elements is compatible with gcd (trac ticket #17671):

```
sage: R.<x> = QQbar[]
sage: zero = R.zero()
sage: zero.xgcd(2*x)
(x, 0, 1/2)
sage: (2*x).xgcd(zero)
(x, 1/2, 0)
sage: zero.xgcd(zero)
(0, 0, 0)
```

class `sage.rings.polynomial.polynomial_element_generic.`**`Polynomial_generic_sparse`**(*parent*, *x=None*, *check=True*, *is_gen=False*, *construct=False*)

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

A generic sparse polynomial.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(PolynomialRing(QQ, 'y'), sparse=True)
sage: f = x^3 - x + 17
sage: type(f)
<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse'>
sage: loads(f.dumps()) == f
True
```

A more extensive example:

```
sage: A.<T> = PolynomialRing(Integers(5),sparse=True) ; f = T^2+1 ; B = A.quo(f)
sage: C.<s> = PolynomialRing(B)
sage: C
Univariate Polynomial Ring in s over Univariate Quotient Polynomial Ring in Tbar over Ring of in
```

```
sage: s + T
s + Tbar
sage: (s + T)**2
s^2 + 2*Tbar*s + 4
```

coefficients(*sparse=True*)

Return the coefficients of the monomials appearing in self.

EXAMPLES:

```
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:

```
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:

```
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:

```
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]
```

list()

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

EXAMPLES:

```
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]
```

quo_rem(*other*)

Returns the quotient and remainder of the Euclidean division of `self` and `other`.

Raises ZerodivisionError if `other` is zero. Raises ArithmeticError if `other` has a nonunit leading coefficient.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(ZZ,sparse=True)
sage: R.<y> = PolynomialRing(P,sparse=True)
sage: f = R.random_element(10)
sage: g = y^5+R.random_element(4)
sage: q,r = f.quo_rem(g)
sage: f == q*g + r and r.degree() < g.degree()
True
sage: g = x*y^5
sage: f.quo_rem(g)
Traceback (most recent call last):
...
ArithmeticError: Nonunit leading coefficient
sage: g = 0
sage: f.quo_rem(g)
Traceback (most recent call last):
...
ZeroDivisionError: Division by zero polynomial
```

TESTS:

```
sage: P.<x> = PolynomialRing(ZZ,sparse=True)
sage: f = x^10-4*x^6-5
sage: g = 17*x^22+x^15-3*x^5+1
sage: q,r = g.quo_rem(f)
sage: g == f*q + r and r.degree() < f.degree()
True
sage: zero = P(0)
sage: zero.quo_rem(f)
(0, 0)
sage: Q.<y> = IntegerModRing(14)[]
sage: f = y^10-4*y^6-5
sage: g = 17*y^22+y^15-3*y^5+1
sage: q,r = g.quo_rem(f)
sage: g == f*q + r and r.degree() < f.degree()
True
sage: f += 2*y^10 # 3 is invertible mod 14
sage: q,r = g.quo_rem(f)
sage: g == f*q + r and r.degree() < f.degree()
True
```

AUTHORS:

- Bruno Grenet (2014-07-09)

shift(*n*)

Returns this polynomial multiplied by the power x^n. If n is negative, terms below x^n will be discarded. Does not change this polynomial.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: p = x^100000 + 2*x + 4
sage: type(p)
<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse'>
```

```
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)

valuation()

EXAMPLES:

```
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
```

class sage.rings.polynomial.polynomial_element_generic.**Polynomial_generic_sparse_field**(*parent, x=None, check=T is_gen= con- struct=F*

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse, sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_field

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Frac(RR['t']), sparse=True)
sage: f = x^3 - x + 17
sage: type(f)
<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_field'>
sage: loads(f.dumps()) == f
True
```

2.4 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.

EXAMPLE:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
sage: from sage.rings.polynomial.polynomial_gf2x import GF2X_BuildIrred_list, GF2X_BuildSparseIr
sage: all([GF2X_BuildSparseIrred_list(n) == GF2X_BuildIrred_list(n)
....:      for n in range(1,33)])
True
sage: GF(2)['x'](GF2X_BuildSparseIrred_list(33))
x^33 + x^10 + 1
```

class sage.rings.polynomial.polynomial_gf2x.**Polynomial_GF2X**

Bases: sage.rings.polynomial.polynomial_gf2x.Polynomial_template

Univariate Polynomials over GF(2) via NTL's GF2X.

EXAMPLE:

```
sage: P.<x> = GF(2)[]
sage: x^3 + x^2 + 1
x^3 + x^2 + 1
```

is_irreducible()

Return True precisely if this polynomial is irreducible over GF(2).

EXAMPLES:

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

modular_composition(*g*, *h*, *algorithm=None*)

Compute $f(g) \pmod h$.

Both implementations use Brent-Kung's Algorithm 2.1 (*Fast Algorithms for Manipulation of Formal Power Series*, JACM 1978).

INPUT:

- `g` – a polynomial
- `h` – a polynomial
- `algorithm` – either 'native' or 'ntl' (default: 'native')

EXAMPLE:

```
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
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 +
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
sage: f.modular_composition(g,h) == f(g) % h
True
```

AUTHORS:

- Paul Zimmermann (2008-10) initial implementation
- Martin Albrecht (2008-10) performance improvements

class `sage.rings.polynomial.polynomial_gf2x.`**`Polynomial_template`**

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

Template for interfacing to external C / C++ libraries for implementations of polynomials.

AUTHORS:

- Robert Bradshaw (2008-10): original idea for templating
- Martin Albrecht (2008-10): initial implementation

This file implements a simple templating engine for linking univariate polynomials to their C/C++ library implementations. It requires a 'linkage' file which implements the `celement_` functions (see `sage.libs.ntl.ntl_GF2X_linkage` for an example). Both parts are then plugged together by inclusion of the linkage file when inheriting from this class. See `sage.rings.polynomial.polynomial_gf2x` for an example.

We illustrate the generic glueing using univariate polynomials over $\mathrm{GF}(2)$.

Note: Implementations using this template MUST implement coercion from base ring elements and `__getitem__`. See `Polynomial_GF2X` for an example.

`degree`()

EXAMPLE:

```
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.

EXAMPLE:

```
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()

EXAMPLE:

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

is_one()

EXAMPLE:

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

is_zero()

EXAMPLE:

```
sage: P.<x> = GF(2)[]
sage: x.is_zero()
False
```

list()

EXAMPLE:

```
sage: P.<x> = GF(2)[]
sage: x.list()
[0, 1]
sage: list(x)
[0, 1]
```

quo_rem(*right*)

EXAMPLE:

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

shift(*n*)

EXAMPLE:

```
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
```

xgcd(*other*)

Computes extended gcd of self and other.

EXAMPLE:

```
sage: P.<x> = GF(7)[]
sage: f = x*(x+1)
sage: f.xgcd(x+1)
(x + 1, 0, 1)
sage: f.xgcd(x^2)
(x, 1, 6)
```

sage.rings.polynomial.polynomial_gf2x.**make_element**(*parent*, *args*)

2.5 Univariate polynomials over number fields.

AUTHOR:

- Luis Felipe Tabera Alonso (2014-02): initial version.

EXAMPLES:

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

```
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:

```
sage: x = var('x')
sage: Q7 = Qp(7)
sage: r1 = Q7(3 + 7 + 2*7^2 + 6*7^3 + 7^4 + 2*7^5 + 7^6 + 2*7^7 + 4*7^8 +\
          6*7^9 + 6*7^10 + 2*7^11 + 7^12 + 7^13 + 2*7^15 + 7^16 + 7^17 +\
          4*7^18 + 6*7^19)
sage: N.<b> = NumberField(x^2-2, embedding = r1)
sage: K.<t> = N[]
sage: f = t^3-2*t+1
```

```
sage: f(r1)
1 + O(7^20)
```

We can also construct polynomials over relative number fields:

```
sage: N.<i, s2> = QQ[I, sqrt(2)]
sage: K.<x> = N[]
sage: f = x - s2
sage: g = x^3 - 2*i*x^2 + s2*x
sage: f*(x + s2)
x^2 - 2
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`**(*pa*
x=
ch
*is_*
co
str

Bases: `sage.rings.polynomial.polynomial_element_generic.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: 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 + 6
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
```

TESTS:

Test for degree one extensions:

```
sage: x = var('x')
sage: N = NumberField(x-3, 'a')
sage: a = N.gen()
sage: R = N['x']
sage: f = R.random_element()
sage: g1 = R.random_element()
sage: g2 = g1*R.random_element() + 1
sage: g1 *= f
sage: g2 *= f
sage: d = gcd(g1, g2)
sage: f.monic() - d
0
sage: d.parent() is R
True
```

Test for coercion with other rings and force weird variables to test PARI behavior:

```
sage: r = var('r')
sage: N = NumberField(r^2 - 2, 'r')
sage: a = N.gen()
sage: R = N['r']
sage: r = R.gen()
sage: f = N.random_element(4)*r + 1
sage: g = ZZ['r']([1, 2, 3, 4, 5, 6, 7]); g
7*r^6 + 6*r^5 + 5*r^4 + 4*r^3 + 3*r^2 + 2*r + 1
sage: gcd(f, g) == gcd(g, f)
True
sage: h = f.gcd(g); h
1
sage: h.parent()
Univariate Polynomial Ring in r over Number Field in r with defining polynomial r^2 - 2
sage: gcd([a*r+2, r^2-2])
r + r
```

class sage.rings.polynomial.polynomial_number_field.**Polynomial_relative_number_field_dense**(*pa*
x=
ch
*is_*
co
str

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_field

Class of dense univariate polynomials over a relative number field.

gcd(*other*)

Compute the monic gcd of two polynomials.

Currently, the method checks corner cases in which one of the polynomials is zero or a constant. Then, computes an absolute extension and performs the computations there.

INPUT:

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

OUTPUT:

•The monic gcd of `self` and `other`.

See `Polynomial_absolute_number_field_dense.gcd()` for more details.

EXAMPLES:

```
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
```

TESTS:

```
sage: x = var('x')
sage: R = NumberField([x^2-2, x^2-3], 'a')['x']
sage: f = R.random_element()
sage: g1 = R.random_element()
sage: g2 = R.random_element()*g1+1
sage: g1 *= f
sage: g2 *= f
sage: f.monic() - g1.gcd(g2)
0
```

Test for degree one extensions:

```
sage: R = NumberField([x-2,x+1,x-3],'a')['x']
sage: f = R.random_element(2)
sage: g1 = R.random_element(2)
sage: g2 = R.random_element(2)*g1+1
sage: g1 *= f
sage: g2 *= f
sage: d = gcd(g1, g2)
sage: d - f.monic()
0
sage: d.parent() is R
True
```

Test for hardcoded variables:

```
sage: R = N['sqrt2sqrt3']
sage: x = R.gen()
sage: f = x^2 - 2
sage: g1 = x^2 - s3
sage: g2 = x - s2
sage: gcd(f, g1)
1
sage: gcd(f, g2)
sqrt2sqrt3 - sqrt2
```

2.6 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)

TESTS:

We check that the buggy gcd is fixed (see trac:17816):

```
sage: R.<q> = ZZ[]
sage: X = 3*q^12 - 8*q^11 - 24*q^10 - 48*q^9 - 84*q^8 - 92*q^7 - 92*q^6 - 70*q^5 - 50*q^4 - 27*q^3 -
sage: Y = q^13 - 2*q^12 + 2*q^10 - q^9
sage: gcd(X,Y)
1
```

class `sage.rings.polynomial.polynomial_integer_dense_flint.`**`Polynomial_integer_dense_flint`**

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

A dense polynomial over the integers, implemented via FLINT.

`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
```

TESTS:

```
sage: t = x^2+x+1
sage: t.content()
1
sage: (123456789123456789123456789123456789123456789*t).content()
123456789123456789123456789123456789123456789
```

Verify that trac ticket #13053 has been resolved:

```
sage: R(-1).content()
-1
```

`degree`(*gen=None*)

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

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: x.degree()
1
sage: (x^2).degree()
2
sage: R(1).degree()
0
```

```
sage: R(0).degree()
-1
```

disc (*proof=True*)

Return the discriminant of self, which is by definition

$$(-1)^{m(m-1)/2} \operatorname{resultant}(a, a') / \operatorname{lc}(a),$$

where $m = \deg(a)$, and $\operatorname{lc}(a)$ 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: R.<x> = ZZ[]
sage: f = 3*x^3 + 2*x + 1
sage: f.discriminant()
-339
sage: f.discriminant(proof=False)
-339
```

TESTS:

Confirm that trac ticket #17603 has been applied:

```
sage: f.disc()
-339
```

discriminant (*proof=True*)

Return the discriminant of self, which is by definition

$$(-1)^{m(m-1)/2} \operatorname{resultant}(a, a') / \operatorname{lc}(a),$$

where $m = \deg(a)$, and $\operatorname{lc}(a)$ 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: R.<x> = ZZ[]
sage: f = 3*x^3 + 2*x + 1
sage: f.discriminant()
-339
sage: f.discriminant(proof=False)
-339
```

TESTS:

Confirm that trac ticket #17603 has been applied:

```
sage: f.disc()
-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> = 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):
...
ValueError: factorization of 0 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
```

A more difficult example:

```
sage: f = 100 * (5*x + 1)^2 * (x + 5)^2
sage: f.factor_padic(5, 10)
(4 + O(5^10)) * ((5 + O(5^11)))^2 * ((1 + O(5^10))*x + (5 + O(5^10)))^2 * ((5 + O(5^10))*x +
```

gcd(*right*)

Return the GCD of self and right. The leading coefficient need not be 1.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = (6*x + 47)*(7*x^2 - 2*x + 38)
sage: g = (6*x + 47)*(3*x^3 + 2*x + 1)
sage: f.gcd(g)
6*x + 47
```

is_zero()

Returns True if self is equal to zero.

EXAMPLES:

```
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:

```
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()

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

EXAMPLES:

```
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 = self`. This function computes polynomials Q and R and an integer d such that

$$\mathrm{lead}(B)^d A = BQ + R$$

where R has degree less than that of B.

INPUT:

- B – a polynomial over **Z**

OUTPUT:

- Q, R – polynomials
- d – nonnegative integer

EXAMPLES:

```
sage: R.<x> = ZZ['x']
sage: A = R(range(10)); B = 3*R([-1, 0, 1])
sage: Q, R, d = A.pseudo_divrem(B)
sage: Q, R, d
(9*x^7 + 8*x^6 + 16*x^5 + 14*x^4 + 21*x^3 + 18*x^2 + 24*x + 20, 75*x + 60, 1)
sage: B.leading_coefficient()^d * A == B*Q + R
True
```

quo_rem(*right*)

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

EXAMPLES:

```
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)
```

TESTS:

```
sage: z = R(0)
sage: z.quo_rem(1)
(0, 0)
sage: z.quo_rem(x)
(0, 0)
sage: z.quo_rem(2*x)
(0, 0)
```

Ticket #383, make sure things get coerced correctly:

```
sage: f = x+1; parent(f)
Univariate Polynomial Ring in x over Integer Ring
sage: g = x/2; parent(g)
Univariate Polynomial Ring in x over Rational Field
sage: f.quo_rem(g)
```

```
(2, 1)
sage: g.quo_rem(f)
(1/2, -1/2)
sage: parent(f.quo_rem(g)[0])
Univariate Polynomial Ring in x over Rational Field
sage: f.quo_rem(3)
(0, x + 1)
sage: (5*x+7).quo_rem(3)
(x + 2, 2*x + 1)
```

real_root_intervals()

Returns isolating intervals for the real roots of this polynomial.

EXAMPLE: We compute the roots of the characteristic polynomial of some Salem numbers:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = 1 - x^2 - x^3 - x^4 + x^6
sage: f.real_root_intervals()
[((1/2, 3/4), 1), ((1, 3/2), 1)]
```

resultant(*other*, *proof=True*)

Returns the resultant of self and other, which must lie in the same polynomial ring.

If `proof = False` (the default is `proof=True`), then this function may use a randomized strategy that 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 and the reverse polynomial will have the specified degree.

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
```

TESTS:

```
sage: p.reverse(degree=1.5r)
Traceback (most recent call last):
...
ValueError: degree argument must be a non-negative integer, got 1.5
```

revert_series(*n*)

Return a polynomial f such that $f(self(x)) = self(f(x)) = xmodx^n$.

EXAMPLES:

```
sage: R.<t> = ZZ[]
sage: f = t - t^3 + t^5
sage: f.revert_series(6)
2*t^5 + t^3 + t

sage: f.revert_series(-1)
Traceback (most recent call last):
...
ValueError: argument n must be a non-negative integer, got -1

sage: g = - t^3 + t^5
sage: g.revert_series(6)
Traceback (most recent call last):
...
ValueError: self must have constant coefficient 0 and a unit for coefficient t^1
```

squarefree_decomposition()

Return the square-free decomposition of self. This is a partial factorization of self into square-free, relatively prime polynomials.

This is a wrapper for the NTL function SquareFreeDecomp.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: p = (x-1)^2 * (x-2)^2 * (x-3)^3 * (x-4)
sage: p.squarefree_decomposition()
(x - 4) * (x^2 - 3*x + 2)^2 * (x - 3)^3
sage: p = 37 * (x-1)^2 * (x-2)^2 * (x-3)^3 * (x-4)
sage: p.squarefree_decomposition()
(37) * (x - 4) * (x^2 - 3*x + 2)^2 * (x - 3)^3
```

TESTS:

Verify that trac ticket #13053 has been resolved:

```
sage: R.<x> = PolynomialRing(ZZ, implementation='FLINT')
sage: f=-x^2
sage: f.squarefree_decomposition()
(-1) * x^2
```

xgcd(*right*)

Return a triple `(g,s,t)` such that $g = s*self+t*right$ and such that g is the gcd of `self` and `right` up to a divisor of the resultant of `self` and `other`.

As integer polynomials do not form a principal ideal domain, it is not always possible given a and b to find a pair s,t such that $gcd(a,b) = sa + tb$. Take $a = x + 2$ and $b = x + 4$ as an example for which the gcd is 1 but the best you can achieve in the Bezout identity is 2.

If `self` and `right` are coprime as polynomials over the rationals, then `g` is guaranteed to be the resultant of `self` and `right`, as a constant polynomial.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(ZZ)

sage: (x+2).xgcd(x+4)
(2, -1, 1)
sage: (x+2).resultant(x+4)
2
sage: (x+2).gcd(x+4)
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, 1, -x^2 - 3*x - 9)
sage: u*F + v*G
27*x^2 + 54

sage: zero = P(0)
sage: x.xgcd(zero)
(x, 1, 0)
sage: zero.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
```

TESTS:

Check that trac ticket #17675 is fixed:

```
sage: R.<x> = ZZ['x']
sage: R(2).xgcd(R(2))
(2, 0, 1)
sage: R.zero().xgcd(R(2))
(2, 0, 1)
sage: R(2).xgcd(R.zero())
(2, 1, 0)
```

2.7 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 **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:

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

class sage.rings.polynomial.polynomial_integer_dense_ntl.**Polynomial_integer_dense_ntl**

Bases: sage.rings.polynomial.polynomial_element.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:

```
sage: R.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: (2*x^2 - 4*x^4 + 14*x^7).content()
2
sage: (2*x^2 - 4*x^4 - 14*x^7).content()
-2
sage: x.content()
1
sage: R(1).content()
1
sage: R(0).content()
0
```

degree(*gen=None*)

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

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: x.degree()
1
sage: (x^2).degree()
2
sage: R(1).degree()
0
sage: R(0).degree()
-1
```

discriminant(*proof=True*)

Return the discriminant of self, which is by definition

$$(-1)^{m(m-1)/2}\texttt{resultant}(a, a')/lc(a),$$

where $m = deg(a)$, 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):
...
ValueError: factorization of 0 not defined

sage: f = 2*x*(x-2)*(x-9)
sage: f.factor_mod(7)
(2) * x * (x + 5)^2
```

factor_padic(*p*, *prec=10*)

Return p-adic factorization of self to given precision.

INPUT:

- `p` – prime
- `prec` – integer; the precision

OUTPUT:

- factorization of `self` over the completion at p.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: f = x^2 + 1
sage: f.factor_padic(5, 4)
((1 + O(5^4))*x + (2 + 5 + 2*5^2 + 5^3 + O(5^4))) * ((1 + O(5^4))*x + (3 + 3*5 + 2*5^2 + 3*5
```

A more difficult example:

```
sage: f = 100 * (5*x + 1)^2 * (x + 5)^2
sage: f.factor_padic(5, 10)
(4 + O(5^10)) * ((5 + O(5^11)))^2 * ((1 + O(5^10))*x + (5 + O(5^10)))^2 * ((5 + O(5^10))*x +
```

gcd(*right*)

Return the GCD of self and right. The leading coefficient need not be 1.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: f = (6*x + 47)*(7*x^2 - 2*x + 38)
sage: g = (6*x + 47)*(3*x^3 + 2*x + 1)
sage: f.gcd(g)
6*x + 47
```

lcm(*right*)

Return the LCM of self and right.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: f = (6*x + 47)*(7*x^2 - 2*x + 38)
sage: g = (6*x + 47)*(3*x^3 + 2*x + 1)
sage: h = f.lcm(g); h
126*x^6 + 951*x^5 + 486*x^4 + 6034*x^3 + 585*x^2 + 3706*x + 1786
sage: h == (6*x + 47)*(7*x^2 - 2*x + 38)*(3*x^3 + 2*x + 1)
True
```

list()

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

EXAMPLES:

```
sage: x = PolynomialRing(ZZ,'x',implementation='NTL').0
sage: f = x^3 + 3*x - 17
sage: f.list()
[-17, 3, 0, 1]
sage: f = PolynomialRing(ZZ,'x',implementation='NTL')(0)
sage: f.list()
[]
```

quo_rem(*right*)

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

If right is monic, then it returns `(q, r)` where $self = q * right + r$ 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: 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)
```

TESTS:

```
sage: z = R(0)
sage: z.quo_rem(1)
(0, 0)
sage: z.quo_rem(x)
(0, 0)
sage: z.quo_rem(2*x)
(0, 0)
```

real_root_intervals()

Returns isolating intervals for the real roots of this polynomial.

EXAMPLE: 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
```

TESTS:

Verify that trac ticket #13053 has been resolved:

```
sage: R.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: f=-x^2
sage: f.squarefree_decomposition()
(-1) * x^2
```

xgcd(*right*)

This function can't in general return `(g,s,t)` as above, since they need not exist. Instead, over the integers, we first multiply g by a divisor of the resultant of a/g and b/g, up to sign, and return `g, u, v` such that `g = s*self + s*right`. But note that this g may be a multiple of the gcd.

If `self` and `right` are coprime as polynomials over the rationals, then `g` is guaranteed to be the resultant of self and right, as a constant polynomial.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: F = (x^2 + 2)*x^3; G = (x^2+2)*(x-3)
sage: g, u, v = F.xgcd(G)
sage: g, u, v
(27*x^2 + 54, 1, -x^2 - 3*x - 9)
sage: u*F + v*G
27*x^2 + 54
sage: x.xgcd(P(0))
(x, 1, 0)
sage: f = P(0)
sage: f.xgcd(x)
(x, 0, 1)
sage: F = (x-3)^3; G = (x-15)^2
sage: g, u, v = F.xgcd(G)
sage: g, u, v
(2985984, -432*x + 8208, 432*x^2 + 864*x + 14256)
sage: u*F + v*G
2985984
```

2.8 Dense univariate polynomials over $\mathbf{Z}/n\mathbf{Z}$, implemented using FLINT.

This module gives a fast implementation of $(\mathbf{Z}/n\mathbf{Z})[x]$ whenever n is at most `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)
<type 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>
sage: R.<a> = PolynomialRing(Integers(5*2^64))
sage: type(a)
<type 'sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_ZZ'>
sage: R.<a> = PolynomialRing(Integers(5*2^64), implementation="FLINT")
Traceback (most recent call last):
...
ValueError: FLINT does not support modulus 92233720368547758080
```

AUTHORS:

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

class `sage.rings.polynomial.polynomial_zmod_flint.`**`Polynomial_template`**

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

Template for interfacing to external C / C++ libraries for implementations of polynomials.

AUTHORS:

- Robert Bradshaw (2008-10): original idea for templating
- Martin Albrecht (2008-10): initial implementation

This file implements a simple templating engine for linking univariate polynomials to their C/C++ library implementations. It requires a 'linkage' file which implements the `celement_` functions (see `sage.libs.ntl.ntl_GF2X_linkage` for an example). Both parts are then plugged together by inclusion of the linkage file when inheriting from this class. See `sage.rings.polynomial.polynomial_gf2x` for an example.

We illustrate the generic glueing using univariate polynomials over $\mathrm{GF}(2)$.

Note: Implementations using this template MUST implement coercion from base ring elements and `__getitem__`. See `Polynomial_GF2X` for an example.

`degree`()

EXAMPLE:

```
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.

EXAMPLE:

```
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()

EXAMPLE:

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

is_one()

EXAMPLE:

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

is_zero()

EXAMPLE:

```
sage: P.<x> = GF(2)[]
sage: x.is_zero()
False
```

list()

EXAMPLE:

```
sage: P.<x> = GF(2)[]
sage: x.list()
[0, 1]
sage: list(x)
[0, 1]
```

quo_rem(*right*)

EXAMPLE:

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

shift(*n*)

EXAMPLE:

```
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
```

xgcd(*other*)

Computes extended gcd of self and other.

EXAMPLE:

```
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: sage.rings.polynomial.polynomial_zmod_flint.Polynomial_template

EXAMPLE:

```
sage: P.<x> = GF(32003)[]
sage: f = 24998*x^2 + 29761*x + 2252
```

factor()

Returns the factorization of the polynomial.

EXAMPLES:

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

TESTS:

```
sage: (2*x^2 + 2).factor()
(2) * (x + 2) * (x + 3)
sage: P.<x> = Zmod(10)[]
sage: (x^2).factor()
Traceback (most recent call last):
...
NotImplementedError: factorization of polynomials over rings with composite characteristic i
```

is_irreducible()

Return True if this polynomial is irreducible.

EXAMPLES:

```
sage: R.<x> = GF(5)[]
sage: (x^2 + 1).is_irreducible()
False
sage: (x^3 + x + 1).is_irreducible()
True
```

TESTS:

```
sage: R(0).is_irreducible()
False
sage: R(1).is_irreducible()
False
sage: R(2).is_irreducible()
False

sage: S.<s> = Zmod(10)[]
sage: (s^2).is_irreducible()
Traceback (most recent call last):
```

```
...
NotImplementedError: checking irreducibility of polynomials over rings with composite charac
sage: S(1).is_irreducible()
False
sage: S(2).is_irreducible()
Traceback (most recent call last):
...
NotImplementedError: checking irreducibility of polynomials over rings with composite charac
```

monic()

Return this polynomial divided by its leading coefficient.

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

EXAMPLES:

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

TESTS:

```
sage: R.<x> = Zmod(10)[]
sage: (5*x).monic()
Traceback (most recent call last):
...
ValueError: leading coefficient must be invertible
```

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

Construct a rational function n/d such that $p * d$ is equivalent to n modulo m where p is this polynomial.

EXAMPLES:

```
sage: P.<x> = GF(5)[]
sage: p = 4*x^5 + 3*x^4 + 2*x^3 + 2*x^2 + 4*x + 2
sage: n, d = p.rational_reconstruct(x^9, 4, 4); n, d
(3*x^4 + 2*x^3 + x^2 + 2*x, x^4 + 3*x^3 + x^2 + x)
sage: (p*d % x^9) == n
True
```

resultant(*other*)

Returns the resultant of self and other, which must lie in the same polynomial ring.

INPUT:

- other – a polynomial

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

EXAMPLES:

```
sage: R.<x> = GF(19)['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 #11782 has been fixed:

```
sage: R.<x> = ZZ.quo(9)['x']
sage: f = 2*x^3 + x^2 + x;  g = 6*x^2 + 2*x + 1
```

```
sage: f.resultant(g)
5
```

reverse(*degree=None*)

Return a polynomial with the coefficients of this polynomial reversed.

If an optional degree argument is given the coefficient list will be truncated or zero padded as necessary and the reverse polynomial will have the specified degree.

EXAMPLES:

```
sage: R.<x> = GF(5)[]
sage: p = R([1,2,3,4]); p
4*x^3 + 3*x^2 + 2*x + 1
sage: p.reverse()
x^3 + 2*x^2 + 3*x + 4
sage: p.reverse(degree=6)
x^6 + 2*x^5 + 3*x^4 + 4*x^3
sage: p.reverse(degree=2)
x^2 + 2*x + 3

sage: R.<x> = GF(101)[]
sage: f = x^3 - x + 2; f
x^3 + 100*x + 2
sage: f.reverse()
2*x^3 + 100*x^2 + 1
sage: f.reverse() == f(1/x) * x^f.degree()
True
```

Note that if f has zero constant coefficient, its reverse will have lower degree.

```
sage: f = x^3 + 2*x
sage: f.reverse()
2*x^2 + 1
```

In this case, reverse is not an involution unless we explicitly specify a degree.

```
sage: f
x^3 + 2*x
sage: f.reverse().reverse()
x^2 + 2
sage: f.reverse(5).reverse(5)
x^3 + 2*x
```

TESTS:

```
sage: p.reverse(degree=1.5r)
Traceback (most recent call last):
...
ValueError: degree argument must be a non-negative integer, got 1.5
```

revert_series(*n*)

Return a polynomial f such that $f(self(x)) = self(f(x)) = xmodx^n$.

EXAMPLES:

```
sage: R.<t> =  GF(5)[]
sage: f = t + 2*t^2 - t^3 - 3*t^4
sage: f.revert_series(5)
3*t^4 + 4*t^3 + 3*t^2 + t
```

```
sage: f.revert_series(-1)
Traceback (most recent call last):
...
ValueError: argument n must be a non-negative integer, got -1

sage: g = - t^3 + t^5
sage: g.revert_series(6)
Traceback (most recent call last):
...
ValueError: self must have constant coefficient 0 and a unit for coefficient t^1

sage: g = t + 2*t^2 - t^3 -3*t^4 + t^5
sage: g.revert_series(6)
Traceback (most recent call last):
...
ValueError: the integers 1 up to n=5 are required to be invertible over the base field
```

small_roots (**args*, ***kwds*)

See `sage.rings.polynomial.polynomial_modn_dense_ntl.small_roots()` for the documentation of this function.

EXAMPLE:

```
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 ()

Returns the squarefree decomposition of this polynomial.

EXAMPLES:

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

TESTS:

```
sage: (2*x*(x+1)^2).squarefree_decomposition()
(2) * x * (x + 1)^2
sage: P.<x> = Zmod(10)[]
sage: (x^2).squarefree_decomposition()
Traceback (most recent call last):
...
NotImplementedError: square free factorization of polynomials over rings with composite char
```

`sage.rings.polynomial.polynomial_zmod_flint.`**make_element** (*parent*, *args*)

2.9 Dense univariate polynomials over $\mathbf{Z}/n\mathbf{Z}$, implemented using NTL.

This implementation is generally slower than the FLINT implementation in `polynomial_zmod_flint`, so we use FLINT by default when the modulus is small enough; but NTL does not require that n be `int`-sized, so we use it as default when n is too large for FLINT.

Note that the classes `Polynomial_dense_modn_ntl_zz` and `Polynomial_dense_modn_ntl_ZZ` are different; the former is limited to moduli less than a certain bound, while the latter supports arbitrarily large moduli.

AUTHORS:

- Robert Bradshaw: Split off from polynomial_element_generic.py (2007-09)
- Robert Bradshaw: Major rewrite to use NTL directly (2007-09)

class `sage.rings.polynomial.polynomial_modn_dense_ntl.`**`Polynomial_dense_mod_n`**

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

A dense polynomial over the integers modulo n, where n is composite, with the underlying arithmetic done using NTL.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(16), implementation='NTL')
sage: f = x^3 - x + 17
sage: f^2
x^6 + 14*x^4 + 2*x^3 + x^2 + 14*x + 1

sage: loads(f.dumps()) == f
True

sage: R.<x> = PolynomialRing(Integers(100), implementation='NTL')
sage: p = 3*x
sage: q = 7*x
sage: p+q
10*x
sage: R.<x> = PolynomialRing(Integers(8), implementation='NTL')
sage: parent(p)
Univariate Polynomial Ring in x over Ring of integers modulo 100 (using NTL)
sage: p + q
10*x
sage: R({10:-1})
7*x^10
```

`degree`(*gen=None*)

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

`int_list`()

`list`()

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]
```

`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
sage: poly_modn_dense(R, ([1,-2,3]))
3*x^2 + 98*x + 1
sage: f = poly_modn_dense(R, 0)
sage: f.ntl_set_directly([1,-2,3])
sage: f
3*x^2 + 98*x + 1
sage: f.ntl_set_directly('[1 -2 3 4]')
sage: f
4*x^3 + 3*x^2 + 98*x + 1
```

quo_rem(*right*)

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

shift(*n*)

Returns this polynomial multiplied by the power x^n. If n is negative, terms below x^n will be discarded. Does not change this polynomial.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(12345678901234567890), implementation='NTL')
sage: p = x^2 + 2*x + 4
sage: p.shift(0)
 x^2 + 2*x + 4
sage: p.shift(-1)
 x + 2
sage: p.shift(-5)
 0
sage: p.shift(2)
 x^4 + 2*x^3 + 4*x^2
```

TESTS:

```
sage: p = R(0)
sage: p.shift(3).is_zero()
True
sage: p.shift(-3).is_zero()
True
```

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.

EXAMPLE:

```
sage: N = 10001
sage: K = Zmod(10001)
sage: P.<x> = PolynomialRing(K, implementation='NTL')
sage: f = x^3 + 10*x^2 + 5000*x - 222
sage: f.small_roots()
[4]
```

class `sage.rings.polynomial.polynomial_modn_dense_ntl.`**`Polynomial_dense_mod_p`**

Bases: `sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n`

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

`discriminant`()

EXAMPLES:

```
sage: _.<x> = PolynomialRing(GF(19),implementation='NTL')
sage: f = x^3 + 3*x - 17
sage: f.discriminant()
12
```

`gcd`(*right*)

Return the greatest common divisor of this polynomial and `other`, as a monic polynomial.

INPUT:

- `other` – a polynomial defined over the same ring as `self`

EXAMPLES:

```
sage: R.<x> = PolynomialRing(GF(3),implementation="NTL")
sage: f,g = x + 2, x^2 - 1
sage: f.gcd(g)
x + 2
```

`resultant`(*other*)

Returns the resultant of self and other, which must lie in the same polynomial ring.

INPUT:

- `other` – a polynomial

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

EXAMPLES:

```
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*self + t*other`.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(GF(3),implementation='NTL')
sage: x.xgcd(x)
(x, 0, 1)
sage: (x^2 - 1).xgcd(x - 1)
(x + 2, 0, 1)
sage: R.zero().xgcd(R.one())
(1, 0, 1)
sage: (x^3 - 1).xgcd((x - 1)^2)
(x^2 + x + 1, 0, 1)
sage: ((x - 1)*(x + 1)).xgcd(x*(x - 1))
(x + 2, 1, 2)
```

class sage.rings.polynomial.polynomial_modn_dense_ntl.**Polynomial_dense_modn_ntl_ZZ**

Bases: sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n

degree()

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(14^34), implementation='NTL')
sage: f = x^4 - x - 1
sage: f.degree()
4
sage: f = 14^43*x + 1
sage: f.degree()
0
```

is_gen()

list()

quo_rem(*right*)

Returns q and r, with the degree of r less than the degree of $right$, such that $q * right + r = self$.

EXAMPLES:

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

Reverses the coefficients of self. The reverse of $f(x)$ is $x^n f(1/x)$.

The degree will go down if the constant term is zero.

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()
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(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
```

TESTS:

```
sage: p = R(0)
sage: p.shift(3).is_zero()
True
sage: p.shift(-3).is_zero()
True
```

truncate (n)

Returns 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 ()

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

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(10^50), implementation='NTL')
sage: x.valuation()
1
sage: f = x-3; f.valuation()
0
sage: f = x^99; f.valuation()
99
sage: f = x-x; f.valuation()
+Infinity
```

class sage.rings.polynomial.polynomial_modn_dense_ntl.**Polynomial_dense_modn_ntl_zz**

Bases: sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n

EXAMPLES:

```
sage: R = Integers(5**21)
sage: S.<x> = PolynomialRing(R, implementation='NTL')
sage: S(1/4)
357627868652344
```

degree ()

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(77), implementation='NTL')
sage: f = x^4 - x - 1
sage: f.degree()
4
sage: f = 77*x + 1
sage: f.degree()
0
```

int_list()

Returns the coefficients of self as efficiently as possible as a list of python ints.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(100), implementation='NTL')
sage: from sage.rings.polynomial.polynomial_modn_dense_ntl import Polynomial_dense_mod_n as
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()]
[<type 'int'>, <type 'int'>, <type 'int'>, <type 'int'>]
```

is_gen()

ntl_set_directly(*v*)

quo_rem(*right*)

Returns q and r, with the degree of r less than the degree of $right$, such that $q * right + r = self$.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(125), implementation='NTL')
sage: f = x^5+1; g = (x+1)^2
sage: q, r = f.quo_rem(g)
sage: q
x^3 + 123*x^2 + 3*x + 121
sage: r
5*x + 5
sage: q*g + r
x^5 + 1
```

reverse()

Reverses the coefficients of self. The reverse of $f(x)$ is $x^n f(1/x)$.

The degree will go down if the constant term is zero.

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 = x^3 - x
sage: f.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
```

TESTS:

```
sage: p = R(0)
sage: p.shift(3).is_zero()
True
sage: p.shift(-3).is_zero()
True
```

truncate(*n*)

Returns this polynomial mod x^n.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(77), implementation='NTL')
sage: f = sum(x^n for n in range(10)); f
x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
sage: f.truncate(6)
x^5 + x^4 + x^3 + x^2 + x + 1
```

valuation()

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

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(10), implementation='NTL')
sage: x.valuation()
1
sage: f = x-3; f.valuation()
0
sage: f = x^99; f.valuation()
99
sage: f = x-x; f.valuation()
+Infinity
```

`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 $X = ceil(1/2N^{\beta^2/\delta-\epsilon})$. The algorithm may also return some roots which are larger than X. 'This algorithm' in this context means Coppersmith's algorithm for finding small roots using the LLL algorithm. The implementation of this algorithm follows Alexander May's PhD thesis referenced below.

INPUT:

- `X` – an absolute bound for the root (default: see above)

- `beta` – compute a root mod b where b is a factor of N and $b \geq N^\beta$. (Default: 1.0, so $b = N$.)
- `epsilon` – the parameter ϵ described above. (Default: $\beta/8$)
- `**kwds` – passed through to method `Matrix_integer_dense.LLL()`.

EXAMPLES:

First consider a small example:

```
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 **Z**):

```
sage: f.change_ring(ZZ).roots()
[]
```

To compute its roots we need to factor the modulus N and use the Chinese remainder theorem:

```
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:

```
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:

```
sage: Nbits, Kbits = 512, 56
sage: e = 3
```

We choose two primes of size 256-bit each:

```
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:

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

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

```
sage: Kdigits = K.digits(2)
sage: M = [0]*Kbits + [1]*(Nbits-Kbits)
sage: for i in range(len(Kdigits)): M[i] = Kdigits[i]

sage: M = ZZ(M, 2)
```

Now we encrypt the resulting message:

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

To recover K we consider the following polynomial modulo N:

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

and recover its small roots:

```
sage: Kbar = f.small_roots()[0]
sage: K == Kbar
True
```

The same algorithm can be used to factor $N = pq$ if partial knowledge about q is available. This example is from the Magma handbook:

First, we set up p, q and N:

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

Now we disturb the low 110 bits of q:

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

And try to recover q from it:

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

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

```
sage: set_verbose(2)
sage: d = f.small_roots(X=2^hidden-1, beta=0.5)[0] # time random
verbose 2 (<module>) m = 4
verbose 2 (<module>) t = 4
verbose 2 (<module>) X = 1298074214633706907132624082305023
verbose 1 (<module>) LLL of 8x8 matrix (algorithm fpLLL:wrapper)
verbose 1 (<module>) LLL finished (time = 0.006998)
sage: q == qbar - d
True
```

REFERENCES:

Don Coppersmith. *Finding a small root of a univariate modular equation.* In Advances in Cryptology, EuroCrypt 1996, volume 1070 of Lecture Notes in Computer Science, p. 155–165. Springer, 1996. http://cr.yp.to/bib/2001/coppersmith.pdf

Alexander May. *New RSA Vulnerabilities Using Lattice Reduction Methods.* PhD thesis, University of Paderborn, 2003. http://www.cs.uni-paderborn.de/uploads/tx_sibibtex/bp.pdf

2.10 Dense univariate polynomials over $\mathbb{R}$, implemented using MPFR

class `sage.rings.polynomial.polynomial_real_mpfr_dense.`**`PolynomialRealDense`**

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

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: PolynomialRealDense(RR['x'], [1, int(2), RR(3), 4/1, pi])
3.14159265358979*x^4 + 4.00000000000000*x^3 + 3.00000000000000*x^2 + 2.00000000000000*x + 1.0000
sage: PolynomialRealDense(RR['x'], None)
0
```

TESTS:

Check that errors and interrupts are handled properly (see #10100):

```
sage: a = var('a')
sage: PolynomialRealDense(RR['x'], [1,a])
Traceback (most recent call last):
...
TypeError: Cannot evaluate symbolic expression to a numeric value.
sage: R.<x> = SR[]
sage: (x-a).change_ring(RR)
Traceback (most recent call last):
...
TypeError: Cannot evaluate symbolic expression to a numeric value.
sage: sig_on_count()
0
sage: alarm(0.5); PolynomialRealDense(RR['x'], ZZ)   # Will loop forever
Traceback (most recent call last):
...
AlarmInterrupt
```

Test that we don't clean up uninitialized coefficients (#9826):

```
sage: k.<a> = GF(7^3)
sage: P.<x> = PolynomialRing(k)
sage: (a*x).complex_roots()
Traceback (most recent call last):
...
TypeError: Unable to convert x (='a') to real number.
```

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()

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
```

gcd(*other*)

Returns the gcd of self and other as a monic polynomial. Due to the inherit instability of division in this inexact ring, the results may not be entirely stable.

EXAMPLES:

```
sage: R.<x> = RR[]
sage: (x^3).gcd(x^5+1)
1.00000000000000
sage: (x^3).gcd(x^5+x^2)
x^2
sage: f = (x+3)^2 * (x-1)
sage: g = (x+3)^5
sage: f.gcd(g)
x^2 + 6.00000000000000*x + 9.00000000000000
```

Unless the division is exact (i.e. no rounding occurs) the returned gcd is almost certain to be 1.

```
sage: f = (x+RR.pi())^2 * (x-1)
sage: g = (x+RR.pi())^5
sage: f.gcd(g)
1.00000000000000
```

integral()

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RR['x'], [3, pi, 1])
sage: f.integral()
0.333333333333333*x^3 + 1.57079632679490*x^2 + 3.00000000000000*x
```

list()

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*)

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: 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()

Returns $x^d f(1/x)$ where d is the degree of f.

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RR['x'], [-3, pi, 0, 1])
sage: f.reverse()
-3.00000000000000*x^3 + 3.14159265358979*x^2 + 1.00000000000000
```

shift(*n*)

Returns this polynomial multiplied by the power x^n. If n is negative, terms below x^n will be discarded. Does not change this polynomial.

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RR['x'], [1, 2, 3]); f
3.00000000000000*x^2 + 2.00000000000000*x + 1.00000000000000
sage: f.shift(10)
3.00000000000000*x^12 + 2.00000000000000*x^11 + x^10
sage: f.shift(-1)
3.00000000000000*x + 2.00000000000000
sage: f.shift(-10)
0
```

TESTS:

```
sage: f = RR['x'](0)
sage: f.shift(3).is_zero()
True
sage: f.shift(-3).is_zero()
True
```

truncate(*n*)

Returns the polynomial of degree $< n$ which is equivalent to self modulo x^n.

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RealField(10)['x'], [1, 2, 4, 8])
sage: f.truncate(3)
4.0*x^2 + 2.0*x + 1.0
sage: f.truncate(100)
8.0*x^3 + 4.0*x^2 + 2.0*x + 1.0
sage: f.truncate(1)
1.0
sage: f.truncate(0)
0
```

truncate_abs(*bound*)

Truncate all high order coefficients below bound.

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RealField(10)['x'], [10^-k for k in range(10)])
sage: f
1.0e-9*x^9 + 1.0e-8*x^8 + 1.0e-7*x^7 + 1.0e-6*x^6 + 0.000010*x^5 + 0.00010*x^4 + 0.0010*x^3
```

```
sage: f.truncate_abs(0.5e-6)
1.0e-6*x^6 + 0.000010*x^5 + 0.00010*x^4 + 0.0010*x^3 + 0.010*x^2 + 0.10*x + 1.0
sage: f.truncate_abs(10.0)
0
sage: f.truncate_abs(1e-100) == f
True
```

`sage.rings.polynomial.polynomial_real_mpfr_dense.`**`make_PolynomialRealDense`**(*parent*, *data*)

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import make_PolynomialRealDense
sage: make_PolynomialRealDense(RR['x'], [1,2,3])
3.00000000000000*x^2 + 2.00000000000000*x + 1.00000000000000
```

2.11 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)

TESTS:

```
sage: R = PolynomialRing(GF(2**8,'a'),10,'x', order='invlex')
sage: R == loads(dumps(R))
True
sage: P.<a,b> = PolynomialRing(GF(7), 2)
sage: f = (a^3 + 2*b^2*a)^7; f
a^21 + 2*a^7*b^14
```

class `sage.rings.polynomial.polynomial_singular_interface.`**`PolynomialRing_singular_repr`**

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`**

Implements coercion of polynomials to Singular polynomials.

This class is a base class for all (univariate and multivariate) polynomial classes which support conversion from and to Singular polynomials.

Due to the incompatibility of Python extension classes and multiple inheritance, this just defers to module-level functions.

`sage.rings.polynomial.polynomial_singular_interface.`**`can_convert_to_singular`**(*R*)

Returns True if this ring's base field or ring can be represented in Singular, and the polynomial ring has at least one generator. If this is True then this polynomial ring can be represented in Singular.

The following base rings are supported: finite fields, rationals, number fields, and real and complex fields.

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_singular_interface import can_convert_to_singular
sage: can_convert_to_singular(PolynomialRing(QQ, names=['x']))
True

sage: can_convert_to_singular(PolynomialRing(QQ, names=[]))
False
```

2.12 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 http://www.sagemath.org:9001/days4schedule .

exception `sage.rings.polynomial.real_roots.`**`PrecisionError`**

Bases: `exceptions.ValueError`

x.__init__(...) initializes x; see help(type(x)) for signature

`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, -164, -474, -816, -11
```

class `sage.rings.polynomial.real_roots.`**`bernstein_polynomial_factory`**

An abstract base class for bernstein_polynomial factories. That is, elements of subclasses represent Bernstein polynomials (exactly), and are responsible for creating interval_bernstein_polynomial_integer approximations at arbitrary precision.

Supports four methods, coeffs_bitsize(), bernstein_polynomial(), lsign(), and usign(). The coeffs_bitsize() method gives an integer approximation to the log2 of the max of the absolute values of the Bernstein coefficients. The bernstein_polynomial(scale_log2) method gives an approximation where the maximum coefficient has approximately coeffs_bitsize() - scale_log2 bits. The lsign() and usign() methods give the (exact) sign of the first and last coefficient, respectively.

`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: `sage.rings.polynomial.real_roots.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:

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

```
sage: p = x^2 - 1
sage: bpf = bernstein_polynomial_factory_ar(p, False)
sage: bpf.bernstein_polynomial(-10)
<IBP: ((-1024, 0, 1024) + [0 .. 1)) * 2^-10>
```

coeffs_bitsize()

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

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(AA)
sage: p = (x - 1) * (x - sqrt(AA(2))) * (x - 2)
sage: bernstein_polynomial_factory_ar(p, False).coeffs_bitsize()
1
```

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

Bases: sage.rings.polynomial.real_roots.bernstein_polynomial_factory

This class holds an exact Bernstein polynomial (represented as a list of integer coefficients), and returns arbitrarily-precise interval approximations of this polynomial on demand.

bernstein_polynomial(*scale_log2*)

Compute an interval_bernstein_polynomial_integer that approximates this polynomial, using the given scale_log2. (Smaller scale_log2 values give more accurate approximations.)

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: bpf = bernstein_polynomial_factory_intlist([10, -20, 30, -40])
sage: print bpf.bernstein_polynomial(0)
degree 3 IBP with 6-bit coefficients
sage: bpf.bernstein_polynomial(20)
<IBP: ((0, -1, 0, -1) + [0 .. 1)) * 2^20; lsign 1>
sage: bpf.bernstein_polynomial(0)
<IBP: (10, -20, 30, -40) + [0 .. 1)>
sage: bpf.bernstein_polynomial(-20)
<IBP: ((10485760, -20971520, 31457280, -41943040) + [0 .. 1)) * 2^-20>
```

coeffs_bitsize()

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

EXAMPLES:

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

class sage.rings.polynomial.real_roots.**bernstein_polynomial_factory_ratlist**(*coeffs*)

Bases: sage.rings.polynomial.real_roots.bernstein_polynomial_factory

This class holds an exact Bernstein polynomial (represented as a list of rational coefficients), and returns arbitrarily-precise interval approximations of this polynomial on demand.

bernstein_polynomial(*scale_log2*)

Compute an interval_bernstein_polynomial_integer that approximates this polynomial, using the given scale_log2. (Smaller scale_log2 values give more accurate approximations.)

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: bpf = bernstein_polynomial_factory_ratlist([1/3, -22/7, 193/71, -140/99])
sage: print bpf.bernstein_polynomial(0)
degree 3 IBP with 3-bit coefficients
sage: bpf.bernstein_polynomial(20)
<IBP: ((0, -1, 0, -1) + [0 .. 1)) * 2^20; lsign 1>
sage: bpf.bernstein_polynomial(0)
<IBP: (0, -4, 2, -2) + [0 .. 1); lsign 1>
sage: bpf.bernstein_polynomial(-20)
<IBP: ((349525, -3295525, 2850354, -1482835) + [0 .. 1)) * 2^-20>
```

coeffs_bitsize()

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

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: bernstein_polynomial_factory_ratlist([1, 2, 3, -60000]).coeffs_bitsize()
15
sage: bernstein_polynomial_factory_ratlist([65535/65536]).coeffs_bitsize()
-1
sage: bernstein_polynomial_factory_ratlist([65536/65535]).coeffs_bitsize()
1
```

sage.rings.polynomial.real_roots.**bernstein_up**(*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:

```
sage: from sage.rings.polynomial.real_roots import *
sage: bernstein_down(3, 7, 4)
[  12/5     -4      3   -2/5]
[-13/15   16/3     -4   8/15]
[  8/15     -4   16/3 -13/15]
[  -2/5      3     -4   12/5]
```

sage.rings.polynomial.real_roots.**bitsize_doctest**(*n*)

sage.rings.polynomial.real_roots.**cl_maximum_root**(*cl*)

Given a polynomial represented by a list of its coefficients (as RealIntervalFieldElements), compute an upper bound on its largest real root.

Uses two algorithms of Akritas, Strzebo'nski, 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.

Uses the first-lambda algorithm from "Implementations of a New Theorem for Computing Bounds for Positive Roots of Polynomials", by Akritas, Strzebo'nski, and Vigklas.

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.

Uses the local-max algorithm from "Implementations of a New Theorem for Computing Bounds for Positive Roots of Polynomials", by Akritas, Strzebo'nski, and Vigklas.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: cl_maximum_root_local_max([RIF(-1), RIF(0), RIF(1)])
1.41421356237310
```

class `sage.rings.polynomial.real_roots.`**`context`**

Bases: `object`

A simple context class, which is passed through parts of the real root isolation algorithm to avoid global variables.

Holds logging information, a random number generator, and the target machine wordsize.

`get_be_log`()

`get_dc_log`()

`sage.rings.polynomial.real_roots.`**`de_casteljau_doublevec`**(*c*, *x*)

Given a polynomial in Bernstein form with floating-point coefficients over the region [0 .. 1], and a split point x, use de Casteljau's algorithm to give polynomials in Bernstein form over [0 .. x] and [x .. 1].

This function will work for an arbitrary rational split point x, as long as 0 < x < 1; but it has a specialized code path for x==1/2.

INPUT:

- `c` – vector of coefficients of polynomial in Bernstein form
- `x` – rational splitting point; 0 < x < 1

OUTPUT:

- `c1` – coefficients of polynomial over range [0 .. x]
- `c2` – coefficients of polynomial over range [x .. 1]
- `err_inc` – number of half-ulps by which error intervals widened

EXAMPLES:

```
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, 0.3111111111111117, 0.20740740740740746, 0.13827160493827165, 0.0921
sage: de_casteljau_doublevec(c, 7/22)  # rel tol
((0.7, 0.4772727272727273, 0.3254132231404959, 0.22187265214124724, 0.15127680827812312, 0.10314
```

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:

- `c1` – coefficients of polynomial over range [0 .. x]
- `c2` – coefficients of polynomial over range [x .. 1]
- `err_inc` – amount by which error intervals widened

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: c = vector(ZZ, [1048576, 0, 0, 0, 0, 0])
sage: de_casteljau_intvec(c, 20, 1/2, 1)
((1048576, 524288, 262144, 131072, 65536, 32768), (32768, 0, 0, 0, 0, 0), 1)
sage: de_casteljau_intvec(c, 20, 1/3, 1)
((1048576, 699050, 466033, 310689, 207126, 138084), (138084, 0, 0, 0, 0, 0), 1)
sage: de_casteljau_intvec(c, 20, 7/22, 1)
((1048576, 714938, 487457, 332357, 226607, 154505), (154505, 0, 0, 0, 0, 0), 1)
```

sage.rings.polynomial.real_roots.**degree_reduction_next_size**(*n*)

Given n (a polynomial degree), returns either a smaller integer or None. This defines the sequence of degrees followed by our degree reduction implementation.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: degree_reduction_next_size(1000)
30
sage: degree_reduction_next_size(20)
15
sage: degree_reduction_next_size(3)
2
sage: degree_reduction_next_size(2) is None
True
```

sage.rings.polynomial.real_roots.**dprod_imatrow_vec**(*m*, *v*, *k*)

Computes the dot product of row k of the matrix m with the vector v (that is, compute one element of the product m*v).

If v has more elements than m has columns, then elements of v are selected using subsample_vec.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: m = matrix(3, range(9))
sage: dprod_imatrow_vec(m, vector(ZZ, [1, 0, 0, 0]), 1)
0
sage: dprod_imatrow_vec(m, vector(ZZ, [0, 1, 0, 0]), 1)
3
sage: dprod_imatrow_vec(m, vector(ZZ, [0, 0, 1, 0]), 1)
4
sage: dprod_imatrow_vec(m, vector(ZZ, [0, 0, 0, 1]), 1)
5
sage: dprod_imatrow_vec(m, vector(ZZ, [1, 0, 0]), 1)
3
sage: dprod_imatrow_vec(m, vector(ZZ, [0, 1, 0]), 1)
4
sage: dprod_imatrow_vec(m, vector(ZZ, [0, 0, 1]), 1)
5
sage: dprod_imatrow_vec(m, vector(ZZ, [1, 2, 3]), 1)
26
```

sage.rings.polynomial.real_roots.**get_realfield_rndu**(*n*)

A simple cache for RealField fields (with rounding set to round-to-positive-infinity).

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: get_realfield_rndu(20)
Real Field with 20 bits of precision and rounding RNDU
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 <= x <= 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 P1 over [a .. b] and P2 over [b .. c], where P1 and P2 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.

region()

region_width()

try_rand_split(*ctx*, *logging_note*)

Compute a random split point r (using the random number generator embedded in ctx). We require 1/4 <= r < 3/4 (to ensure that recursive algorithms make progress).

Then, try doing a de Casteljau split of this polynomial at r, resulting in polynomials p1 and p2. If we see that the sign of this polynomial is determined at r, then return (p1, p2, r); otherwise, return None.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([50, 20, -90, -70, 200], error=5)
sage: bp1, bp2, _ = bp.try_rand_split(mk_context(), None)
sage: bp1
<IBP: (50, 29, -27, -56, -11) + [0 .. 6) over [0 .. 43/64]>
sage: bp2
<IBP: (-11, 10, 49, 111, 200) + [0 .. 6) over [43/64 .. 1]>
sage: bp1, bp2, _ = bp.try_rand_split(mk_context(seed=42), None)
sage: bp1
<IBP: (50, 32, -11, -41, -29) + [0 .. 6) over [0 .. 583/1024]>
sage: bp2
<IBP: (-29, -20, 13, 83, 200) + [0 .. 6) over [583/1024 .. 1]>
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, _ = bp.try_rand_split(mk_context(), None)
sage: bp1   # rel tol
<IBP: (0.5, 0.2984375, -0.2642578125, -0.5511661529541015, -0.3145806974172592) + [-0.1 .. 0
sage: bp2   # rel tol
<IBP: (-0.3145806974172592, -0.19903896331787108, 0.04135986328125002, 0.43546875, 0.99) + [
```

try_split(*ctx*, *logging_note*)

Try doing a de Casteljau split of this polynomial at 1/2, resulting in polynomials p1 and p2. If we see that the sign of this polynomial is determined at 1/2, then return (p1, p2, 1/2); otherwise, return None.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([50, 20, -90, -70, 200], error=5)
sage: bp1, bp2, _ = bp.try_split(mk_context(), None)
sage: bp1
<IBP: (50, 35, 0, -29, -31) + [0 .. 6) over [0 .. 1/2]>
sage: bp2
<IBP: (-31, -33, -8, 65, 200) + [0 .. 6) over [1/2 .. 1]>
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, _ = bp.try_split(mk_context(), None)
sage: bp1
<IBP: (0.5, 0.35, 0.0, -0.2875, -0.369375) + [-0.1 .. 0.01] over [0 .. 1/2]>
sage: bp2
<IBP: (-0.369375, -0.45125, -0.3275, 0.14500000000000002, 0.99) + [-0.1 .. 0.01] over [1/2 .
```

variations()

Consider a polynomial (written in either the normal power basis or the Bernstein basis). Take its list of coefficients, omitting zeroes. Count the number of positions in the list where the sign of one coefficient is opposite the sign of the next coefficient.

This count is the number of sign variations of the polynomial. According to Descartes' rule of signs, the number of real roots of the polynomial (counted with multiplicity) in a certain interval is always less than or equal to the number of sign variations, and the difference is always even. (If the polynomial is written in the power basis, the region is the positive reals; if the polynomial is written in the Bernstein basis over a particular region, then we count roots in that region.)

In particular, a polynomial with no sign variations has no real roots in the region, and a polynomial with one sign variation has one real root in the region.

In an interval Bernstein polynomial, we do not necessarily know the signs of the coefficients (if some of the coefficient intervals contain zero), so the polynomials contained by this interval polynomial may not all have the same number of sign variations. However, we can compute a range of possible numbers of sign variations.

This function returns the range, as a 2-tuple of integers.

class `sage.rings.polynomial.real_roots.`**`interval_bernstein_polynomial_float`**

Bases: `sage.rings.polynomial.real_roots.interval_bernstein_polynomial`

This is the subclass of interval_bernstein_polynomial where polynomial coefficients are represented using floating-point numbers.

In the floating-point representation, each coefficient is represented as an IEEE double-precision float A, and the (shared) lower and upper interval widths E1 and E2. These represent the coefficients (A+E1)*2^n <= c <= (A+E2)*2^n.

Note that we always have E1 <= 0 <= E2. Also, each floating-point coefficient has absolute value less than one.

(Note that mk_ibpf is a simple helper function for creating elements of interval_bernstein_polynomial_float in doctests.)

EXAMPLES:

```
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
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; leve
sage: bp.variations()
(3, 3)
```

`as_float`()

`de_casteljau`(*ctx*, *mid*, *msign=0*)

Uses de Casteljau's algorithm to compute the representation of this polynomial in a Bernstein basis over new regions.

INPUT:

- `mid` – where to split the Bernstein basis region; 0 < mid < 1
- `msign` – default 0 (unknown); the sign of this polynomial at mid

OUTPUT:

•bp1, bp2 – the new interval Bernstein polynomials

•ok – a boolean; True if the sign of the original polynomial at mid is known

EXAMPLES:

```
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) + [-0.1 .. 0.01] over [0 .. 1/2]>
sage: bp2
<IBP: (-0.369375, -0.45125, -0.3275, 0.14500000000000002, 0.99) + [-0.1 .. 0.01] over [1/2 .
sage: bp1, bp2, ok = bp.de_casteljau(ctx, 2/3)
sage: bp1 # rel tol 2e-16
<IBP: (0.5, 0.30000000000000004, -0.2555555555555555, -0.5444444444444444, -0.32172839506172
sage: bp2  # rel tol 3e-15
<IBP: (-0.32172839506172846, -0.21037037037037046, 0.028888888888888797, 0.4266666666666666,
sage: bp1, bp2, ok = bp.de_casteljau(ctx, 7/39)
sage: bp1  # rel tol
<IBP: (0.5, 0.4461538461538461, 0.36653517422748183, 0.27328680523946786, 0.1765692706232836
sage: bp2  # rel tol
<IBP: (0.1765692706232836, -0.26556803047927313, -0.7802038132807364, -0.3966666666666666, 0
```

get_msb_bit()

Returns an approximation of the log2 of the maximum of the absolute values of the coefficients, as an integer.

slope_range()

Compute a bound on the derivative of this polynomial, over its region.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpf([0.5, 0.2, -0.9, -0.7, 0.99], neg_err=-0.1, pos_err=0.01)
sage: bp.slope_range().str(style='brackets')
'[-4.8400000000000017 .. 7.2000000000000011]'
```

class sage.rings.polynomial.real_roots.**interval_bernstein_polynomial_integer**

Bases: sage.rings.polynomial.real_roots.interval_bernstein_polynomial

This is the subclass of interval_bernstein_polynomial where polynomial coefficients are represented using 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:

```
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([1, 2, 3], error=5); print bp
degree 2 IBP with 2-bit coefficients
sage: bp
<IBP: (1, 2, 3) + [0 .. 5)>
sage: bp.variations()
(0, 0)
sage: bp = mk_ibpi([-3, -1, 1, -1, -3, -1], lower=1, upper=5/4, usign=1, error=2, scale_log2=-3,
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
sage: bp.variations()
(3, 3)
```

as_float()

Compute an interval_bernstein_polynomial_float which contains (or bounds) all the polynomials this interval polynomial contains (or bounds).

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([50, 20, -90, -70, 200], error=5)
sage: print bp.as_float()
degree 4 IBP with floating-point coefficients
sage: bp.as_float()
<IBP: ((0.1953125, 0.078125, -0.3515625, -0.2734375, 0.78125) + [-1.12757025938e-16 .. 0.019
```

de_casteljau(*ctx*, *mid*, *msign=0*)

Uses de Casteljau's algorithm to compute the representation of this polynomial in a Bernstein basis over new regions.

INPUT:

- `mid` – where to split the Bernstein basis region; 0 < mid < 1
- `msign` – default 0 (unknown); the sign of this polynomial at mid

OUTPUT:

- `bp1`, `bp2` – the new interval Bernstein polynomials
- `ok` – a boolean; True if the sign of the original polynomial at mid is known

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([50, 20, -90, -70, 200], error=5)
sage: ctx = mk_context()
sage: bp1, bp2, ok = bp.de_casteljau(ctx, 1/2)
sage: bp1
<IBP: (50, 35, 0, -29, -31) + [0 .. 6) over [0 .. 1/2]>
sage: bp2
<IBP: (-31, -33, -8, 65, 200) + [0 .. 6) over [1/2 .. 1]>
sage: bp1, bp2, ok = bp.de_casteljau(ctx, 2/3)
sage: bp1
<IBP: (50, 30, -26, -55, -13) + [0 .. 6) over [0 .. 2/3]>
sage: bp2
<IBP: (-13, 8, 47, 110, 200) + [0 .. 6) over [2/3 .. 1]>
sage: bp1, bp2, ok = bp.de_casteljau(ctx, 7/39)
sage: bp1
<IBP: (50, 44, 36, 27, 17) + [0 .. 6) over [0 .. 7/39]>
sage: bp2
<IBP: (17, -26, -75, -22, 200) + [0 .. 6) over [7/39 .. 1]>
```

down_degree(*ctx*, *max_err*, *exp_err_shift*)

Compute an interval_bernstein_polynomial_integer which bounds all the polynomials this interval polynomial bounds, but is of lesser degree.

During the computation, we find an "expected error" expected_err, which is the error inherent in our approach (this depends on the degrees involved, and is proportional to the error of the current polynomial).

We require that the error of the new interval polynomial be bounded both by max_err, and by expected_err << exp_err_shift. If we find such a polynomial p, then we return a pair of p and some debugging/logging information. Otherwise, we return the pair (None, None).

If the resulting polynomial would have error more than 2^17, then it is downscaled before returning.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([0, 100, 400, 903], error=2)
sage: ctx = mk_context()
sage: bp
<IBP: (0, 100, 400, 903) + [0 .. 2)>
sage: dbp, _ = bp.down_degree(ctx, 10, 32)
sage: dbp
<IBP: (-1, 148, 901) + [0 .. 4); level 1; slope_err 0.?e2>
```

down_degree_iter(*ctx*, *max_scale*)

Compute a degree-reduced version of this interval polynomial, by iterating down_degree.

We stop when degree reduction would give a polynomial which is too inaccurate, meaning that either we think the current polynomial may have more roots in its region than the degree of the reduced polynomial, or that the least significant accurate bit in the result (on the absolute scale) would be larger than 1 << max_scale.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([0, 100, 400, 903, 1600, 2500], error=2)
sage: ctx = mk_context()
sage: bp
<IBP: (0, 100, 400, 903, 1600, 2500) + [0 .. 2)>
sage: rbp = bp.down_degree_iter(ctx, 6)
sage: rbp
<IBP: (-4, 249, 2497) + [0 .. 9); level 2; slope_err 0.?e3>
```

downscale(*bits*)

Compute an interval_bernstein_polynomial_integer which contains (or bounds) all the polynomials this interval polynomial contains (or bounds), but uses "bits" fewer bits.

EXAMPLES:

```
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:

```
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')
'[294.00000000000000 .. 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:

```
sage: from sage.rings.polynomial.real_roots import *
sage: intvec_to_doublevec(vector(ZZ, [1, 2, 3, 4, 5]), 3)
((0.125, 0.25, 0.375, 0.5, 0.625), -1.1275702593849246e-16, 0.37500000000000017, 3)
```

class `sage.rings.polynomial.real_roots.`**`island`**

Bases: `object`

This implements the island portion of my ocean-island root isolation algorithm. See the documentation for class ocean, for more information on the overall algorithm.

Island root refinement starts with a Bernstein polynomial whose region is the whole island (or perhaps slightly more than the island in certain cases). There are two subalgorithms; one when looking at a Bernstein polynomial covering a whole island (so we know that there are gaps on the left and right), and one when looking at a Bernstein polynomial covering the left segment of an island (so we know that there is a gap on the left, but the right is in the middle of an island). An important invariant of the left-segment subalgorithm over the region [l .. r] is that it always finds a gap [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*)
A simple class to map linearly between original coordinates (ranging from [lower .. upper]) and ocean coordinates (ranging from [0 .. 1]).

from_ocean(*region*)

to_ocean(*region*)

sage.rings.polynomial.real_roots.**max_abs_doublevec**(*c*)
Given a floating-point vector, return the maximum of the absolute values of its elements.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: max_abs_doublevec(vector(RDF, [0.1, -0.767, 0.3, 0.693]))
0.767
```

sage.rings.polynomial.real_roots.**max_bitsize_intvec_doctest**(*b*)

sage.rings.polynomial.real_roots.**maximum_root_first_lambda**(*p*)
Given a polynomial with real coefficients, computes an upper bound on its largest real root, using the first-lambda algorithm from "Implementations of a New Theorem for Computing Bounds for Positive Roots of Polynomials", by Akritas, Strzebo'nski, and Vigklas.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: maximum_root_first_lambda((x-1)*(x-2)*(x-3))
6.00000000000001
sage: maximum_root_first_lambda((x+1)*(x+2)*(x+3))
0
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, Strzebo'nski, 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))
12.0000000000001
sage: maximum_root_local_max((x+1)*(x+2)*(x+3))
0.000000000000000
sage: maximum_root_local_max(x^2 - 1)
1.41421356237310
```

sage.rings.polynomial.real_roots.**min_max_delta_intvec**(*a*, *b*)

Given two integer vectors a and b (of equal, nonzero length), return a pair of the minimum and maximum values taken on by a[i] - b[i].

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: a = vector(ZZ, [10, -30])
sage: b = vector(ZZ, [15, -60])
sage: min_max_delta_intvec(a, b)
(30, -5)
```

sage.rings.polynomial.real_roots.**min_max_diff_doublevec**(*c*)

Given a floating-point vector b = (b0, ..., bn), compute the minimum and maximum values of b_{j+1} - b_j.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: min_max_diff_doublevec(vector(RDF, [1, 7, -2]))
(-9.0, 6.0)
```

sage.rings.polynomial.real_roots.**min_max_diff_intvec**(*b*)

Given an integer vector b = (b0, ..., bn), compute the minimum and maximum values of b_{j+1} - b_j.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: min_max_diff_intvec(vector(ZZ, [1, 7, -2]))
(-9, 6)
```

sage.rings.polynomial.real_roots.**mk_context**(*do_logging=False*, *seed=0*, *wordsize=32*)

A simple wrapper for creating context objects with coercions, defaults, etc.

For use in doctests.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: mk_context(do_logging=True, seed=3, wordsize=64)
root isolation context: seed=3; do_logging=True; wordsize=64
```

sage.rings.polynomial.real_roots.**mk_ibpf**(*coeffs*, *lower=0*, *upper=1*, *lsign=0*, *usign=0*, *neg_err=0*, *pos_err=0*, *scale_log2=0*, *level=0*, *slope_err=None*)

A simple wrapper for creating interval_bernstein_polynomial_float objects with coercions, defaults, etc.

For use in doctests.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: print mk_ibpf([0.5, 0.2, -0.9, -0.7, 0.99], pos_err=0.1, neg_err=-0.01)
degree 4 IBP with floating-point coefficients
```

`sage.rings.polynomial.real_roots.`**`mk_ibpi`**(*coeffs*, *lower=0*, *upper=1*, *lsign=0*, *usign=0*, *error=1*, *scale_log2=0*, *level=0*, *slope_err=None*)

A simple wrapper for creating interval_bernstein_polynomial_integer objects with coercions, defaults, etc.

For use in doctests.

EXAMPLES:

```
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 `sage.rings.polynomial.real_roots.`**`ocean`**

Bases: `object`

Given the tools we've defined so far, there are many possible root isolation algorithms that differ on where to select split points, what precision to work at when, and when to attempt degree reduction.

Here we implement one particular algorithm, which I call the ocean-island algorithm. We start with an interval Bernstein polynomial defined over the region [0 .. 1]. This region is the "ocean". Using de Casteljau's algorithm and Descartes' rule of signs, we divide this region into subregions which may contain roots, and subregions which are guaranteed not to contain roots. Subregions which may contain roots are "islands"; subregions known not to contain roots are "gaps".

All the real root isolation work happens in class island. See the documentation of that class for more information.

An island can be told to refine itself until it contains only a single root. This may not succeed, if the island's interval Bernstein polynomial does not have enough precision. The ocean basically loops, refining each of its islands, then increasing the precision of islands which did not succeed in isolating a single root; until all islands are done.

Increasing the precision of unsuccessful islands is done in a single pass using split_for_target(); this means it is possible to share work among multiple islands.

`all_done`()

Returns true iff all islands are known to contain exactly one root.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1/3, -22/7, 193/71, -14
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:

```
sage: from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1/3, -22/7, 193/71, -14
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:

```
sage: from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1/3, -22/7, 193/71, -14
sage: oc
ocean with precision 120 and 1 island(s)
sage: oc.find_roots()
sage: oc
ocean with precision 120 and 3 island(s)
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1, 0, -1111/2, 0, 11108
sage: oc.find_roots()
sage: oc
ocean with precision 240 and 3 island(s)
```

increase_precision()

Increase the precision of the interval Bernstein polynomial held by any islands which are not done. (In normal use, calls to this function are separated by calls to self.refine_all().)

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1/3, -22/7, 193/71, -14
sage: oc
ocean with precision 120 and 1 island(s)
sage: oc.increase_precision()
sage: oc.increase_precision()
sage: oc.increase_precision()
sage: oc
ocean with precision 960 and 1 island(s)
```

refine_all()

Refine all islands which are not done (which are not known to contain exactly one root).

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1/3, -22/7, 193/71, -14
sage: oc
ocean with precision 120 and 1 island(s)
sage: oc.refine_all()
sage: oc
ocean with precision 120 and 3 island(s)
```

reset_root_width(*isle_num*, *target_width*)

Require that the isle_num island have a width at most target_width.

If this is followed by a call to find_roots(), then the corresponding root will be refined to the specified width.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([-1, -1, 1]), lmap)
sage: oc.find_roots()
sage: oc.roots()
[(1/2, 3/4)]
sage: oc.reset_root_width(0, 1/2^200)
sage: oc.find_roots()
sage: oc
ocean with precision 240 and 1 island(s)
sage: RR(RealIntervalField(300)(oc.roots()[0]).absolute_diameter()).log2()
-232.668979560890
```

roots()

Return the locations of all islands in this ocean. (If run after find_roots(), this is the location of all roots in the ocean.)

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1/3, -22/7, 193/71, -14
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, 11108
sage: oc.find_roots()
sage: oc.roots()
[(95761241267509487747625/9671406556917033397649408, 191522482605387719863145/19342813113834
```

sage.rings.polynomial.real_roots.**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)
)
```

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
sage: real_roots(x^5 * (x^2 - 9999)^2 - 1)
[((-29274496381311/9007199254740992, 419601125186091/2251799813685248), 1), ((2126658450145849453
sage: real_roots(x^5 * (x^2 - 9999)^2 - 1, seed=42)
[((-123196838480289/18014398509481984, 293964743458749/9007199254740992), 1), ((8307259573979551
sage: real_roots(x^5 * (x^2 - 9999)^2 - 1, wordsize=64)
[((-62866503803202151050003/19342813113834066795298816, 901086554512564177624143/4835703278458516
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)
[((209/256, 593/512), 1)]
sage: real_roots(x*(x-1)*(x-2), bounds=(0, 2))
[((0, 0), 1), ((81/128, 337/256), 1), ((2, 2), 1)]
sage: real_roots(x*(x-1)*(x-2), bounds=(0, 2), retval='algebraic_real')
[(0, 1), (1, 1), (2, 1)]
sage: v = 2^40
sage: real_roots((x^2-1)^2 * (x^2 - (v+1)/v))
[((-12855504354077768210885019021174120740504020581912910106032833/128555043540719222043356967386
sage: real_roots(x^2 - 2)
[((-3/2, -1), 1), ((1, 3/2), 1)]
sage: real_roots(x^2 - 2, retval='interval')
[(-2.?, 1), (2.?, 1)]
sage: real_roots(x^2 - 2, max_diameter=1/2^30)
[((-22506280506048041472675379598886543645348790970912519198456805737131269246430553365310109/15
sage: real_roots(x^2 - 2, retval='interval', max_diameter=1/2^500)
[(-1.41421356237309504880168872420969807856967187537694807317667973799073247846210703885038753432
sage: ar_rts = real_roots(x^2 - 2, retval='algebraic_real'); ar_rts
[(-1.414213562373095?, 1), (1.414213562373095?, 1)]
sage: ar_rts[0][0]^2 - 2 == 0
True
sage: v = 2^40
sage: real_roots((x-1) * (x-(v+1)/v), retval='interval')
[(1.000000000000?, 1), (1.000000000001?, 1)]
sage: v = 2^60
sage: real_roots((x-1) * (x-(v+1)/v), retval='interval')
[(1.000000000000000000?, 1), (1.000000000000000001?, 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')
[((-1713/335, -689/335), 1), ((-2067/2029, -689/1359), 1), ((0, 0), 1), ((499/525, 1173/875), 1)
sage: real_roots((x+3)*(x+1)*x*(x-1)*(x-2), strategy='warp', retval='algebraic_real')
[(-3.000000000000000?, 1), (-1.000000000000000?, 1), (0, 1), (1.000000000000000?, 1), (2.0000000
sage: ar_rts = real_roots(x-1, retval='algebraic_real')
sage: ar_rts[0][0] == 1
True
```

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^5
sage: cc = real_roots(p, retval='algebraic_real')[2][0] # long time
sage: RealField(180)(cc)                                 # long time
1.3035772690342963912570991121525518907307025046594049
```

Now we play with algebraic real coefficients.

```
sage: x = polygen(AA)
sage: p = (x - 1) * (x - sqrt(AA(2))) * (x - 2)
sage: real_roots(p)
[((499/525, 2171/1925), 1), ((1173/875, 2521/1575), 1), ((337/175, 849/175), 1)]
sage: ar_rts = real_roots(p, retval='algebraic_real'); ar_rts
[(1.000000000000000?, 1), (1.414213562373095?, 1), (2.000000000000000?, 1)]
sage: ar_rts[1][0]^2 == 2
True
sage: ar_rts = real_roots(x*(x-1), retval='algebraic_real')
sage: ar_rts[0][0] == 0
True
sage: p2 = p * (p - 1/100); p2
x^6 - 8.82842712474619?*x^5 + 31.97056274847714?*x^4 - 60.77955262170047?*x^3 + 63.9852676325780
sage: real_roots(p2, retval='interval')
[(1.00?, 1), (1.1?, 1), (1.38?, 1), (1.5?, 1), (2.00?, 1), (2.1?, 1)]
sage: p = (x - 1) * (x - sqrt(AA(2)))^2 * (x - 2)^3 * sqrt(AA(3))
sage: real_roots(p, retval='interval')
[(1.000000000000000?, 1), (1.414213562373095?, 2), (2.000000000000000?, 3)]
```

Check that #10803 is fixed

```
sage: f = 2503841067*x^13 - 15465014877*x^12 + 37514382885*x^11 - 44333754994*x^10 + 24138665092
sage: len(real_roots(f,seed=1))
13
```

`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, 0)
sage: root_bounds(x*(x+1))
(-1.00000000000000, 0)
sage: root_bounds((x+2)*(x-3))
(-2.44948974278317, 3.46410161513776)
sage: root_bounds(x^995 * (x^2 - 9999) - 1)
(-99.9949998749937, 141.414284992713)
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) subregion of [0 .. 1]), and a list of targets, uses de Casteljau's method to compute representations of the Bernstein polynomial over each target. Uses degree reduction as often as possible while maintaining the requested precision.

Each target is of the form (lgap, ugap, b). Suppose lgap.region() is (l1, l2), and ugap.region() is (u1, u2). Then we will compute an interval Bernstein polynomial over a region [l .. u], where l1 <= l <= l2 and u1 <= u <= u2. (split_for_targets() is free to select arbitrary region endpoints within these bounds; it picks endpoints which make the computation easier.) The third component of the target, b, is the maximum allowed scale_log2 of the result; this is used to decide when degree reduction is allowed.

The pair (l1, l2) can be replaced by None, meaning [-infinity .. 0]; or, (u1, u2) can be replaced by None, meaning [1 .. infinity].

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(1/1234567
sage: bps[0]
<IBP: (999992, 999992, 999992) + [0 .. 15) over [8613397477114467984778830327/10633823966279326
sage: bps[1]
<IBP: (-1562500, -1875001, -2222223, -2592593, -2969137, -3337450) + [0 .. 4) over [1/2 .. 28633
```

sage.rings.polynomial.real_roots.**subsample_vec_doctest**(*a*, *slen*, *llen*)

sage.rings.polynomial.real_roots.**taylor_shift1_intvec**(*c*)

Given a vector of integers c of length d+1, representing the coefficients of a degree-d polynomial p, modify the vector to perform a Taylor shift by 1 (that is, p becomes p(x+1)).

This is the straightforward algorithm, which is not asymptotically optimal.

Modifies the input vector; has no return value.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: p = (x-1)*(x-2)*(x-3)
sage: v = vector(ZZ, p.list())
sage: p, v
(x^3 - 6*x^2 + 11*x - 6, (-6, 11, -6, 1))
sage: taylor_shift1_intvec(v)
sage: p(x+1), v
(x^3 - 3*x^2 + 2*x, (0, 2, -3, 1))
```

sage.rings.polynomial.real_roots.**to_bernstein**(*p*, *low=0*, *high=1*, *degree=None*)

Given a polynomial p with integer coefficients, and rational bounds low and high, compute the exact rational Bernstein coefficients of p over the region [low .. high]. The optional parameter degree can be used to give a formal degree higher than the actual degree.

The return value is a pair (c, scale); c represents the same polynomial as p*scale. (If you only care about the roots of the polynomial, then of course scale can be ignored.)

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: to_bernstein(x)
([0, 1], 1)
sage: to_bernstein(x, degree=5)
([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)
```

sage.rings.polynomial.real_roots.**to_bernstein_warp**(*p*)

Given a polynomial p with rational coefficients, compute the exact rational Bernstein coefficients of p(x/(x+1)).

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: to_bernstein_warp(1 + x + x^2 + x^3 + x^4 + x^5)
[1, 1/5, 1/10, 1/10, 1/5, 1]
```

class sage.rings.polynomial.real_roots.**warp_map**(*neg*)

A class to map between original coordinates and ocean coordinates. If neg is False, then the original->ocean transform is x -> x/(x+1), and the ocean->original transform is x/(1-x); this maps between [0 .. infinity] and [0 .. 1]. If neg is True, then the original->ocean transform is x -> -x/(1-x), and the ocean->original transform is the same thing: -x/(1-x). This maps between [0 .. -infinity] and [0 .. 1].

from_ocean(*region*)

to_ocean(*region*)

sage.rings.polynomial.real_roots.**wordsize_rational**(*a*, *b*, *wordsize*)

Given rationals a and b, selects a de Casteljau split point r between a and b. An attempt is made to select an efficient split point (according to the criteria mentioned in the documentation for de_casteljau_intvec), with a bias towards split points near a.

In full detail:

Takes as input two rationals, a and b, such that 0<=a<=1, 0<=b<=1, and a!=b. Returns rational r, such that a<=r<=b or b<=r<=a. The denominator of r is a power of 2. Let m be min(r, 1-r), nm be numerator(m), and dml be log2(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 <= 1/8, or 7/8 <= a, then class 2 is preferred to class 1).

1. dml < wordsize
2. bitsize(nm) <= wordsize
3. bitsize(nm) <= 2*wordsize
4. bitsize(nm) <= 3*wordsize

...

11. bitsize(nm) <= (k-1)*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
sage: wordsize_rational(1/1234567890, 1/1234567891, 32)
933866427/1152921504606846976
sage: wordsize_rational(1/1234567890, 1/1234567891, 64)
4010925763784056541/4951760157141521099596496896
```

2.13 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:

```
sage: x = polygen(ZZ)
sage: (x^5 - x - 1).roots(ring=CIF)
[(1.167303978261419?, 1), (-0.764884433600585? - 0.352471546031727?*I, 1), (-0.764884433600585? + 0.3
```

`sage.rings.polynomial.complex_roots.`**`complex_roots`**(*p*, *skip_squarefree=False*, *retval='interval'*, *min_prec=0*)

Compute the complex roots of a given polynomial with exact coefficients (integer, rational, Gaussian rational, and algebraic coefficients are supported). Returns a list of pairs of a root and its multiplicity.

Roots are returned as a ComplexIntervalFieldElement; each interval includes exactly one root, and the intervals are disjoint.

By default, the algorithm will do a squarefree decomposition to get squarefree polynomials. The skip_squarefree parameter lets you skip this step. (If this step is skipped, and the polynomial has a repeated root, then the algorithm will loop forever!)

You can specify retval='interval' (the default) to get roots as complex intervals. The other options are retval='algebraic' to get elements of QQbar, or retval='algebraic_real' to get only the real roots, and to get them as elements of AA.

EXAMPLES:

```
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?
sage: v=complex_roots(x^2 + 27*x + 181)
```

Unfortunately due to numerical noise there can be a small imaginary part to each root depending on CPU, compiler, etc, and that affects the printing order. So we verify the real part of each root and check that the imaginary part is small in both cases:

```
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() < 1e25
True
sage: v[1][0].imag() < 1e25
True

sage: K.<im> = NumberField(x^2 + 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]
[-7.8887?e-31, 0, 7.8887?e-31, -7.8887?e-31*I, 7.8887?e-31*I]
```

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 x.real()
...       if tiny(x.real()): return CIF(0, x.imag())
sage: rts = complex_roots(p); type(rts[0][0]), sorted(map(smash, rts))
(<type 'sage.rings.complex_interval.ComplexIntervalFieldElement'>, [-1.618033988749895?, -0.6180
sage: rts = complex_roots(p, retval='algebraic'); type(rts[0][0]), sorted(map(smash, rts))
(<class 'sage.rings.qqbar.AlgebraicNumber'>, [-1.618033988749895?, -0.618033988749895?*I, 1.6180
sage: rts = complex_roots(p, retval='algebraic_real'); type(rts[0][0]), rts
(<class 'sage.rings.qqbar.AlgebraicReal'>, [(-1.618033988749895?, 1), (0.618033988749895?, 1)])
```

TESTS:

Verify that trac 12026 is fixed:

```
sage: f = matrix(QQ, 8, lambda i, j: 1/(i + j + 1)).charpoly()
sage: from sage.rings.polynomial.complex_roots import complex_roots
sage: len(complex_roots(f))
8
```

`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:

```
sage: x = polygen(ZZ)
sage: p = x^3 - 1
sage: rts = [CC.zeta(3)^i for i in range(0, 3)]
sage: from sage.rings.polynomial.complex_roots import interval_roots
sage: interval_roots(p, rts, 53)
[1, -0.500000000000000? + 0.866025403784439?*I, -0.500000000000000? - 0.866025403784439?*I]
sage: interval_roots(p, rts, 200)
[1, -0.500000000000000000000000000000000000000000000000000000000000? + 0.86602540378443864676372
```

`sage.rings.polynomial.complex_roots.`**`intervals_disjoint`**(*intvs*)

Given a list of complex intervals, check whether they are pairwise disjoint.

EXAMPLES:

```
sage: from sage.rings.polynomial.complex_roots import intervals_disjoint
sage: a = CIF(RIF(0, 3), 0)
sage: b = CIF(0, RIF(1, 3))
sage: c = CIF(RIF(1, 2), RIF(1, 2))
sage: d = CIF(RIF(2, 3), RIF(2, 3))
sage: intervals_disjoint([a,b,c,d])
False
sage: d2 = CIF(RIF(2, 3), RIF(2.001, 3))
sage: intervals_disjoint([a,b,c,d2])
True
```

`sage.rings.polynomial.complex_roots.`**`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:

```
sage: from sage.rings.polynomial.complex_roots import *
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
0.766044443118978? + 0.642787609686540?*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
```

2.14 Quotients of Univariate Polynomial Rings

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: S = R.quotient(x**3-3*x+1, 'alpha')
```

```
sage: S.gen()**2 in S
True
sage: x in S
True
sage: S.gen() in R
False
sage: 1 in S
True
```

`sage.rings.polynomial.polynomial_quotient_ring.`**`PolynomialQuotientRing`**(*ring*, *polynomial*, *names=None*)

Create a quotient of a polynomial ring.

INPUT:

- `ring` - a univariate polynomial ring in one variable.
- `polynomial` - element with 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 the polynomial.

EXAMPLES:

We create the quotient ring $\mathbf{Z}[x]/(x^3+7)$, and demonstrate many basic functions with it:

```
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

$$R = (\mathbf{F}_2[y]/(y^2+y+1))[x]/(x^3-5).$$

```
sage: A.<y> = PolynomialRing(GF(2)); A
Univariate Polynomial Ring in y over Finite Field of size 2 (using NTL)
sage: B = A.quotient(y^2 + y + 1, 'y2'); print 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(); print C
Univariate Polynomial Ring in x over Univariate Quotient Polynomial Ring in y2 over Finite Field
sage: R = C.quotient(x^3 - 5); print R
Univariate Quotient Polynomial Ring in xbar over Univariate Quotient Polynomial Ring in y2 over
```

Next we create a number field, but viewed as a quotient of a polynomial ring over **Q**:

```
sage: R = PolynomialRing(RationalField(), 'x'); x = R.gen()
sage: S = R.quotient(x^3 + 2*x - 5, 'a')
sage: 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 **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 trac 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
```

class `sage.rings.polynomial.polynomial_quotient_ring.`**`PolynomialQuotientRing_domain`**(*ring*, *polynomial*, *name=None*, *category=None*)

Bases: `sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic`, `sage.rings.ring.IntegralDomain`

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: S.<xbar> = R.quotient(x^2 + 1)
sage: S
Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 + 1
sage: loads(S.dumps()) == S
True
sage: loads(xbar.dumps()) == xbar
True
```

field_extension(*names*)

Takes a polynomial quotient ring, and returns a tuple with three elements: the NumberField defined by the same polynomial quotient ring, a homomorphism from its parent to the NumberField sending the generators to one another, and the inverse isomorphism.

OUTPUT:

- field
- homomorphism from self to field
- homomorphism from field to self

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Rationals())
sage: S.<alpha> = R.quotient(x^3-2)
sage: F.<b>, f, g = S.field_extension()
sage: F
Number Field in b with defining polynomial x^3 - 2
sage: a = F.gen()
sage: f(alpha)
b
sage: g(a)
alpha
```

Note that the parent ring must be an integral domain:

```
sage: R.<x> = GF(25,'f25')['x']
sage: S.<a> = R.quo(x^3 - 2)
sage: F, g, h = S.field_extension('b')
Traceback (most recent call last):
...
AttributeError: 'PolynomialQuotientRing_generic_with_category' object has no attribute 'fiel
```

Over a finite field, the corresponding field extension is not a number field:

```
sage: R.<x> = GF(25, 'a')['x']
sage: S.<a> = R.quo(x^3 + 2*x + 1)
sage: F, g, h = S.field_extension('b')
sage: h(F.0^2 + 3)
a^2 + 3
sage: g(x^2 + 2)
b^2 + 2
```

We do an example involving a relative number field:

```
sage: R.<x> = QQ['x']
sage: K.<a> = NumberField(x^3 - 2)
sage: S.<X> = K['X']
sage: Q.<b> = S.quo(X^3 + 2*X + 1)
sage: Q.field_extension('b')
```

```
(Number Field in b with defining polynomial X^3 + 2*X + 1 over its base field, ...
  Defn: b |--> b, Relative number field morphism:
  From: Number Field in b with defining polynomial X^3 + 2*X + 1 over its base field
  To:   Univariate Quotient Polynomial Ring in b over Number Field in a with defining polyno
  Defn: b |--> b
        a |--> a)
```

We slightly change the example above so it works.

```
sage: R.<x> = QQ['x']
sage: K.<a> = NumberField(x^3 - 2)
sage: S.<X> = K['X']
sage: f = (X+a)^3 + 2*(X+a) + 1
sage: f
X^3 + 3*a*X^2 + (3*a^2 + 2)*X + 2*a + 3
sage: Q.<z> = S.quo(f)
sage: F.<w>, g, h = Q.field_extension()
sage: c = g(z)
sage: f(c)
0
sage: h(g(z))
z
sage: g(h(w))
w
```

AUTHORS:

•Craig Citro (2006-08-07)

•William Stein (2006-08-06)

is_finite()

Return whether or not this quotient ring is finite.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: R.quo(1).is_finite()
True
sage: R.quo(x^3-2).is_finite()
False

sage: R.<x> = GF(9,'a')[]
sage: R.quo(2*x^3+x+1).is_finite()
True
sage: R.quo(2).is_finite()
True
```

class sage.rings.polynomial.polynomial_quotient_ring.**PolynomialQuotientRing_field**(*ring*, *polynomial*, *name=None*)

Bases: sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_domain, sage.rings.ring.Field

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<xbar> = R.quotient(x^2 + 1)
sage: S
```

```
Univariate Quotient Polynomial Ring in xbar over Rational Field with modulus x^2 + 1
sage: loads(S.dumps()) == S
True
sage: loads(xbar.dumps()) == xbar
True
```

base_field()

Alias for base_ring, when we're defined over a field.

complex_embeddings(*prec=53*)

Return all homomorphisms of this ring into the approximate complex field with precision prec.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^5 + x + 17
sage: k = R.quotient(f)
sage: v = k.complex_embeddings(100)
sage: [phi(k.0^2) for phi in v]
[2.9757207403766761469671194565, -2.4088994371613850098316292196 + 1.9025410530350528612407
```

class sage.rings.polynomial.polynomial_quotient_ring.**PolynomialQuotientRing_generic**(*ring, polynomial, name=None, category=None*)

Bases: sage.rings.ring.CommutativeRing

Quotient of a univariate polynomial ring by an ideal.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(8)); R
Univariate Polynomial Ring in x over Ring of integers modulo 8
sage: S.<xbar> = R.quotient(x^2 + 1); S
Univariate Quotient Polynomial Ring in xbar over Ring of integers modulo 8 with modulus x^2 + 1
```

We demonstrate object persistence.

```
sage: loads(S.dumps()) == S
True
sage: loads(xbar.dumps()) == xbar
True
```

We create some sample homomorphisms;

```
sage: R.<x> = PolynomialRing(ZZ)
sage: S = R.quo(x^2-4)
sage: f = S.hom([2])
sage: f
Ring morphism:
  From: Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 - 4
  To:   Integer Ring
  Defn: xbar |--> 2
sage: f(x)
2
sage: f(x^2 - 4)
```

```
0
sage: f(x^2)
4
```

TESTS:

By trac ticket trac ticket #11900, polynomial quotient rings use Sage's category framework. They do so in an unusual way: During their initialisation, they are declared to be objects in the category of quotients of commutative algebras over a base ring. However, if it is tested whether a quotient ring is actually a field, the category might be refined, which also includes a change of the class of the quotient ring and its newly created elements.

Thus, in order to document that this works fine, we go into some detail:

```
sage: P.<x> = QQ[]
sage: Q = P.quotient(x^2+2)
sage: Q.category()
Join of Category of integral domains
 and Category of commutative algebras over Rational Field
 and Category of subquotients of monoids
 and Category of quotients of semigroups
```

The test suite passes:

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

We verify that the elements belong to the correct element class. Also, we list the attributes that are provided by the element class of the category, and store the current class of the quotient ring:

```
sage: isinstance(Q.an_element(),Q.element_class)
True
sage: [s for s in dir(Q.category().element_class) if not s.startswith('_')]
['cartesian_product', 'is_idempotent', 'is_one', 'is_unit', 'lift', 'powers']
sage: first_class = Q.__class__
```

We try to find out whether Q is a field. Indeed it is, and thus its category, including its class and element class, is changed accordingly:

```
sage: Q in Fields()
True
sage: Q.category()
Join of Category of fields and Category of commutative algebras over Rational Field and Category
sage: first_class == Q.__class__
False
sage: [s for s in dir(Q.category().element_class) if not s.startswith('_')]
['cartesian_product', 'euclidean_degree', 'gcd', 'is_idempotent', 'is_one', 'is_unit', 'lcm', 'l
```

As one can see, the elements are now inheriting additional methods: lcm and gcd. Even though `Q.an_element()` belongs to the old and not to the new element class, it still inherits the new methods from the category of fields, thanks to `Element.__getattr__()`:

```
sage: e = Q.an_element()
sage: isinstance(e, Q.element_class)
False
sage: e.gcd(e+1)
1
```

Since the category has changed, we repeat the test suite. However, we have to skip the test for its elements, since $an_element$ has been cached in the previous run of the test suite, and we have already seen that its class is not matching the new element class:

```
sage: TestSuite(Q).run(skip=['_test_elements'])
```

Newly created elements are fine, though, and their test suite passes:

```
sage: TestSuite(Q(x)).run()
sage: isinstance(Q(x), Q.element_class)
True
```

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 $\mathbf{Q}(\sqrt{-5})$ has the same class group as its base:

```
sage: K.<a> = QuadraticField(-5)
sage: R.<x> = K[]
sage: S.<xbar> = R.quotient(x)
sage: S.S_class_group([])
[((2, -a + 1), 2)]
```

When we include the prime $(2, -a + 1)$, the S-class group becomes trivial:

```
sage: S.S_class_group([K.ideal(2, -a+1)])
[]
```

Here is an example where the base and the extension both contribute to the class group:

```
sage: K.<a> = QuadraticField(-5)
sage: K.class_group()
Class group of order 2 with structure C2 of Number Field in a with defining polynomial x^2 +
sage: R.<x> = K[]
sage: S.<xbar> = R.quotient(x^2 + 23)
sage: S.S_class_group([])
[((2, -a + 1, 1/2*xbar + 1/2, -1/2*a*xbar + 1/2*a + 1), 6)]
sage: S.S_class_group([K.ideal(3, a-1)])
[]
sage: S.S_class_group([K.ideal(2, a+1)])
[]
sage: S.S_class_group([K.ideal(a)])
[((2, -a + 1, 1/2*xbar + 1/2, -1/2*a*xbar + 1/2*a + 1), 6)]
```

Now we take an example over a nontrivial base with two factors, each contributing to the class group:

```
sage: K.<a> = QuadraticField(-5)
sage: R.<x> = K[]
sage: S.<xbar> = R.quotient((x^2 + 23)*(x^2 + 31))
sage: S.S_class_group([])
[((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/
```

By using the ideal (a), we cut the part of the class group coming from $x^2 + 31$ from 12 to 2, i.e. we lose a generator of order 6 (this was fixed in trac ticket #14489):

```
sage: S.S_class_group([K.ideal(a)])
[((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/
```

Note that all the returned values live where we expect them to:

```
sage: CG = S.S_class_group([])
sage: type(CG[0][0][1])
<class 'sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRing_generi
sage: type(CG[0][1])
<type 'sage.rings.integer.Integer'>
```

S_units(*S*, *proof=True*)

If self is an étale algebra D over a number field K (i.e. a quotient of $K[x]$ by a squarefree polynomial) and S is a finite set of places of K, return a list of generators of the group of S-units of D.

INPUT:

- `S` - a set of primes of the base field
- `proof` - if False, assume the GRH in computing the class group

OUTPUT:

A list of generators of the S-unit group, in the form `(gen, order)`, where `gen` is a unit of order `order`.

EXAMPLES:

```
sage: K.<a> = QuadraticField(-3)
sage: K.unit_group()
Unit group with structure C6 of Number Field in a with defining polynomial x^2 + 3
sage: K.<a> = QQ['x'].quotient(x^2 + 3)
sage: u,o = K.S_units([])[0]; u, o
(-1/2*a + 1/2, 6)
sage: u^6
1
sage: u^3
-1
sage: u^2
-1/2*a - 1/2

sage: K.<a> = QuadraticField(-3)
sage: y = polygen(K)
sage: L.<b> = K['y'].quotient(y^3 + 5); L
Univariate Quotient Polynomial Ring in b over Number Field in a with defining polynomial x^2
sage: L.S_units([])
[(-1/2*a + 1/2, 6), ((1/3*a - 1)*b^2 + 4/3*a*b + 5/6*a + 7/2, +Infinity), ((-1/3*a + 1)*b^2
sage: L.S_units([K.ideal(1/2*a - 3/2)])
[((-1/6*a - 1/2)*b^2 + (1/3*a - 1)*b + 4/3*a, +Infinity), (-1/2*a + 1/2, 6), ((1/3*a - 1)*b^
sage: L.S_units([K.ideal(2)])
[((-1/2*a + 1/2)*b^2 + (-a - 1)*b - 3, +Infinity), ((-1/6*a - 1/2)*b^2 + (1/3*a - 1)*b + 5/6
```

Note that all the returned values live where we expect them to:

```
sage: U = L.S_units([])
sage: type(U[0][0])
<class 'sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRing_field_
sage: type(U[0][1])
<type 'sage.rings.integer.Integer'>
sage: 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 $\mathbf{Z}[z]/(z^3 + z^2 + z + 1)$ is $\mathbf{Z}$.

```
sage: R.<z> = PolynomialRing(ZZ)
sage: S.<beta> = R.quo(z^3 + z^2 + z + 1)
sage: S.base_ring()
Integer Ring
```

Next we make a polynomial quotient ring over S and ask for its base ring.

```
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.

EXAMPLES:

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

sage: R.<x> = GF(9,'a')[]
sage: R.quo(2*x^3+x+1).cardinality()
729
sage: GF(9,'a').extension(2*x^3+x+1).cardinality()
729
sage: R.quo(2).cardinality()
1
```

characteristic()

Return the characteristic of this quotient ring.

This is always the same as the characteristic of the base ring.

EXAMPLES:

```
sage: R.<z> = PolynomialRing(ZZ)
sage: S.<a> = R.quo(z - 19)
sage: S.characteristic()
0
sage: R.<x> = PolynomialRing(GF(9,'a'))
sage: S = R.quotient(x^3 + 1)
```

```
sage: S.characteristic()
3
```

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.

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 class group.

INPUT:

•`proof` - if False, assume the GRH in computing the class group

OUTPUT:

A list of pairs `(gen, order)`, where `gen` is a tuple of elements generating a fractional ideal and `order` is the order of I in the class group.

EXAMPLES:

```
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
sage: K.<a> = QQ['x'].quotient(x^2 + 3)
sage: K.class_group()
[]
```

A trivial algebra over $\mathbf{Q}(\sqrt{-5})$ has the same class group as its base:

```
sage: K.<a> = QuadraticField(-5)
sage: R.<x> = K[]
sage: S.<xbar> = R.quotient(x)
sage: S.class_group()
[((2, -a + 1), 2)]
```

The same algebra constructed in a different way:

```
sage: K.<a> = QQ['x'].quotient(x^2 + 5)
sage: K.class_group(())
[((2, a + 1), 2)]
```

Here is an example where the base and the extension both contribute to the class group:

```
sage: K.<a> = QuadraticField(-5)
sage: K.class_group()
Class group of order 2 with structure C2 of Number Field in a with defining polynomial x^2 +
sage: R.<x> = K[]
sage: S.<xbar> = R.quotient(x^2 + 23)
sage: S.class_group()
[((2, -a + 1, 1/2*xbar + 1/2, -1/2*a*xbar + 1/2*a + 1), 6)]
```

Here is an example of a product of number fields, both of which contribute to the class group:

```
sage: R.<x> = QQ[]
sage: S.<xbar> = R.quotient((x^2 + 23)*(x^2 + 47))
sage: S.class_group()
[((1/12*xbar^2 + 47/12, 1/48*xbar^3 - 1/48*xbar^2 + 47/48*xbar - 47/48), 3), ((-1/12*xbar^2
```

Now we take an example over a nontrivial base with two factors, each contributing to the class group:

```
sage: K.<a> = QuadraticField(-5)
sage: R.<x> = K[]
sage: S.<xbar> = R.quotient((x^2 + 23)*(x^2 + 31))
sage: S.class_group()
[((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/
```

Note that all the returned values live where we expect them to:

```
sage: CG = S.class_group()
sage: type(CG[0][0][1])
<class 'sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRing_generi
sage: type(CG[0][1])
<type '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
sage: P.<t> = GF(3)[]
sage: Q = P.quo([2+t^2])
sage: F, R = Q.construction()
sage: F(R) == Q
True
```

AUTHOR:

– Simon King (2010-05)

cover_ring()

Return the polynomial ring of which this ring is the quotient.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^2-2)
sage: S.polynomial_ring()
Univariate Polynomial Ring in x over Rational Field
```

degree()

Return the degree of this quotient ring. The degree is the degree of the polynomial that we quotiented out by.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(GF(3))
sage: S = R.quotient(x^2005 + 1)
sage: S.degree()
2005
```

discriminant(*v=None*)

Return the discriminant of this ring over the base ring. This is by definition the discriminant of the polynomial that we quotiented out by.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^3 + x^2 + x + 1)
sage: S.discriminant()
-16
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: S = R.quotient(x^2 - 8)
sage: S.number_field().discriminant()
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:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^2 - 8, 'gamma')
sage: S.gen()
gamma
```

is_field(*proof=True*)

Return whether or not this quotient ring is a field.

EXAMPLES:

```
sage: R.<z> = PolynomialRing(ZZ)
sage: S = R.quo(z^2-2)
sage: S.is_field()
False
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^2 - 2)
sage: S.is_field()
True
```

krull_dimension()

x.__init__(...) initializes x; see help(type(x)) for signature

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
sage: T.ngens()
1
```

number_field()

Return the number field isomorphic to this quotient polynomial ring, if possible.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<alpha> = R.quotient(x^29 - 17*x - 1)
sage: K = S.number_field()
sage: K
Number Field in alpha with defining polynomial x^29 - 17*x - 1
sage: alpha = K.gen()
sage: alpha^29
17*alpha + 1
```

order()

Return the number of elements of this quotient ring.

EXAMPLES:

```
sage: F1.<a> = GF(2^7)
sage: P1.<x> = F1[]
sage: F2 = F1.extension(x^2+x+1, 'u')
sage: F2.order()
16384

sage: F1 = QQ
sage: P1.<x> = F1[]
sage: F2 = F1.extension(x^2+x+1, 'u')
sage: F2.order()
+Infinity
```

polynomial_ring()

Return the polynomial ring of which this ring is the quotient.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^2-2)
sage: S.polynomial_ring()
```

```
Univariate Polynomial Ring in x over Rational Field
```

random_element (**args*, ***kwds*)

Return a random element of this quotient ring.

INPUT:

- `*args, **kwds` - Arguments for randomization that are passed on to the `random_element` method of the polynomial ring, and from there to the base ring

OUTPUT:

- Element of this quotient ring

EXAMPLES:

```
sage: F1.<a> = GF(2^7)
sage: P1.<x> = F1[]
sage: F2 = F1.extension(x^2+x+1, 'u')
sage: F2.random_element()
(a^6 + 1)*u + a^5 + a^4 + a^3 + 1
```

retract (*x*)

Return the coercion of x into this polynomial quotient ring.

The rings that coerce into the quotient ring canonically are:

- this ring
- any canonically isomorphic ring
- anything that coerces into the ring of which this is the quotient

selmer_group (*S*, *m*, *proof=True*)

If self is an étale algebra D over a number field K (i.e. a quotient of $K[x]$ by a squarefree polynomial) and S is a finite set of places of K, compute the Selmer group $D(S,m)$. This is the subgroup of $D^*/(D^*)^m$ consisting of elements a such that $D(\sqrt[m]{a})/D$ is unramified at all primes of D lying above a place outside of S.

INPUT:

- `S` - A set of primes of the coefficient ring (which is a number field).
- `m` - a positive integer
- `proof` - if False, assume the GRH in computing the class group

OUTPUT:

A list of generators of $D(S,m)$.

EXAMPLES:

```
sage: K.<a> = QuadraticField(-5)
sage: R.<x> = K[]
sage: D.<T> = R.quotient(x)
sage: D.selmer_group((), 2)
[-1, 2]
sage: D.selmer_group([K.ideal(2, -a+1)], 2)
[2, -1]
sage: D.selmer_group([K.ideal(2, -a+1), K.ideal(3, a+1)], 2)
[2, -a - 1, -1]
sage: D.selmer_group((K.ideal(2, -a+1),K.ideal(3, a+1)), 4)
[2, -a - 1, -1]
```

```
sage: D.selmer_group([K.ideal(2, -a+1)], 3)
[2]
sage: D.selmer_group([K.ideal(2, -a+1), K.ideal(3, a+1)], 3)
[2, -a - 1]
sage: D.selmer_group([K.ideal(2, -a+1), K.ideal(3, a+1), K.ideal(a)], 3)
[2, -a - 1, a]
```

units (*proof=True*)

If this quotient ring is over a number field K, by a polynomial of nonzero discriminant, returns a list of generators of the units.

INPUT:

•`proof` - if False, assume the GRH in computing the class group

OUTPUT:

A list of generators of the unit group, in the form `(gen, order)`, where `gen` is a unit of order `order`.

EXAMPLES:

```
sage: K.<a> = QuadraticField(-3)
sage: K.unit_group()
Unit group with structure C6 of Number Field in a with defining polynomial x^2 + 3
sage: K.<a> = QQ['x'].quotient(x^2 + 3)
sage: u = K.units()[0][0]; u
-1/2*a + 1/2
sage: u^6
1
sage: u^3
-1
sage: u^2
-1/2*a - 1/2
sage: K.<a> = QQ['x'].quotient(x^2 + 5)
sage: K.units(())
[(-1, 2)]

sage: K.<a> = QuadraticField(-3)
sage: y = polygen(K)
sage: L.<b> = K['y'].quotient(y^3 + 5); L
Univariate Quotient Polynomial Ring in b over Number Field in a with defining polynomial x^2
sage: L.units()
[(-1/2*a + 1/2, 6), ((1/3*a - 1)*b^2 + 4/3*a*b + 5/6*a + 7/2, +Infinity), ((-1/3*a + 1)*b^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 o
sage: L.unit_group().gens()    # abstract generators
(u0, u1, u2)
sage: L.unit_group().gens_values()
[-1/2*a + 1/2, (1/3*a - 1)*b^2 + 4/3*a*b + 5/6*a + 7/2, (-1/3*a + 1)*b^2 + (2/3*a - 2)*b - 5
```

Note that all the returned values live where we expect them to:

```
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_element.PolynomialQuotientRing_field_
sage: type(U[0][1])
<type 'sage.rings.integer.Integer'>
sage: type(U[1][1])
<class 'sage.rings.infinity.PlusInfinity'>
```

`sage.rings.polynomial.polynomial_quotient_ring.`**`is_PolynomialQuotientRing`**`(`*x*`)`
x.__init__(...) initializes x; see help(type(x)) for signature

2.15 Elements of Quotients of Univariate Polynomial Rings

EXAMPLES: We create a quotient of a univariate polynomial ring over $\mathbf{Z}$.

```
sage: R.<x> = ZZ[]
sage: S.<a> = R.quotient(x^3 + 3*x -1)
sage: 2 * a^3
-6*a + 2
```

Next we make a univariate polynomial ring over $\mathbf{Z}[x]/(x^3 + 3x - 1)$.

```
sage: S1.<y> = S[]
```

And, we quotient out that by $y^2 + a$.

```
sage: T.<z> = S1.quotient(y^2+a)
```

In the quotient z^2 is $-a$.

```
sage: z^2
-a
```

And since $a^3 = -3x + 1$, we have:

```
sage: z^6
3*a - 1
```

```
sage: R.<x> = PolynomialRing(Integers(9))
sage: S.<a> = R.quotient(x^4 + 2*x^3 + x + 2)
sage: a^100
7*a^3 + 8*a + 7
```

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3-2)
sage: a
a
sage: a^3
2
```

For the purposes of comparison in Sage the quotient element a^3 is equal to x^3. This is because when the comparison is performed, the right element is coerced into the parent of the left element, and x^3 coerces to a^3.

```
sage: a == x
True
sage: a^3 == x^3
True
sage: x^3
x^3
sage: S(x^3)
2
```

AUTHORS:

- William Stein

class sage.rings.polynomial.polynomial_quotient_ring_element.**PolynomialQuotientRingElement** (*pa
po
no
mi
ch*

Bases: sage.rings.polynomial.polynomial_singular_interface.Polynomial_singular_repr, sage.structure.element.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)
xi
sage: (singular(xi)*singular(xi)).NF('std(0)')
-1
```

charpoly (*var*)

The characteristic polynomial of this element, which is by definition the characteristic polynomial of right multiplication by this element.

INPUT:

- var - string - the variable name

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quo(x^3 -389*x^2 + 2*x - 5)
sage: a.charpoly('X')
X^3 - 389*X^2 + 2*X - 5
```

fcp (*var='x'*)

Return the factorization of the characteristic polynomial of this element.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 -389*x^2 + 2*x - 5)
sage: a.fcp('x')
x^3 - 389*x^2 + 2*x - 5
sage: S(1).fcp('y')
(y - 1)^3
```

field_extension (*names*)

Given a polynomial with base ring a quotient ring, return a 3-tuple: a number field defined by the same polynomial, a homomorphism from its parent to the number field sending the generators to one another, and the inverse isomorphism.

INPUT:

- names - name of generator of output field

OUTPUT:

- field

- homomorphism from self to field
- homomorphism from field to self

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<alpha> = R.quotient(x^3-2)
sage: F.<a>, f, g = alpha.field_extension()
sage: F
Number Field in a with defining polynomial x^3 - 2
sage: a = F.gen()
sage: f(alpha)
a
sage: g(a)
alpha
```

Over a finite field, the corresponding field extension is not a number field:

```
sage: R.<x> = GF(25,'b')['x']
sage: S.<a> = R.quo(x^3 + 2*x + 1)
sage: F.<b>, g, h = a.field_extension()
sage: h(b^2 + 3)
a^2 + 3
sage: g(x^2 + 2)
b^2 + 2
```

We do an example involving a relative number field:

```
sage: R.<x> = QQ['x']
sage: K.<a> = NumberField(x^3-2)
sage: S.<X> = K['X']
sage: Q.<b> = S.quo(X^3 + 2*X + 1)
sage: F, g, h = b.field_extension('c')
```

Another more awkward example:

```
sage: R.<x> = QQ['x']
sage: K.<a> = NumberField(x^3-2)
sage: S.<X> = K['X']
sage: f = (X+a)^3 + 2*(X+a) + 1
sage: f
X^3 + 3*a*X^2 + (3*a^2 + 2)*X + 2*a + 3
sage: Q.<z> = S.quo(f)
sage: F.<w>, g, h = z.field_extension()
sage: c = g(z)
sage: f(c)
0
sage: h(g(z))
z
sage: g(h(w))
w
```

AUTHORS:

- 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
```

TESTS:

Raise an exception if the base ring is not a field (see trac ticket #13303):

```
sage: Z16x.<x> = Integers(16)[]
sage: S.<y> =  Z16x.quotient(x^2 + x + 1)
sage: (2*y).is_unit()
Traceback (most recent call last):
...
NotImplementedError: The base ring (=Ring of integers modulo 16) is not a field
```

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()

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:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 + 2*x - 5)
sage: a.matrix()
[ 0  1  0]
[ 0  0  1]
[ 5 -2  0]
```

minpoly()

The minimal polynomial of this element, which is by definition the minimal polynomial of right multiplication by this element.

norm()

The norm of this element, which is the norm of the matrix of right multiplication by this element.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 -389*x^2 + 2*x - 5)
sage: a.norm()
5
```

trace()

The trace of this element, which is the trace of the matrix of right multiplication by this element.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 -389*x^2 + 2*x - 5)
sage: a.trace()
389
```

CHAPTER

THREE

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

$$\mathbf{F}_2[x_1, ..., x_n]/\langle x_1^2 + x_1, ..., x_n^2 + x_n\rangle.$$

are implemented using the PolyBoRi library (cf. `sage.rings.polynomial.pbori`).

3.1 Term orders

Sage supports the following term orders:

Lexicographic (lex) $x^a < x^b$ if and only if there exists $1 \le i \le n$ such that $a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i < b_i$. This term order is called 'lp' in Singular.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
sage: x > y
True
sage: x > y^2
True
sage: x > 1
True
sage: x^1*y^2 > y^3*z^4
True
sage: x^3*y^2*z^4 < x^3*y^2*z^1
False
```

Degree reverse lexicographic (degrevlex) Let $\deg(x^a) = a_1 + a_2 + \cdots + a_n$, then $x^a < x^b$ if and only if $\deg(x^a) < \deg(x^b)$ or $\deg(x^a) = \deg(x^b)$ and there exists $1 \le i \le n$ 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 $1 \le i \le n$ such that $a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i < b_i$. This term order is called 'Dp' in Singular.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='deglex')
sage: x > y
True
sage: x > y^2*z
False
sage: x > 1
True
sage: x^1*y^2*z^3 > x^3*y^2*z^0
True
sage: x^2*y*z^2 > x*y^3*z
True
```

Inverse lexicographic (invlex) $x^a < x^b$ if and only if there exists $1 \le i \le n$ such that $a_n = b_n, \ldots, a_{i+1} = b_{i+1}, a_i < b_i$. This order is called 'rp' in Singular.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='invlex')
sage: x > y
False
sage: y > x^2
True
sage: x > 1
True
sage: x*y > z
False
```

This term order only makes sense in a non-commutative setting because if P is the ring $k[x_1, \ldots, x_n]$ and term order 'invlex' then it is equivalent to the ring $k[x_n, \ldots, x_1]$ with term order 'lex'.

Negative lexicographic (neglex) $x^a < x^b$ if and only if there exists $1 \le i \le n$ such that $a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i > b_i$. This term order is called 'ls' in Singular.

EXAMPLES:

```
sage: 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 $1 \le i \le n$ 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:

```
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 $1 \le i \le n$ 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:

```
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 $1 \le i \le n$ 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:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order=TermOrder('wdegrevlex',(1,2,3)))
sage: x > y
False
sage: x > x^2
False
sage: x > 1
True
sage: x^1*y^2 > x^2*z
True
sage: y*z > x^3*y
False
```

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 $1 \le i \le n$ 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:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order=TermOrder('wdeglex',(1,2,3)))
sage: x > y
False
sage: x > x^2
False
sage: x > 1
True
sage: x^1*y^2 > x^2*z
False
sage: y*z > x^3*y
False
```

Negative weighted degree reverse lexicographic (negwdegrevlex), positive integral weights 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 $1 \le i \le n$ such that $a_n = b_n, \dots, a_{i+1} = b_{i+1}, a_i > b_i$. This term order is called 'ws' in Singular.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order=TermOrder('negwdegrevlex',(1,2,3)))
sage: x > y
True
sage: x > x^2
True
sage: x > 1
False
sage: x^1*y^2 > x^2*z
True
sage: y*z > x^3*y
False
```

Degree negative lexicographic (degneglex) 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 $1 \le i \le n$ such that $a_1 = b_1, \dots, a_{i-1} = b_{i-1}, a_i > b_i$. This term order is called 'dp_asc' in PolyBoRi. Singular has the extra weight vector ordering '(r(1:n),rp)' for this purpose.

EXAMPLES:

```
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 $1 \le i \le n$ such that $a_1 = b_1, \dots, a_{i-1} = b_{i-1}, a_i < b_i$. This term order is called 'Ws' in Singular.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order=TermOrder('negwdeglex',(1,2,3)))
sage: x > y
True
sage: x > x^2
True
sage: x > 1
False
sage: x^1*y^2 > x^2*z
False
```

```
    sage: y*z > x^3*y
    False
```

Of these, only 'degrevlex', 'deglex', 'degneglex', 'wdegrevlex', 'wdeglex', 'invlex' and 'lex' are global orders.

Sage also supports matrix term order. Given a square matrix A,

$x^a <_A x^b$ if and only if $Aa < Ab$

where $<$ is the lexicographic term order.

EXAMPLE:

```
sage: m = matrix(2,[2,3,0,1]); m
[2 3]
[0 1]
sage: T = TermOrder(m); T
Matrix term order with matrix
[2 3]
[0 1]
sage: P.<a,b> = PolynomialRing(QQ,2,order=T)
sage: P
Multivariate Polynomial Ring in a, b over Rational Field
sage: a > b
False
sage: a^3 < b^2
True
sage: S = TermOrder('M(2,3,0,1)')
sage: T == S
True
```

Additionally all these monomial orders may be combined to product or block orders, defined as:

Let $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_m)$ 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 ($x^a = x^A$ 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.

EXAMPLE:

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.

```
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:

```
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:

```
sage: T = TermOrder("royalorder")
Traceback (most recent call last):
...
TypeError: 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, refactoring

class `sage.rings.polynomial.term_order.`**`TermOrder`**(*name='lex'*, *n=0*, *force=False*)

Bases: `sage.structure.sage_object.SageObject`

A term order.

See `sage.rings.polynomial.term_order` for details on supported term orders.

`blocks`()

Return the term order blocks of self.

NOTE:

This method has been added in trac ticket #11316. There used to be an *attribute* of the same name and the same content. So, it is a backward incompatible syntax change.

EXAMPLE:

```
sage: t=TermOrder('deglex',2)+TermOrder('lex',2)
sage: t.blocks()
(Degree lexicographic term order, Lexicographic term order)
```

`compare_tuples_block`(*f*, *g*)

Compares two exponent tuples with respect to the block order as specified when constructing this element.

INPUT:

- `f` - exponent tuple
- `g` - exponent tuple

EXAMPLE:

```
sage: P.<a,b,c,d,e,f>=PolynomialRing(QQbar, 6, order='degrevlex(3),degrevlex(3)')
sage: a > c^4 # indirect doctest
False
sage: a > e^4
True
```

compare_tuples_deglex (*f*, *g*)

Compares two exponent tuples with respect to the degree lexicographical term order.

INPUT:

- `f` - exponent tuple
- `g` - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='deglex')
sage: x > y^2 # indirect doctest
False
sage: x > 1
True
```

compare_tuples_degneglex (*f*, *g*)

Compares two exponent tuples with respect to the degree negative lexicographical term order.

INPUT:

- `f` - exponent tuple
- `g` - exponent tuple

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='degneglex')
sage: x*y > y*z # indirect doctest
False
sage: x*y > x
True
```

compare_tuples_degrevlex (*f*, *g*)

Compares two exponent tuples with respect to the degree reversed lexicographical term order.

INPUT:

- `f` - exponent tuple
- `g` - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='degrevlex')
sage: x > y^2 # indirect doctest
False
sage: x > 1
True
```

compare_tuples_invlex (*f*, *g*)

Compares two exponent tuples with respect to the inversed lexicographical term order.

INPUT:

- `f` - exponent tuple

- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='invlex')
sage: x > y^2 # indirect doctest
False
sage: x > 1
True
```

compare_tuples_lex (*f*, *g*)

Compares two exponent tuples with respect to the lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='lex')
sage: x > y^2 # indirect doctest
True
sage: x > 1
True
```

compare_tuples_matrix (*f*, *g*)

Compares two exponent tuples with respect to the matrix term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='m(1,3,1,0)')
sage: y > x^2 # indirect doctest
True
sage: y > x^3
False
```

compare_tuples_negdeglex (*f*, *g*)

Compares two exponent tuples with respect to the negative degree lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='negdeglex')
sage: x > y^2 # indirect doctest
True
sage: x > 1
False
```

compare_tuples_negdegrevlex (*f*, *g*)

Compares two exponent tuples with respect to the negative degree reverse lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='negdegrevlex')
sage: x > y^2 # indirect doctest
True
sage: x > 1
False
```

compare_tuples_neglex (*f, g*)

Compares two exponent tuples with respect to the negative lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='neglex')
sage: x > y^2 # indirect doctest
False
sage: x > 1
False
```

compare_tuples_negwdeglex (*f, g*)

Compares two exponent tuples with respect to the negative weighted degree lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: t = TermOrder('negwdeglex',(3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)
sage: x > y^2 # indirect doctest
True
sage: x^2 > y^3
True
```

compare_tuples_negwdegrevlex (*f, g*)

Compares two exponent tuples with respect to the negative weighted degree reverse lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: t = TermOrder('negwdegrevlex',(3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)
sage: x > y^2 # indirect doctest
True
sage: x^2 > y^3
True
```

compare_tuples_wdeglex (*f*, *g*)

Compares two exponent tuples with respect to the weighted degree lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: t = TermOrder('wdeglex',(3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)
sage: x > y^2 # indirect doctest
False
sage: x > y
True
```

compare_tuples_wdegrevlex (*f*, *g*)

Compares two exponent tuples with respect to the weighted degree reverse lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: t = TermOrder('wdegrevlex',(3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)
sage: x > y^2 # indirect doctest
False
sage: x^2 > y^3
True
```

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

EXAMPLE:

```
sage: P.<a,b,c,d,e,f>=PolynomialRing(QQbar, 6, order='degrevlex(3),degrevlex(3)')
sage: f = a + c^4; f.lm() # indirect doctest
c^4
sage: g = a + e^4; g.lm()
a
```

greater_tuple_deglex (*f*, *g*)

Return the greater exponent tuple with respect to the total degree lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='deglex')
sage: f = x + y; f.lm() # indirect doctest
x
sage: f = x + y^2*z; f.lm()
y^2*z
```

This method is called by the lm/lc/lt methods of `MPolynomial_polydict`.

greater_tuple_degneglex (*f*, *g*)

Return the greater exponent tuple with respect to the degree negative lexicographical term order.

INPUT:

- `f` - exponent tuple
- `g` - exponent tuple

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='degneglex')
sage: f = x + y; f.lm() # indirect doctest
y
sage: f = x + y^2*z; f.lm()
y^2*z
```

This method is called by the lm/lc/lt methods of `MPolynomial_polydict`.

greater_tuple_degrevlex (*f*, *g*)

Return the greater exponent tuple with respect to the total degree reversed lexicographical term order.

INPUT:

- `f` - exponent tuple
- `g` - exponent tuple

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='degrevlex')
sage: f = x + y; f.lm() # indirect doctest
x
sage: f = x + y^2*z; f.lm()
y^2*z
```

This method is called by the lm/lc/lt methods of `MPolynomial_polydict`.

greater_tuple_invlex (*f*, *g*)

Return the greater exponent tuple with respect to the inversed lexicographical term order.

INPUT:

- `f` - exponent tuple
- `g` - exponent tuple

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='invlex')
sage: f = x + y; f.lm() # indirect doctest
y
sage: f = y + x^2; f.lm()
y
```

This method is called by the lm/lc/lt methods of `MPolynomial_polydict`.

greater_tuple_lex (*f*, *g*)

Return the greater exponent tuple with respect to the lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='lex')
sage: f = x + y^2; f.lm() # indirect doctest
x
```

This method is called by the lm/lc/lt methods of `MPolynomial_polydict`.

greater_tuple_matrix (*f*, *g*)

Return the greater exponent tuple with respect to the matrix term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='m(1,3,1,0)')
sage: y > x^2 # indirect doctest
True
sage: y > x^3
False
```

greater_tuple_negdeglex (*f*, *g*)

Return the greater exponent tuple with respect to the negative degree lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
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 `MPolynomial_polydict`.

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

EXAMPLE:

```
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 `MPolynomial_polydict`.

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 `MPolynomial_polydict`.

INPUT:

- `f` - exponent tuple
- `g` - exponent tuple

EXAMPLE:

```
sage: P.<a,b,c,d,e,f>=PolynomialRing(QQbar, 6, order='degrevlex(3),degrevlex(3)')
sage: f = a + c^4; f.lm() # indirect doctest
c^4
sage: g = a + e^4; g.lm()
a
```

greater_tuple_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

EXAMPLE:

```
sage: t = TermOrder('negwdeglex',(1,2,3))
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order=t)
sage: f = x + y; f.lm() # indirect doctest
x
sage: f = x + x^2; f.lm()
x
sage: f = x^3 + z; f.lm()
x^3
```

This method is called by the lm/lc/lt methods of `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

EXAMPLE:

```
sage: t = TermOrder('negwdegrevlex',(1,2,3))
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order=t)
sage: f = x + y; f.lm() # indirect doctest
x
sage: f = x + x^2; f.lm()
x
sage: f = x^3 + z; f.lm()
x^3
```

This method is called by the 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

EXAMPLE:

```
sage: t = TermOrder('wdeglex',(1,2,3))
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order=t)
sage: f = x + y; f.lm() # indirect doctest
y
sage: f = x*y + z; f.lm()
x*y
```

This method is called by the 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:

```
sage: t = TermOrder('wdegrevlex',(1,2,3))
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order=t)
sage: f = x + y; f.lm() # indirect doctest
y
sage: f = x + y^2*z; f.lm()
y^2*z
```

This method is called by the lm/lc/lt methods of `MPolynomial_polydict`.

is_block_order ()

Return true if self is a block term order.

EXAMPLE:

```
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.

EXAMPLE:

```
sage: T = TermOrder('lex')
sage: T.is_global()
True
sage: T = TermOrder('degrevlex', 3) + TermOrder('degrevlex', 3)
sage: T.is_global()
True
sage: T = TermOrder('degrevlex', 3) + TermOrder('negdegrevlex', 3)
sage: T.is_global()
False
sage: T = TermOrder('degneglex', 3)
sage: T.is_global()
True
```

is_local()

Return true if this term order is definitely local. Return false otherwise, which includes unknown term orders.

EXAMPLE:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
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_() # optional - macaulay2
 ZZ
---[x0, x1, x2, x3, x4, x5, x6, x7, Degrees => {8:1}, Heft => {1}, MonomialOrder => {Monomia
127                                                                                  {GRevLex
                                                                                     {Lex =>
                                                                                     {Positio
```

magma_str()

Return a MAGMA representation of self.

Used to convert polynomial rings to their MAGMA representation.

EXAMPLE:

```
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

sage: T = P.term_order()
sage: T.magma_str()
'"grevlex"'
```

matrix()

Return the matrix defining matrix term order.

EXAMPLE:

```
sage: t = TermOrder("M(1,2,0,1)")
sage: t.matrix()
[1 2]
[0 1]
```

name()

EXAMPLE:

```
sage: TermOrder('lex').name()
'lex'
```

singular_moreblocks()

Return a the number of additional blocks SINGULAR needs to allocate for handling non-native orderings like $degneglex$.

EXAMPLE:

```
sage: P = PolynomialRing(GF(127),10,names='x',order='lex(3),deglex(5),lex(2)')
sage: T = P.term_order()
sage: T.singular_moreblocks()
0
sage: P = PolynomialRing(GF(127),10,names='x',order='lex(3),degneglex(5),lex(2)')
sage: T = P.term_order()
sage: T.singular_moreblocks()
1
sage: P = PolynomialRing(GF(127),10,names='x',order='degneglex(5),degneglex(5)')
sage: T = P.term_order()
sage: T.singular_moreblocks()
2
```

TEST:

The 'degneglex' ordering is somehow special: SINGULAR handles it using an extra weight vector block.

> sage: T = TermOrder("degneglex", 2) sage: P = PolynomialRing(QQ,2, names='x', order=T) sage: T = P.term_order() sage: T.singular_moreblocks() 1 sage: T = TermOrder("degneglex", 2) + TermOrder("degneglex", 2) sage: P = PolynomialRing(QQ,4, names='x', order=T) sage: T = P.term_order() sage: T.singular_moreblocks() 2

singular_str()

Return a SINGULAR representation of self.

Used to convert polynomial rings to their SINGULAR representation.

EXAMPLE:

```
sage: P = PolynomialRing(GF(127),10,names='x',order='lex(3),deglex(5),lex(2)')
sage: T = P.term_order()
sage: T.singular_str()
'(lp(3),Dp(5),lp(2))'
sage: P._singular_()
//   characteristic : 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
```

TEST:

The 'degneglex' ordering is somehow special, it looks like a block ordering in SINGULAR.

sage: T = TermOrder("degneglex", 2) sage: P = PolynomialRing(QQ,2, names='x', order=T) sage: T = P.term_order() sage: T.singular_str() '(a(1:2),ls(2))'

sage: T = TermOrder("degneglex", 2) + TermOrder("degneglex", 2) sage: P = PolynomialRing(QQ,4, names='x', order=T) sage: T = P.term_order() sage: T.singular_str() '(a(1:2),ls(2),a(1:2),ls(2))' sage: P._singular_() // characteristic : 0 // 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

tuple_weight (*f*)

Return the weight of tuple f.

INPUT:

- f - exponent tuple

EXAMPLE:

```
sage: t=TermOrder('wdeglex',(1,2,3))
sage: P.<a,b,c>=PolynomialRing(QQbar, order=t)
sage: P.term_order().tuple_weight([3,2,1])
10
```

weights ()

Return the weights for weighted term orders.

EXAMPLE:

```
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.

NOTE:

A term order in Singular also involves information on orders for modules. This is not taken into account in Sage.

EXAMPLE:

```
sage: singular.eval('ring r1 = (9,x),(a,b,c,d,e,f),(M((1,2,3,0)),wp(2,3),lp)')
''
sage: from sage.rings.polynomial.term_order import termorder_from_singular
sage: termorder_from_singular(singular)
Block term order with blocks:
(Matrix term order with matrix
[1 2]
[3 0],
 Weighted degree reverse lexicographic term order with weights (2, 3),
 Lexicographic term order of length 2)
```

AUTHOR:

- Simon King (2011-06-06)

3.2 Base class for multivariate polynomial rings

class `sage.rings.polynomial.multi_polynomial_ring_generic.`**`MPolynomialRing_generic`**

Bases: `sage.rings.ring.CommutativeRing`

Create a polynomial ring in several variables over a commutative ring.

EXAMPLES:

```
sage: R.<x,y> = ZZ['x,y']; R
Multivariate Polynomial Ring in x, y over Integer Ring
sage: class CR(CommutativeRing):
...       def __init__(self):
...           CommutativeRing.__init__(self,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 <class '....CR_with_category'>
```

TESTS:

Check that containment works correctly (ticket #10355):

```
sage: A1.<a> = PolynomialRing(QQ)
sage: A2.<a,b> = PolynomialRing(QQ)
sage: 3 in A2
True
sage: A1(a) in A2
True
```

`change_ring`(*base_ring=None*, *names=None*, *order=None*)

Return a new multivariate polynomial ring which isomorphic to self, but has a different ordering given by the parameter 'order' or names given by the parameter 'names'.

INPUT:

- `base_ring` – a base ring
- `names` – variable names

- `order` – a term order

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(GF(127),3,order='lex')
sage: x > y^2
True
sage: Q.<x,y,z> = P.change_ring(order='degrevlex')
sage: x > y^2
False
```

characteristic()

Return the characteristic of this polynomial ring.

EXAMPLES:

```
sage: R = PolynomialRing(QQ, 'x', 3)
sage: R.characteristic()
0
sage: R = PolynomialRing(GF(7),'x', 20)
sage: R.characteristic()
7
```

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

Return the completion of self with respect to the ideal generated by the variable(s) `p`.

INPUT:

- `p` – variable or tuple of variables
- `prec` – default precision of resulting power series ring
- `extras` – ignored; present for backward compatibility

EXAMPLES:

```
sage: P.<x,y,z,w> = PolynomialRing(ZZ)
sage: P.completion((w,x,y))
Multivariate Power Series Ring in w, x, y over Univariate Polynomial Ring in z over Integer
sage: P.completion((w,x,y,z))
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
```

construction()

Returns a functor F and base ring R such that F(R) == self.

EXAMPLES:

```
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
```

gen(*n=0*)

irrelevant_ideal()

Return the irrelevant ideal of this multivariate polynomial ring, which is the ideal generated by all of the indeterminate generators of this ring.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: R.irrelevant_ideal()
Ideal (x, y, z) of Multivariate Polynomial Ring in x, y, z over Rational Field
```

is_field(*proof=True*)

Return True if this multivariate polynomial ring is a field, i.e., it is a ring in 0 generators over a field.

is_finite()

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
```

krull_dimension()

macaulay_resultant(**args, **kwds*)

This is an implementation of the Macaulay Resultant. It computes the resultant of universal polynomials as well as polynomials with constant coefficients. This is a project done in sage days 55. It's based on the implementation in Maple by Manfred Minimair, which in turn is based on the references listed below: It calculates the Macaulay resultant for a list of polynomials, up to sign!

REFERENCES:

AUTHORS:

- Hao Chen, Solomon Vishkautsan (7-2014)

INPUT:

- **args – a list of n homogeneous polynomials in n variables.** works when `args[0]` is the list of polynomials, or `args` is itself the list of polynomials

kwds:

- **sparse – boolean (optional - default: False)** if `True` function creates sparse matrices.

OUTPUT:

- the macaulay resultant, an element of the base ring of `self`

Todo

Working with sparse matrices should usually give faster results, but with the current implementation it actually works slower. There should be a way to improve performance with regards to this.

EXAMPLES:

The number of polynomials has to match the number of variables:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: R.macaulay_resultant([y,x+z])
Traceback (most recent call last):
...
TypeError: number of polynomials(= 2) must equal number of variables (= 3)
```

The polynomials need to be all homogeneous:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: R.macaulay_resultant([y, x+z, z+x^3])
Traceback (most recent call last):
...
TypeError: resultant for non-homogeneous polynomials is not supported
```

All polynomials must be in the same ring:

```
sage: S.<x,y> = PolynomialRing(QQ, 2)
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: S.macaulay_resultant([y, z+x])
Traceback (most recent call last):
...
TypeError: not all inputs are polynomials in the calling ring
```

The following example recreates Proposition 2.10 in Ch.3 in [CLO]:

```
sage: K.<x,y> = PolynomialRing(ZZ, 2)
sage: flist,R = K._macaulay_resultant_universal_polynomials([1,1,2])
sage: R.macaulay_resultant(flist)
u2^2*u4^2*u6 - 2*u1*u2*u4*u5*u6 + u1^2*u5^2*u6 - u2^2*u3*u4*u7 + u1*u2*u3*u5*u7 + u0*u2*u4*u
```

The following example degenerates into the determinant of a $3 * 3$ matrix:

```
sage: K.<x,y> = PolynomialRing(ZZ, 2)
sage: flist,R = K._macaulay_resultant_universal_polynomials([1,1,1])
sage: R.macaulay_resultant(flist)
-u2*u4*u6 + u1*u5*u6 + u2*u3*u7 - u0*u5*u7 - u1*u3*u8 + u0*u4*u8
```

The following example is by Patrick Ingram(arxiv:1310.4114):

```
sage: U = PolynomialRing(ZZ,'y',2); y0,y1 = U.gens()
sage: R = PolynomialRing(U,'x',3); x0,x1,x2 = R.gens()
sage: f0 = y0*x2^2 - x0^2 + 2*x1*x2
sage: f1 = y1*x2^2 - x1^2 + 2*x0*x2
sage: f2 = x0*x1 - x2^2
sage: flist = [f0,f1,f2]
sage: R.macaulay_resultant([f0,f1,f2])
y0^2*y1^2 - 4*y0^3 - 4*y1^3 + 18*y0*y1 - 27
```

a simple example with constant rational coefficients:

```
sage: R.<x,y,z,w> = PolynomialRing(QQ,4)
sage: R.macaulay_resultant([w,z,y,x])
1
```

an example where the resultant vanishes:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: R.macaulay_resultant([x+y,y^2,x])
0
```

an example of bad reduction at a prime $p = 5$:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: R.macaulay_resultant([y,x^3+25*y^2*x,5*z])
125
```

The input can given as an unpacked list of polynomials:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: R.macaulay_resultant(y,x^3+25*y^2*x,5*z)
125
```

an example when the coefficients live in a finite field:

```
sage: F = FiniteField(11)
sage: R.<x,y,z,w> = PolynomialRing(F,4)
sage: R.macaulay_resultant([z,x^3,5*y,w])
4
```

example when the denominator in the algorithm vanishes(in this case the resultant is the constant term of the quotient of char polynomials of numerator/denominator):

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: R.macaulay_resultant([y, x+z, z^2])
-1
```

when there are only 2 polynomials, macaulay resultant degenerates to the traditional resultant:

```
sage: R.<x> = PolynomialRing(QQ,1)
sage: f =  x^2+1; g = x^5+1
sage: fh = f.homogenize()
sage: gh = g.homogenize()
sage: RH = fh.parent()
sage: f.resultant(g) == RH.macaulay_resultant([fh,gh])
True
```

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:

```
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: P.random_element(2, 5)
-6/5*x^2 + 2/3*z^2 - 1

sage: P.random_element(2, 5, choose_degree=True)
-1/4*x*y - x - 1/14*z - 1
```

Stacked rings:

```
sage: R = QQ['x,y']
sage: S = R['t,u']
sage: S.random_element(degree=2, terms=1)
-1/2*x^2 - 1/4*x*y - 3*y^2 + 4*y
sage: S.random_element(degree=2, terms=1)
(-x^2 - 2*y^2 - 1/3*x + 2*y + 9)*u^2
```

Default values apply if no degree and/or number of terms is provided:

```
sage: random_matrix(QQ['x,y,z'], 2, 2)
[357*x^2 + 1/4*y^2 + 2*y*z + 2*z^2 + 28*x      2*x*y + 3/2*y^2 + 2*y*z - 2*z^2 - z]
[                       x*y - y*z + 2*z^2         -x^2 - 4/3*x*z + 2*z^2 - x + 4*y]

sage: random_matrix(QQ['x,y,z'], 2, 2, terms=1, degree=2)
[ 1/2*y -1/4*x]
[   1/2  1/3*x]

sage: P.random_element(0, 1)
1

sage: P.random_element(2, 0)
0

sage: R.<x> = PolynomialRing(Integers(3), 1)
sage: R.random_element()
-x^2 + x
```

To produce a dense polynomial, pick `terms=Infinity`:

```
sage: P.<x,y,z> = GF(127)[]
sage: P.random_element(degree=2, terms=Infinity)
-55*x^2 - 51*x*y + 5*y^2 + 55*x*z - 59*y*z + 20*z^2 + 19*x - 55*y - 28*z + 17
sage: P.random_element(degree=3, terms=Infinity)
-54*x^3 + 15*x^2*y - x*y^2 - 15*y^3 + 61*x^2*z - 12*x*y*z + 20*y^2*z - 61*x*z^2 - 5*y*z^2 +
sage: P.random_element(degree=3, terms=Infinity, choose_degree=True)
57*x^3 - 58*x^2*y + 21*x*y^2 + 36*y^3 + 7*x^2*z - 57*x*y*z + 8*y^2*z - 11*x*z^2 + 7*y*z^2 +
```

The number of terms is silently reduced to the maximum available if more terms are requested:

```
sage: P.<x,y,z> = GF(127)[]
sage: P.random_element(degree=2, terms=1000)
```

```
5*x^2 - 10*x*y + 10*y^2 - 44*x*z + 31*y*z + 19*z^2 - 42*x - 50*y - 49*z - 60
```

remove_var(*order=None*, **var*)

Remove a variable or sequence of variables from self.

If `order` is not specified, then the subring inherits the term order of the original ring, if possible.

EXAMPLES:

```
sage: P.<x,y,z,w> = PolynomialRing(ZZ)
sage: P.remove_var(z)
Multivariate Polynomial Ring in x, y, w over Integer Ring
sage: P.remove_var(z,x)
Multivariate Polynomial Ring in y, w over Integer Ring
sage: P.remove_var(y,z,x)
Univariate Polynomial Ring in w over Integer Ring
```

Removing all variables results in the base ring:

```
sage: P.remove_var(y,z,x,w)
Integer Ring
```

If possible, the term order is kept:

```
sage: R.<x,y,z,w> = PolynomialRing(ZZ, order='deglex')
sage: R.remove_var(y).term_order()
Degree lexicographic term order

sage: R.<x,y,z,w> = PolynomialRing(ZZ, order='lex')
sage: R.remove_var(y).term_order()
Lexicographic term order
```

Be careful with block orders when removing variables:

```
sage: R.<x,y,z,u,v> = PolynomialRing(ZZ, order='deglex(2),lex(3)')
sage: R.remove_var(x,y,z)
Traceback (most recent call last):
...
ValueError: impossible to use the original term order (most likely because it was a block or
sage: R.remove_var(x,y,z, order='degrevlex')
Multivariate Polynomial Ring in u, v over Integer Ring
```

repr_long()

Return structured string representation of self.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ,order=TermOrder('degrevlex',1)+TermOrder('lex',2))
sage: print P.repr_long()
Polynomial Ring
 Base Ring : Rational Field
      Size : 3 Variables
  Block  0 : Ordering : degrevlex
             Names    : x
  Block  1 : Ordering : lex
             Names    : y, z
```

term_order()

univariate_ring(*x*)

Return a univariate polynomial ring whose base ring comprises all but one variables of self.

INPUT:

- x – a variable of self.

EXAMPLE:

```
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 Fiel
```

variable_names_recursive (*depth=None*)

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

EXAMPLES:

```
sage: R = QQ['x,y']['z,w']
sage: R.variable_names_recursive()
('x', 'y', 'z', 'w')
sage: R.variable_names_recursive(3)
('y', 'z', 'w')
```

weyl_algebra ()

Return the Weyl algebra generated from `self`.

EXAMPLES:

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

sage.rings.polynomial.multi_polynomial_ring_generic.**is_MPolynomialRing** (*x*)

sage.rings.polynomial.multi_polynomial_ring_generic.**unpickle_MPolynomialRing_generic** (*base_ri n, names, order*)

sage.rings.polynomial.multi_polynomial_ring_generic.**unpickle_MPolynomialRing_generic_v1** (*bas n, nan or- der*

3.3 Base class for elements of multivariate polynomial rings

class sage.rings.polynomial.multi_polynomial.**MPolynomial**

Bases: `sage.structure.element.CommutativeRingElement`

INPUT:

- `parent` - a SageObject

args ()

Returns the named of the arguments of self, in the order they are accepted from call.

EXAMPLES:

```
sage: R.<x,y> = ZZ[]
sage: x.args()
(x, y)
```

change_ring(*R*)

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

INPUT:

- R – a ring

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: f = x^3 + 3/5*y + 1
sage: f.change_ring(GF(7))
x^3 + 2*y + 1

sage: R.<x,y> = GF(9,'a')[]
sage: (x+2*y).change_ring(GF(3))
x - y
```

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.monomial`

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 $\mathbf{Z}$ – different implementation:

```
sage: R.<x,y,z> = PolynomialRing(ZZ,3,order='degrevlex')
sage: f=23*x^6*y^7 + x^3*y+6*x^7*z
sage: f.coefficients()
[23, 6, 1]
sage: R.<x,y,z> = PolynomialRing(ZZ,3,order='lex')
sage: f=23*x^6*y^7 + x^3*y+6*x^7*z
sage: f.coefficients()
[6, 23, 1]
```

AUTHOR:

- Didier Deshommes

content()

Returns the content of this polynomial. Here, we define content as the gcd of the coefficients in the base ring.

EXAMPLES:

```
sage: R.<x,y> = ZZ[]
sage: f = 4*x+6*y
```

```
sage: f.content()
2
sage: f.content().parent()
Integer Ring
```

TESTS:

Since trac ticket #10771, the gcd in QQ restricts to the gcd in ZZ:

```
sage: R.<x,y> = QQ[]
sage: f = 4*x+6*y
sage: f.content(); f.content().parent()
2
Rational Field
```

denominator()

Return a denominator of self.

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

Note that some subclases may implement its own denominator function.

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

EXAMPLES:

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

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

Next we compute the denominator of a polynomial over a number field.

```
sage: R.<x,y> = NumberField(symbolic_expression(x^2+3)  ,'a')['x,y']
sage: f = (1/17)*x^19 + (1/6)*y - (2/3)*x + 1/3; f
1/17*x^19 - 2/3*x + 1/6*y + 1/3
sage: f.denominator()
102
```

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

```
sage: R.<a,b,c> = RR[]
sage: f = a + b + RR('0.3'); f
a + b + 0.300000000000000
sage: f.denominator()
1.00000000000000
```

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

```
sage: R.<a,b,c> = GF(5)[]
sage: x = R(0)
sage: x.denominator()
1
sage: type(x.denominator())
<type 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
```

```
sage: type(a.denominator())
<type '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 derivative() function for more details.

See also:

`_derivative()`

EXAMPLES:

Polynomials implemented via Singular:

```
sage: R.<x, y> = PolynomialRing(FiniteField(5))
sage: f = x^3*y^5 + x^7*y
sage: type(f)
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>
sage: f.derivative(x)
2*x^6*y - 2*x^2*y^5
sage: f.derivative(y)
x^7
```

Generic multivariate polynomials:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: S.<x, y> = PolynomialRing(R)
sage: f = (t^2 + O(t^3))*x^2*y^3 + (37*t^4 + O(t^5))*x^3
sage: type(f)
<class 'sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict'>
sage: f.derivative(x)   # with respect to x
(2*t^2 + O(t^3))*x*y^3 + (111*t^4 + O(t^5))*x^2
sage: f.derivative(y)   # with respect to y
(3*t^2 + O(t^3))*x^2*y^2
sage: f.derivative(t)   # with respect to t (recurses into base ring)
(2*t + O(t^2))*x^2*y^3 + (148*t^3 + O(t^4))*x^3
sage: f.derivative(x, y) # with respect to x and then y
(6*t^2 + O(t^3))*x*y^2
sage: f.derivative(y, 3) # with respect to y three times
(6*t^2 + O(t^3))*x^2
sage: f.derivative()    # can't figure out the variable
Traceback (most recent call last):
...
ValueError: must specify which variable to differentiate with respect to
```

Polynomials over the symbolic ring (just for fun....):

```
sage: x = var("x")
sage: S.<u, v> = PolynomialRing(SR)
sage: f = u*v*x
sage: f.derivative(x) == u*v
True
```

```
sage: f.derivative(u) == v*x
True
```

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]
```

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:

```
sage: R.<x,y> = QQ[]
sage: f = x^2 + y + 1 + 5*x*y^10
sage: f.homogenize()
5*x*y^10 + x^2*h^9 + y*h^10 + h^11
```

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

```
sage: g = f.homogenize('z'); g
5*x*y^10 + x^2*z^9 + y*z^10 + z^11
sage: g.parent()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

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

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

If you want the ring of the result to be independent of whether the polynomial is homogenized, you can use `var` to use an existing variable to homogenize:

```
sage: R.<x,y,z> = QQ[]
sage: f = x^2 + y^2
sage: g = f.homogenize(z); g
x^2 + y^2
sage: g.parent()
Multivariate Polynomial Ring in x, y, z over Rational Field
sage: f = x^2 - y
```

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

The parameter `var` can also be given as a zero-based index in the list of variables:

```
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:

```
sage: g(x,y,1)
x^2 - y
```

If it is present already, this might not be the case:

```
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:

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

TESTS:

```
sage: R = PolynomialRing(QQ, 'x', 5)
sage: p = R.random_element()
sage: q1 = p.homogenize()
sage: q2 = p.homogenize()
sage: q1.parent() is q2.parent()
True
```

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: R.<x1,x2> = QQ[]
sage: I = R.ideal(x2**2 + x1 - 2, x1**2 - 1)
sage: f = x1 + 3*x2^2; g = f.inverse_mod(I); g
1/16*x1 + 3/16
sage: (f*g).reduce(I)
1
```

Test a non-invertible element:

```
sage: R.<x1,x2> = QQ[]
sage: I = R.ideal(x2**2 + x1 - 2, x1**2 - 1)
sage: f = x1 + x2
sage: f.inverse_mod(I)
Traceback (most recent call last):
...
ArithmeticError: element is non-invertible
```

is_generator()

Returns `True` if this polynomial is a generator of its parent.

EXAMPLES:

```
sage: R.<x,y>=ZZ[]
sage: x.is_generator()
True
sage: (x+y-y).is_generator()
True
sage: (x*y).is_generator()
False
sage: R.<x,y>=QQ[]
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.

TESTS:

```
sage: from sage.rings.polynomial.multi_polynomial import MPolynomial
sage: P.<x, y> = PolynomialRing(QQ, 2)
sage: MPolynomial.is_homogeneous(x+y)
True
sage: MPolynomial.is_homogeneous(P(0))
True
sage: MPolynomial.is_homogeneous(x+y^2)
False
sage: MPolynomial.is_homogeneous(x^2 + y^2)
True
sage: MPolynomial.is_homogeneous(x^2 + y^2*x)
False
sage: MPolynomial.is_homogeneous(x^2*y + y^2*x)
True
```

Note: This is a generic implementation which is likely overridden by subclasses.

jacobian_ideal()

Return the Jacobian ideal of the polynomial self.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: f = x^3 + y^3 + z^3
sage: f.jacobian_ideal()
Ideal (3*x^2, 3*y^2, 3*z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field
```

lift(*I*)

given an ideal `I = (f_1,...,f_r)` and some `g (== self)` in `I`, find `s_1,...,s_r` such that `g = s_1 f_1 + ... + s_r f_r`.

EXAMPLE:

```
sage: A.<x,y> = PolynomialRing(CC,2,order='degrevlex')
sage: I = A.ideal([x^10 + x^9*y^2, y^8 - x^2*y^7 ])
sage: f = x*y^13 + y^12
sage: M = f.lift(I)
sage: M
[y^7, x^7*y^2 + x^8 + x^5*y^3 + x^6*y + x^3*y^4 + x^4*y^2 + x*y^5 + x^2*y^3 + y^4]
sage: sum( map( mul , zip( M, I.gens() ) ) ) == f
True
```

macaulay_resultant(**args*)

This is an implementation of the Macaulay Resultant. It computes the resultant of universal polynomials as well as polynomials with constant coefficients. This is a project done in sage days 55. It's based on the implementation in Maple by Manfred Minimair, which in turn is based on the references [CLO], [Can], [Mac]. It calculates the Macaulay resultant for a list of Polynomials, up to sign!

AUTHORS:

- Hao Chen, Solomon Vishkautsan (7-2014)

INPUT:

- **args – a list of $n-1$ homogeneous polynomials in n variables.** works when `args[0]` is the list of polynomials, or `args` is itself the list of polynomials

OUTPUT:

- the macaulay resultant

EXAMPLES:

The number of polynomials has to match the number of variables:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: y.macaulay_resultant(x+z)
Traceback (most recent call last):
...
TypeError: number of polynomials(= 2) must equal number of variables (= 3)
```

The polynomials need to be all homogeneous:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: y.macaulay_resultant([x+z, z+x^3])
Traceback (most recent call last):
...
TypeError: resultant for non-homogeneous polynomials is not supported
```

All polynomials must be in the same ring:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: S.<x,y> = PolynomialRing(QQ, 2)
sage: y.macaulay_resultant(z+x,z)
Traceback (most recent call last):
...
TypeError: not all inputs are polynomials in the calling ring
```

The following example recreates Proposition 2.10 in Ch.3 of Using Algebraic Geometry:

```
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:])
u2^2*u4^2*u6 - 2*u1*u2*u4*u5*u6 + u1^2*u5^2*u6 - u2^2*u3*u4*u7 + u1*u2*u3*u5*u7 + u0*u2*u4*u
```

The following example degenerates into the determinant of a $3 * 3$ 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:])
-u2*u4*u6 + u1*u5*u6 + u2*u3*u7 - u0*u5*u7 - u1*u3*u8 + u0*u4*u8
```

The following example is by Patrick Ingram(arxiv:1310.4114):

```
sage: U = PolynomialRing(ZZ,'y',2); y0,y1 = U.gens()
sage: R = PolynomialRing(U,'x',3); x0,x1,x2 = R.gens()
sage: f0 = y0*x2^2 - x0^2 + 2*x1*x2
sage: f1 = y1*x2^2 - x1^2 + 2*x0*x2
sage: f2 = x0*x1 - x2^2
sage: f0.macaulay_resultant(f1,f2)
y0^2*y1^2 - 4*y0^3 - 4*y1^3 + 18*y0*y1 - 27
```

a simple example with constant rational coefficients:

```
sage: R.<x,y,z,w> = PolynomialRing(QQ,4)
sage: w.macaulay_resultant([z,y,x])
1
```

an example where the resultant vanishes:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: (x+y).macaulay_resultant([y^2,x])
0
```

an example of bad reduction at a prime `p = 5`:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: y.macaulay_resultant([x^3+25*y^2*x,5*z])
125
```

The input can given as an unpacked list of polynomials:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: y.macaulay_resultant(x^3+25*y^2*x,5*z)
125
```

an example when the coefficients live in a finite field:

```
sage: F = FiniteField(11)
sage: R.<x,y,z,w> = PolynomialRing(F,4)
sage: z.macaulay_resultant([x^3,5*y,w])
4
```

example when the denominator in the algorithm vanishes(in this case the resultant is the constant term of the quotient of char polynomials of numerator/denominator):

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: y.macaulay_resultant([x+z, z^2])
-1
```

when there are only 2 polynomials, macaulay resultant degenerates to the traditional resultant:

```
sage: R.<x> = PolynomialRing(QQ,1)
sage: f =  x^2+1; g = x^5+1
sage: fh = f.homogenize()
sage: gh = g.homogenize()
sage: RH = fh.parent()
sage: f.resultant(g) == fh.macaulay_resultant(gh)
True
```

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

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

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

INPUT:

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

EXAMPLES:

```
sage: k.<a> = GF(9); R.<x,y> = k[];  f = x*a + 2*x^3*y*a + a
sage: f.map_coefficients(lambda a : a + 1)
(-a + 1)*x^3*y + (a + 1)*x + (a + 1)
```

Examples with different base ring:

```
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
(-s^3 - s^2 + 1)*X + Y
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 in r of size 3^2
sage: g = f.map_coefficients(h, new_base_ring=GF(3)); 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:

```
sage: R.<x,y> = QQ[]
sage: f = 1 + x*y + x^3 + y^3
sage: P = f.newton_polytope()
sage: P
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices
```

```
sage: P.is_simple()
True
```

TESTS:

```
sage: R.<x,y> = QQ[]
sage: R(0).newton_polytope()
The empty polyhedron in ZZ^0
sage: R(1).newton_polytope()
A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex
sage: R(x^2+y^2).newton_polytope().integral_points()
((0, 2), (1, 1), (2, 0))
```

numerator()

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

Note that some subclases may implement its own numerator function.

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

EXAMPLES:

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

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

Next we compute the numerator of a polynomial over a number field.

```
sage: R.<x,y> = NumberField(symbolic_expression(x^2+3)  ,'a')['x,y']
sage: f = (1/17)*y^19 - (2/3)*x + 1/3; f
1/17*y^19 - 2/3*x + 1/3
sage: f.numerator()
3*y^19 - 34*x + 17
sage: f == f.numerator()
False
```

We try to compute the numerator of a polynomial with coefficients in the finite field of 3 elements.

```
sage: K.<x,y,z> = GF(3)['x, y, z']
sage: f = 2*x*z + 2*z^2 + 2*y + 1; f
-x*z - z^2 - y + 1
sage: f.numerator()
-x*z - z^2 - y + 1
```

We check that the computation the numerator and denominator are valid

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

polynomial(*var*)

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

EXAMPLES:

```
sage: R.<x,w,z> = QQ[]
sage: f = x^3 + 3*w*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 Fiel

sage: f.polynomial(w)
w^5 + 17*x*w^3 + 3*x*w + z^5 + x^3
sage: f.polynomial(z)
z^5 + w^5 + 17*x*w^3 + x^3 + 3*x*w
sage: R.<x,w,z,k> = ZZ[]
sage: f = x^3 + 3*w*x + w^5 + (17*w^3)*x + z^5 +x*w*z*k + 5
sage: f.polynomial(x)
x^3 + (17*w^3 + w*z*k + 3*w)*x + w^5 + z^5 + 5
sage: f.polynomial(w)
w^5 + 17*x*w^3 + (x*z*k + 3*x)*w + z^5 + x^3 + 5
sage: f.polynomial(z)
z^5 + x*w*k*z + w^5 + 17*x*w^3 + x^3 + 3*x*w + 5
sage: f.polynomial(k)
x*w*z*k + w^5 + z^5 + 17*x*w^3 + x^3 + 3*x*w + 5
sage: R.<x,y>=GF(5)[]
sage: f=x^2+x+y
sage: f.polynomial(x)
x^2 + x + y
sage: f.polynomial(y)
y + x^2 + x
```

sylvester_matrix(*right*, *variable=None*)

Given two nonzero polynomials self and right, returns the Sylvester matrix of the polynomials with respect to a given variable.

Note that the Sylvester matrix is not defined if one of the polynomials is zero.

INPUT:

- self , right: multivariate polynomials
- variable: optional, compute the Sylvester matrix with respect to this variable. If variable is not provided, the first variable of the polynomial ring is used.

OUTPUT:

- The Sylvester matrix of self and right.

EXAMPLES:

```
sage: R.<x, y> = PolynomialRing(ZZ)
sage: f = (y + 1)*x + 3*x**2
sage: g = (y + 2)*x + 4*x**2
sage: M = f.sylvester_matrix(g, x)
sage: print 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:

```
sage: M.determinant()
0

sage: f.sylvester_matrix(1 + g, x).determinant()
y^2 - y + 7
```

If both polynomials are of positive degree with respect to variable, the determinant of the Sylvester matrix is the resultant:

```
sage: f = R.random_element(4)
sage: g = R.random_element(4)
sage: f.sylvester_matrix(g, x).determinant() == f.resultant(g, x)
True
```

TEST:

The variable is optional:

```
sage: f = x + y
sage: g = x + y
sage: f.sylvester_matrix(g)
[1 y]
[1 y]
```

Polynomials must be defined over compatible base rings:

```
sage: K.<x, y> = QQ[]
sage: f = x + y
sage: L.<x, y> = ZZ[]
sage: g = x + y
sage: R.<x, y> = GF(25, 'a')[]
sage: h = x + y
sage: f.sylvester_matrix(g, 'x')
[1 y]
[1 y]
sage: g.sylvester_matrix(h, 'x')
[1 y]
[1 y]
sage: f.sylvester_matrix(h, 'x')
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents: 'Multivariate Polynomial Rin
sage: K.<x, y, z> = QQ[]
sage: f = x + y
sage: L.<x, z> = QQ[]
sage: g = x + z
sage: f.sylvester_matrix(g)
[1 y]
[1 z]
```

Corner cases:

```
sage: K.<x ,y>=QQ[]
sage: f = x^2+1
sage: g = K(0)
sage: f.sylvester_matrix(g)
Traceback (most recent call last):
...
```

```
ValueError: The Sylvester matrix is not defined for zero polynomials
sage: g.sylvester_matrix(f)
Traceback (most recent call last):
...
ValueError: The Sylvester matrix is not defined for zero polynomials
sage: g.sylvester_matrix(g)
Traceback (most recent call last):
...
ValueError: The Sylvester matrix is not defined for zero polynomials
sage: K(3).sylvester_matrix(x^2)
[3 0]
[0 3]
sage: K(3).sylvester_matrix(K(4))
[]
```

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)
680564733841876926945195958937245974528
sage: q = R.random_element(100, 20) #random
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
sage: p.weighted_degree(2/1,1,1)
6
```

The `weighted_degree` coincides with the `degree` of a weighted polynomial ring, but the later is faster.

```
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
```

TESTS:

```
sage: R = PolynomialRing(QQ, 'a', 5)
sage: f = R.random_element(terms=20)
sage: w = random_vector(ZZ,5)
sage: d1 = f.weighted_degree(w)
sage: d2 = (f*1.0).weighted_degree(w)
sage: d1 == d2
True
```

`sage.rings.polynomial.multi_polynomial.`**`is_MPolynomial`**`(x)`

3.4 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
- Robert Bradshaw (2007-08-15): Coercions from rings in a subset of the variables.

EXAMPLES:

We construct the Frobenius morphism on $\mathbf{F}_5[x, y, z]$ over $\mathbf{F}_5$:

```
sage: R, (x,y,z) = PolynomialRing(GF(5), 3, 'xyz').objgens()
sage: frob = R.hom([x^5, y^5, z^5])
sage: frob(x^2 + 2*y - z^4)
-z^20 + x^10 + 2*y^5
sage: frob((x + 2*y)^3)
x^15 + x^10*y^5 + 2*x^5*y^10 - 2*y^15
sage: (x^5 + 2*y^5)^3
x^15 + x^10*y^5 + 2*x^5*y^10 - 2*y^15
```

We make a polynomial ring in one variable over a polynomial ring in two variables:

```
sage: R.<x, y> = PolynomialRing(QQ, 2)
sage: S.<t> = PowerSeriesRing(R)
sage: t*(x+y)
(x + y)*t
```

class sage.rings.polynomial.multi_polynomial_ring.**MPolynomialRing_macaulay2_repr**

is_exact()
x.__init__(...) initializes x; see help(type(x)) for signature

class sage.rings.polynomial.multi_polynomial_ring.**MPolynomialRing_polydict**(*base_ring*, *n*, *names*, *order*)

Bases: sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_macaulay2_repr, sage.rings.polynomial.polynomial_singular_interface.PolynomialRing_singular_repr, sage.rings.polynomial.multi_polynomial_ring_generic.MPolynomialRing_generic

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
```

class sage.rings.polynomial.multi_polynomial_ring.**MPolynomialRing_polydict_domain**(*base_ring*, *n*, *names*, *order*)

Bases: sage.rings.ring.IntegralDomain, sage.rings.polynomial.multi_polynomial_ring.MPolyno

ideal(**gens*, ***kwds*)
Create an ideal in this polynomial ring.

is_field(*proof=True*)
x.__init__(...) initializes x; see help(type(x)) for signature

is_integral_domain(*proof=True*)
x.__init__(...) initializes x; see help(type(x)) for signature

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

EXAMPLE:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
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*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.

INPUT:

- a - monomial
- b - monomial

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
```

TESTS:

```
sage: P.<x,y,z>=PolynomialRing(ZZ,3, order='degrevlex')
sage: P.monomial_divides(P(1), P(0))
True
sage: P.monomial_divides(P(1), x)
True
```

monomial_lcm (*f*, *g*)

LCM for monomials. Coefficients are ignored.

INPUT:

- f - monomial
- g - monomial

EXAMPLE:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_lcm(3/2*x*y,x)
x*y
```

TESTS:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: R.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_lcm(x*y,R.gen())
x*y

sage: P.monomial_lcm(P(3/2),P(2/3))
1

sage: P.monomial_lcm(x,P(1))
x
```

monomial_pairwise_prime (*h*, *g*)

Return True if h and g are pairwise prime. Both are treated as monomials.

INPUT:

- h - monomial
- g - monomial

EXAMPLES:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_pairwise_prime(x^2*z^3, y^4)
True

sage: P.monomial_pairwise_prime(1/2*x^3*y^2, 3/4*y^3)
False
```

TESTS:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: Q.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_pairwise_prime(x^2*z^3, Q('y^4'))
True

sage: P.monomial_pairwise_prime(1/2*x^3*y^2, Q(0))
True

sage: P.monomial_pairwise_prime(P(1/2),x)
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)

EXAMPLE:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ, 3, order='degrevlex')
sage: P.monomial_quotient(3/2*x*y,x)
y

sage: P.monomial_quotient(3/2*x*y,2*x,coeff=True)
3/4*y
```

TESTS:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: R.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_quotient(x*y,x)
y

sage: P.monomial_quotient(x*y,R.gen())
y

sage: P.monomial_quotient(P(0),P(1))
0

sage: P.monomial_quotient(P(1),P(0))
Traceback (most recent call last):
...
ZeroDivisionError
```

```
sage: P.monomial_quotient(P(3/2),P(2/3), coeff=True)
9/4

sage: P.monomial_quotient(x,y) # Note the wrong result
x*y^-1

sage: P.monomial_quotient(x,P(1))
x
```

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 (g,flt) is returned, (0,0) otherwise, where flt is f/g.lm().

It is assumed that G is iterable and contains ONLY elements in self.

INPUT:

- `f` - monomial
- `G` - list/set of mpolynomials

EXAMPLES:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
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)
```

TESTS:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
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(P(0),G)
(0, 0)

sage: P.monomial_reduce(f,[P(0)])
(0, 0)
```

3.5 Generic Multivariate Polynomials

AUTHORS:

- David Joyner: first version
- William Stein: use dict's instead of lists
- Martin Albrecht malb@informatik.uni-bremen.de: some functions added
- William Stein (2006-02-11): added better __div__ behavior.
- Kiran S. Kedlaya (2006-02-12): added Macaulay2 analogues of some Singular features

- William Stein (2006-04-19): added e.g., `f[1,3]` to get coeff of xy^3; added examples of the new `R.x,y = PolynomialRing(QQ,2)` notation.
- Martin Albrecht: improved singular coercions (restructured class hierarchy) and added ETuples
- Robert Bradshaw (2007-08-14): added support for coercion of polynomials in a subset of variables (including multi-level univariate rings)
- Joel B. Mohler (2008-03): Refactored interactions with ETuples.

EXAMPLES:

We verify Lagrange's four squares identity:

```
sage: R.<a0,a1,a2,a3,b0,b1,b2,b3> = QQbar[]
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 +
True
```

class `sage.rings.polynomial.multi_polynomial_element.`**`MPolynomial_element`**(*parent*, *x*)

Bases: `sage.rings.polynomial.multi_polynomial.MPolynomial`

EXAMPLE:

```
sage: K.<cuberoot2> = NumberField(x^3 - 2)
sage: L.<cuberoot3> = K.extension(x^3 - 3)
sage: S.<sqrt2> = L.extension(x^2 - 2)
sage: S
Number Field in sqrt2 with defining polynomial x^2 - 2 over its base field
sage: P.<x,y,z> = PolynomialRing(S) # indirect doctest
```

`change_ring`(*R*)

x.__init__(...) initializes x; see help(type(x)) for signature

`element`()

x.__init__(...) initializes x; see help(type(x)) for signature

class `sage.rings.polynomial.multi_polynomial_element.`**`MPolynomial_polydict`**(*parent*, *x*)

Bases: `sage.rings.polynomial.polynomial_singular_interface.Polynomial_singular_repr`, `sage.rings.polynomial.multi_polynomial_element.MPolynomial_element`

Multivariate polynomials implemented in pure python using polydicts.

`coefficient`(*degrees*)

Return the coefficient of the variables with the degrees specified in the python dictionary `degrees`. Mathematically, this is the coefficient in the base ring adjoined by the variables of this ring not listed in `degrees`. However, the result has the same parent as this polynomial.

This function contrasts with the function `monomial_coefficient` which returns the coefficient in the base ring of a monomial.

INPUT:

- `degrees` - Can be any of:
  - a dictionary of degree restrictions
  - a list of degree restrictions (with None in the unrestricted variables)
  - a monomial (very fast, but not as flexible)

OUTPUT: element of the parent of self

See also:

For coefficients of specific monomials, look at `monomial_coefficient()`.

EXAMPLES:

```
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([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 expone
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

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:

```
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*)

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.

OUTPUT: integer

EXAMPLES:

```
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 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 trac ticket #17366.)

```
sage: x, y = ZZ['x','y'].gens()
sage: GF(3037000453)['x','y'].gen(0).degree(x)
1

sage: x0, y0 = QQ['x','y'].gens()
sage: GF(3037000453)['x','y'].gen(0).degree(x0)
Traceback (most recent call last):
...
TypeError: x must canonically coerce to parent

sage: GF(3037000453)['x','y'].gen(0).degree(x^2)
Traceback (most recent call last):
...
TypeError: x must be one of the generators of the parent
```

degrees()

Returns a tuple (precisely - an ETuple) with the degree of each variable in this polynomial. The list of degrees is, of course, ordered by the order of the generators.

EXAMPLES:

```
sage: R.<x,y,z>=PolynomialRing(QQbar)
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.degrees()
(2, 2, 0)
sage: f = x^2+z^2
sage: f.degrees()
(2, 0, 2)
sage: f.total_degree()  # this simply illustrates that total degree is not the sum of the de
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:

Return the list of exponents as a list of ETuples or tuples.

EXAMPLES:

```
sage: R.<a,b,c> = PolynomialRing(QQbar, 3)
sage: f = a^3 + b + 2*b^2
sage: f.exponents()
[(3, 0, 0), (0, 2, 0), (0, 1, 0)]
```

Be default the list of exponents is a list of ETuples:

```
sage: type(f.exponents()[0])
<type 'sage.rings.polynomial.polydict.ETuple'>
sage: type(f.exponents(as_ETuples=False)[0])
<type 'tuple'>
```

factor(*proof=True*)

Compute the irreducible factorization of this polynomial.

INPUT:

- `proof'' - insist on provably correct results (ignored, always ``True)`

ALGORITHM: Use univariate factorization code.

If a polynomial is univariate, the appropriate univariate factorization code is called:

```
sage: R.<z> = PolynomialRing(CC,1)
sage: f = z^4 - 6*z + 3
sage: f.factor()
(z - 1.60443920904349) * (z - 0.511399619393097) * (z + 1.05791941421830 - 1.59281852704435*
```

TESTS:

Check if we can handle polynomials with no variables, see trac ticket #7950:

```
sage: P = PolynomialRing(ZZ,0,'')
sage: res = P(10).factor(); res
2 * 5
sage: res[0][0].parent()
Multivariate Polynomial Ring in no variables over Integer Ring
sage: R = PolynomialRing(QQ,0,'')
sage: res = R(10).factor(); res
10
sage: res.unit().parent()
Rational Field
sage: P(0).factor()
Traceback (most recent call last):
...
ArithmeticError: Prime factorization of 0 not defined.
```

Check if we can factor a constant polynomial, see trac ticket #8207:

```
sage: R.<x,y> = CC[]
sage: R(1).factor()
1.00000000000000
```

Check that we prohibit too large moduli, trac ticket #11829:

```
sage: R.<x,y> = GF(previous_prime(2^31))[]
sage: factor(x+y+1,proof=False)
Traceback (most recent call last):
...
NotImplementedError: Factorization of multivariate polynomials over prime fields with charac
```

We check that the original issue in trac ticket #7554 is fixed:

```
sage: K.<a> = PolynomialRing(QQ)
sage: R.<x,y> = PolynomialRing(FractionField(K))
sage: factor(x)
x
```

integral(*var=None*)

Integrates `self` with respect to variable `var`.

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

If `var` is not one of the generators of this ring, integral(var) is called recursively on each coefficient of this polynomial.

EXAMPLES:

On polynomials with rational coefficients:

```
sage: x, y = PolynomialRing(QQ, 'x, y').gens()
sage: ex = x*y + x - y
sage: it = ex.integral(x); it
1/2*x^2*y + 1/2*x^2 - x*y
sage: it.parent() == x.parent()
True
```

On polynomials with coefficients in power series:

```
sage: R.<t> = PowerSeriesRing(QQbar)
sage: S.<x, y> = PolynomialRing(R)
sage: f = (t^2 + 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
(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
```

TESTS:

```
sage: f.integral()     # can't figure out the variable
Traceback (most recent call last):
...
```

```
ValueError: must specify which variable to integrate with respect to
```

inverse_of_unit()

x.__init__(...) initializes x; see help(type(x)) for signature

is_constant()

True if polynomial is constant, and False otherwise.

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.is_constant()
False
sage: g = 10*x^0
sage: g.is_constant()
True
```

is_generator()

Returns True if self is a generator of it's parent.

EXAMPLES:

```
sage: R.<x,y>=QQbar[]
sage: x.is_generator()
True
sage: (x+y-y).is_generator()
True
sage: (x*y).is_generator()
False
```

is_homogeneous()

Return True if self is a homogeneous polynomial.

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: (x+y).is_homogeneous()
True
sage: (x.parent()(0)).is_homogeneous()
True
sage: (x+y^2).is_homogeneous()
False
sage: (x^2 + y^2).is_homogeneous()
True
sage: (x^2 + y^2*x).is_homogeneous()
False
sage: (x^2*y + y^2*x).is_homogeneous()
True
```

is_monomial()

Returns True if self is a monomial, which we define to be a product of generators with coefficient 1.

Use is_term to allow the coefficient to not be 1.

EXAMPLES:

```
sage: R.<x,y>=QQbar[]
sage: x.is_monomial()
True
sage: (x+2*y).is_monomial()
```

```
False
sage: (2*x).is_monomial()
False
sage: (x*y).is_monomial()
True
```

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

```
sage: (2*x*y).is_term()
True
sage: (2*x*y).is_monomial()
False
```

is_term()

Returns 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 coefficent be 1.

EXAMPLES:

```
sage: R.<x,y>=QQbar[]
sage: x.is_term()
True
sage: (x+2*y).is_term()
False
sage: (2*x).is_term()
True
sage: (7*x^5*y).is_term()
True
```

To require leading coefficient 1, use is_monomial():

```
sage: (2*x*y).is_monomial()
False
sage: (2*x*y).is_term()
True
```

is_unit()

Return True if self is a unit.

EXAMPLES:

```
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
```

is_univariate()

Returns True if this multivariate polynomial is univariate and False otherwise.

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.is_univariate()
False
sage: g = f.subs({x:10}); g
700*y^2 + (-2)*y + 305
sage: g.is_univariate()
True
sage: f = x^0
sage: f.is_univariate()
True
```

lc()

Returns the leading coefficient of self i.e., self.coefficient(self.lm())

EXAMPLES:

```
sage: R.<x,y,z>=QQbar[]
sage: f=3*x^2-y^2-x*y
sage: f.lc()
3
```

lift(*I*)

given an ideal I = (f_1,...,f_r) and some g (== self) in I, find s_1,...,s_r such that g = s_1 f_1 + ... + s_r f_r

ALGORITHM: Use Singular.

EXAMPLE:

```
sage: A.<x,y> = PolynomialRing(CC,2,order='degrevlex')
sage: I = A.ideal([x^10 + x^9*y^2, y^8 - x^2*y^7 ])
sage: f = x*y^13 + y^12
sage: M = f.lift(I)
sage: M
[y^7, x^7*y^2 + x^8 + x^5*y^3 + x^6*y + x^3*y^4 + x^4*y^2 + x*y^5 + x^2*y^3 + y^4]
sage: sum( map( mul , zip( M, I.gens() ) ) ) == f
True
```

lm()

Returns the lead monomial of self with respect to the term order of self.parent().

EXAMPLES:

```
sage: R.<x,y,z>=PolynomialRing(GF(7),3,order='lex')
sage: (x^1*y^2 + y^3*z^4).lm()
x*y^2
sage: (x^3*y^2*z^4 + x^3*y^2*z^1).lm()
x^3*y^2*z^4

sage: R.<x,y,z>=PolynomialRing(CC,3,order='deglex')
sage: (x^1*y^2*z^3 + x^3*y^2*z^0).lm()
x*y^2*z^3
sage: (x^1*y^2*z^4 + x^1*y^1*z^5).lm()
x*y^2*z^4

sage: R.<x,y,z>=PolynomialRing(QQbar,3,order='degrevlex')
sage: (x^1*y^5*z^2 + x^4*y^1*z^3).lm()
x*y^5*z^2
sage: (x^4*y^7*z^1 + x^4*y^2*z^3).lm()
x^4*y^7*z
```

TESTS:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict
sage: R.<x,y>=MPolynomialRing_polydict(GF(2),2,order='lex')
sage: f=x+y
sage: f.lm()
x
```

lt()

Returns the leading term of self i.e., self.lc()*self.lm(). The notion of "leading term" depends on the ordering defined in the parent ring.

EXAMPLES:

```
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
```

TESTS:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict
sage: R.<x,y>=MPolynomialRing_polydict(GF(2),2,order='lex')
sage: f=x+y
sage: f.lt()
x
```

monomial_coefficient(*mon*)

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

This function contrasts with the function `coefficient` which returns the coefficient of a monomial viewing this polynomial in a polynomial ring over a base ring having fewer variables.

INPUT:

- `mon` - a monomial

OUTPUT: coefficient in base ring

See also:

For coefficients in a base ring of fewer variables, look at `coefficient()`.

EXAMPLES:

The parent of the return is a member of the base ring.

```
sage: R.<x,y>=QQbar[]
```

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()
Algebraic Field
```

```
sage: f = y^2 + y^2*x - x^9 - 7*x + 5*x*y
sage: f.monomial_coefficient(y^2)
1
sage: f.monomial_coefficient(x*y)
5
sage: f.monomial_coefficient(x^9)
-1
sage: f.monomial_coefficient(x^10)
0

sage: var('a')
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()

Returns the list of monomials in self. The returned list is decreasingly ordered by the term ordering of self.parent().

OUTPUT: list of MPolynomials representing Monomials

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.monomials()
[x^2*y^2, x^2, y, 1]

sage: R.<fx,fy,gx,gy> = QQbar[]
sage: F = ((fx*gy - fy*gx)^3)
sage: F
-fy^3*gx^3 + 3*fx*fy^2*gx^2*gy + (-3)*fx^2*fy*gx*gy^2 + fx^3*gy^3
sage: F.monomials()
[fy^3*gx^3, fx*fy^2*gx^2*gy, fx^2*fy*gx*gy^2, fx^3*gy^3]
sage: F.coefficients()
[-1, 3, -3, 1]
sage: sum(map(mul,zip(F.coefficients(),F.monomials()))) == F
True
```

nvariables()

Number of variables in this polynomial

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.nvariables ()
2
sage: g = f.subs({x:10}); g
700*y^2 + (-2)*y + 305
sage: g.nvariables ()
1
```

quo_rem(*right*)

Returns quotient and remainder of self and right.

EXAMPLE:

```
sage: R.<x,y> = CC[]
sage: f = y*x^2 + x + 1
sage: f.quo_rem(x)
(x*y + 1.00000000000000, 1.00000000000000)
```

ALGORITHM: Use Singular.

reduce (*I*)

Reduce this polynomial by the the polynomials in I.

INPUT:

- I - a list of polynomials or an ideal

EXAMPLE:

```
sage: P.<x,y,z> = QQbar[]
sage: f1 = -2 * x^2 + x^3
sage: f2 = -2 * y + x* y
sage: f3 = -x^2 + y^2
sage: F = Ideal([f1,f2,f3])
sage: g = x*y - 3*x*y^2
sage: g.reduce(F)
(-6)*y^2 + 2*y
sage: g.reduce(F.gens())
(-6)*y^2 + 2*y

sage: f = 3*x
sage: f.reduce([2*x,y])
0

sage: k.<w> = CyclotomicField(3)
sage: A.<y9,y12,y13,y15> = PolynomialRing(k)
sage: J = [ y9 + y12]
sage: f = y9 - y12; f.reduce(J)
-2*y12
sage: f = y13*y15; f.reduce(J)
y13*y15
sage: f = y13*y15 + y9 - y12; f.reduce(J)
y13*y15 - 2*y12
```

Make sure the remainder returns the correct type, fixing trac ticket #13903:

```
sage: R.<y1,y2>=PolynomialRing(Qp(5),2, order='lex')
sage: G=[y1^2 + y2^2, y1*y2 + y2^2, y2^3]
sage: type((y2^3).reduce(G))
<class 'sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict'>
```

resultant (*other*, *variable=None*)

Compute the resultant of `self` and `other` with respect to `variable`.

If a second argument is not provided, the first variable of `self.parent()` is chosen.

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)
-2*y^3
sage: a.resultant(b, y)
2*x^3
```

TESTS:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: P.<x,y> = MPolynomialRing_polydict_domain(QQ, 2, order='degrevlex')
sage: a = x + y
sage: b = x^3 - y^3
sage: a.resultant(b)
-2*y^3
sage: a.resultant(b, y)
2*x^3
```

Check that trac ticket #15061 is fixed:

```
sage: R.<x, y> = AA[]
sage: (x^2 + 1).resultant(x^2 - y)
y^2 + 2*y + 1
```

subs (*fixed=None*, ***kw*)

Fixes some given variables in a given multivariate polynomial and returns the changed multivariate polynomials. The polynomial itself is not affected. The variable,value pairs for fixing are to be provided as a dictionary of the form {variable:value}.

This is a special case of evaluating the polynomial with some of the variables constants and the others the original variables.

INPUT:

- `fixed` - (optional) dictionary of inputs
- `**kw` - named parameters

OUTPUT: new MPolynomial

EXAMPLES:

```
sage: 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 self, which is the maximum degree of any monomial in self.

EXAMPLES:

```
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
sage: f.total_degree()
7
```

```
sage: f=99*x^6*y^3*z^9
sage: f.total_degree()
18
sage: f=x*y^3*z^6+3*x^2
sage: f.total_degree()
10
sage: f=z^3+8*x^4*y^5*z
sage: f.total_degree()
10
sage: f=z^9+10*x^4+y^8*x^2
sage: f.total_degree()
10
```

univariate_polynomial(*R=None*)

Returns a univariate polynomial associated to this multivariate polynomial.

INPUT:

- `R` - (default: None) PolynomialRing

If this polynomial is not in at most one variable, then a ValueError exception is raised. This is checked using the is_univariate() method. The new Polynomial is over the same base ring as the given MPolynomial.

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.univariate_polynomial()
Traceback (most recent call last):
...
TypeError: polynomial must involve at most one variable
sage: g = f.subs({x:10}); g
700*y^2 + (-2)*y + 305
sage: g.univariate_polynomial ()
700*y^2 - 2*y + 305
sage: g.univariate_polynomial(PolynomialRing(QQ,'z'))
700*z^2 - 2*z + 305
```

TESTS:

```
sage: P = PolynomialRing(QQ, 0, '')
sage: P(5).univariate_polynomial()
5
```

variable(*i*)

Returns i-th variable occurring in this polynomial.

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.variable(0)
x
sage: f.variable(1)
y
```

variables()

Returns the tuple of variables occurring in this polynomial.

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.variables()
(x, y)
sage: g = f.subs({x:10}); g
700*y^2 + (-2)*y + 305
sage: g.variables()
(y,)
```

TESTS:

This shows that the issue at trac ticket #7077 is fixed:

```
sage: x,y,z=polygens(QQ,'x,y,z')
sage: (x^2).variables()
(x,)
```

`sage.rings.polynomial.multi_polynomial_element.`**`degree_lowest_rational_function`**(*r*, *x*)

INPUT:

- `r` - a multivariate rational function
- `x` - a multivariate polynomial ring generator x

OUTPUT:

- `integer` - the degree of r in x and its "leading" (in the x-adic sense) coefficient.

Note: This function is dependent on the ordering of a python dict. Thus, it isn't really mathematically well-defined. I think that it should made a method of the FractionFieldElement class and rewritten.

EXAMPLES:

```
sage: R1 = PolynomialRing(FiniteField(5), 3, names = ["a","b","c"])
sage: F = FractionField(R1)
sage: a,b,c = R1.gens()
sage: f = 3*a*b^2*c^3+4*a*b*c
sage: g = a^2*b*c^2+2*a^2*b^4*c^7
```

Consider the quotient $f/g = \frac{4+3bc^2}{ac+2ab^3c^6}$ (note the cancellation).

```
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, 3)
sage: degree_lowest_rational_function(r,b)
(0, 4)
sage: degree_lowest_rational_function(r,c)
(-1, 4)
```

`sage.rings.polynomial.multi_polynomial_element.`**`is_MPolynomial`**(*x*)

x.__init__(...) initializes x; see help(type(x)) for signature

3.6 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.

AUTHORS:

- William Stein
- Kiran S. Kedlaya (2006-02-12): added Macaulay2 analogues of some Singular features
- Martin Albrecht (2008,2007): refactoring, many Singular related functions
- Martin Albrecht (2009): added Groebner basis over rings functionality from Singular 3.1
- John Perry (2012): bug fixing equality & containment of ideals

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()
sage: I = ideal(x^5 + y^4 + z^3 - 1,  x^3 + y^3 + z^2 - 1)
sage: B = I.groebner_basis()
sage: 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
sage: (2*x*f + g).reduce(B)
0

sage: (2*x*f + g) in I
True

sage: (2*x*f + 2*z*h + y^3).reduce(B)
y^3

sage: (2*x*f + 2*z*h + 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']
sage: I = sage.rings.ideal.Cyclic(R,6)
sage: B = I.groebner_basis()
sage: len(B)
45
```

We compute in a quotient of a polynomial ring over $\mathbf{Z}/17\mathbf{Z}$:

```
sage: R.<x,y> = ZZ[]
sage: S.<a,b> = R.quotient((x^2 + y^2, 17))
sage: S
Quotient of Multivariate Polynomial Ring in x, y over Integer Ring
by the ideal (x^2 + y^2, 17)

sage: a^2 + b^2 == 0
True
sage: a^3 - b^2
-a*b^2 - b^2
```

Note that the result of a computation is not necessarily reduced:

```
sage: (a+b)^17
256*a*b^16 + 256*b^17
sage: S(17) == 0
True
```

Or we can work with $\mathbf{Z}/17\mathbf{Z}$ directly:

```
sage: R.<x,y> = Zmod(17)[]
sage: S.<a,b> = R.quotient((x^2 + y^2,))
sage: S
Quotient of Multivariate Polynomial Ring in x, y over Ring of
integers modulo 17 by the ideal (x^2 + y^2)

sage: a^2 + b^2 == 0
True
sage: a^3 - b^2
-a*b^2 - b^2
sage: (a+b)^17
a*b^16 + b^17
sage: S(17) == 0
True
```

Working with a polynomial ring over $\mathbf{Z}$:

```
sage: R.<x,y,z,w> = ZZ[]
sage: I = ideal(x^2 + y^2 - z^2 - w^2, x-y)
sage: J = I^2
sage: J.groebner_basis()
[4*y^4 - 4*y^2*z^2 + z^4 - 4*y^2*w^2 + 2*z^2*w^2 + w^4,
 2*x*y^2 - 2*y^3 - x*z^2 + y*z^2 - x*w^2 + y*w^2,
 x^2 - 2*x*y + y^2]

sage: y^2 - 2*x*y + x^2 in J
True
sage: 0 in J
True
```

We do a Groebner basis computation over a number field:

```
sage: K.<zeta> = CyclotomicField(3)
sage: R.<x,y,z> = K[]; R
Multivariate Polynomial Ring in x, y, z over Cyclotomic Field of order 3 and degree 2

sage: i = ideal(x - zeta*y + 1, x^3 - zeta*y^3); i
Ideal (x + (-zeta)*y + 1, x^3 + (-zeta)*y^3) of Multivariate
Polynomial Ring in x, y, z over Cyclotomic Field of order 3 and degree 2

sage: i.groebner_basis()
[y^3 + (2*zeta + 1)*y^2 + (zeta - 1)*y + (-1/3*zeta - 2/3), x + (-zeta)*y + 1]

sage: S = R.quotient(i); S
Quotient of Multivariate Polynomial Ring in x, y, z over
Cyclotomic Field of order 3 and degree 2 by the ideal (x +
(-zeta)*y + 1, x^3 + (-zeta)*y^3)

sage: S.0  - zeta*S.1
-1
```

```
sage: S.0^3 - zeta*S.1^3
0
```

Two examples from the Mathematica documentation (done in Sage):

We compute a Groebner basis:

```
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:

```
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 $\mathbf{Z}$ 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 $\mathbf{Z}[x, y, z]$, and note that the Groebner basis of I over $\mathbf{Q}$ contains 1, so there are no solutions over $\mathbf{Q}$ or an algebraic closure of it (this is not surprising as there are 4 equations in 3 unknowns).

```
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 $\mathbf{Z}$), we note that there is a certain integer in the ideal which is not 1.

```
sage: I.groebner_basis()
[x + 130433*y + 59079*z, y^2 + 3*y + 17220, y*z + 5*y + 14504, 2*y + 158864, z^2 + 17223, 2*z +
```

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.

```
sage: factor(164878)
2 * 7 * 11777

sage: I.change_ring(P.change_ring( GF(2) )).groebner_basis()
[x + y + z, y^2 + y, y*z + y, z^2 + 1]
sage: I.change_ring(P.change_ring( GF(7) )).groebner_basis()
[x - 1, y + 3, z - 2]
sage: I.change_ring(P.change_ring( GF(11777 ))).groebner_basis()
[x + 5633, y - 3007, z - 2626]
```

The Groebner basis modulo any product of the prime factors is also non-trivial.

```
sage: I.change_ring(P.change_ring( IntegerModRing(2*7) )).groebner_basis()
[x + y + z, y^2 + 3*y, y*z + 11*y + 4, 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: I.change_ring( P.change_ring( GF(3) ) ).groebner_basis()
[1]
```

TESTS:

```
sage: x,y,z = QQ['x,y,z'].gens()
sage: I = ideal(x^5 + y^4 + z^3 - 1,  x^3 + y^3 + z^2 - 1)
sage: I == loads(dumps(I))
True
```

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`.

class `sage.rings.polynomial.multi_polynomial_ideal.`**`MPolynomialIdeal`**(*ring*, *gens*, *coerce=True*)

Bases: `sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr`, `sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_macaulay2_repr`, `sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_magma_repr`, `sage.rings.ideal.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` - coerce elements to the ring `ring`?

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(IntegerRing(), 2, order='lex')
sage: R.ideal([x, y])
Ideal (x, y) of Multivariate Polynomial Ring in x, y over Integer Ring
sage: R.<x0,x1> = GF(3)[]
sage: R.ideal([x0^2, x1^3])
Ideal (x0^2, x1^3) of Multivariate Polynomial Ring in x0, x1 over Finite Field of size 3
```

`basis`

Shortcut to `gens()`.

EXAMPLE:

```
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 $g_1, ..., g_n$ of self as returned by `self.gens()`.

INPUT:

- `P` - a multivariate polynomial ring

EXAMPLE:

```
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 $\{f_1, ..., f_m\} \subset K[x_1, ..., x_n]$ be homogeneous polynomials of degrees $d_1, ..., d_m$ respectively. This sequence is semi-regular if:

- $\{f_1, ..., f_m\} \neq K[x_1, ..., x_n]$
- for all $1 \leq i \leq m$ and $g \in K[x_1, \dots, x_n]$: $deg(g \cdot pi) < D$ and $g \cdot f_i \in < f_1, \dots, f_{i-1} >$ implies that $g \in < f_1, ..., f_{i-1} >$ 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 $f_1, ..., f_m$ of respective degrees $d_1, ..., d_m$ is given by the index of the first non-positive coefficient of:

$$\sum c_k z^k = \frac{\prod(1-z^{d_i})}{(1-z)^n}$$

EXAMPLE:

We consider a homogeneous example:

```
sage: n = 8
sage: K = GF(127)
sage: P = PolynomialRing(K,n,'x')
sage: s = [K.random_element() for _ in range(n)]
sage: L = []
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()
4
```

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 [BFS04].

REFERENCES:

gens()
Return a set of generators / a basis of this ideal. This is usually the set of generators provided during object creation.

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQ,2)
sage: I = Ideal([x,y+1]); I
Ideal (x, y + 1) of Multivariate Polynomial Ring in x, y over Rational Field
sage: I.gens()
[x, y + 1]
```

groebner_basis(*algorithm='‘*, *deg_bound=None*, *mult_bound=None*, *prot=False*, **args*, ***kwds*)
Return the reduced Groebner basis of this ideal.

A Groebner basis $g_1, ..., g_n$ for an ideal I is a generating set such that $< LM(g_i) >= LM(I)$, i.e., the leading monomial ideal of I is spanned by the leading terms of $g_1, ..., g_n$. Groebner bases are the key concept in computational ideal theory in multivariate polynomial rings which allows a variety of problems to be solved.

Additionally, a *reduced* Groebner basis G is a unique representation for the ideal $< G >$ with respect to the chosen monomial ordering.

INPUT:

- **`algorithm` - determines the algorithm to use, see below** for available algorithms.
- `deg_bound` - only compute to degree `deg_bound`, that is, ignore all S-polynomials of higher degree. (default: `None`)
- `mult_bound` - the computation is stopped if the ideal is zero-dimensional in a ring with local ordering and its multiplicity is lower than `mult_bound`. Singular only. (default: `None`)

- `prot` - if set to `True` the computation protocol of the underlying implementation is printed. If an algorithm from the `singular:` or `magma:` family is used, `prot` may also be `sage` in which case the output is parsed and printed in a common format where the amount of information printed can be controlled via calls to `set_verbose()`.
- **`*args` - additional parameters passed to the respective** implementations
- **`**kwds` - additional keyword parameters passed to the** respective implementations

ALGORITHMS:

'' autoselect (default)

'singular:groebner' Singular's `groebner` command

'singular:std' Singular's `std` command

'singular:stdhilb' Singular's `stdhib` command

'singular:stdfglm' Singular's `stdfglm` command

'singular:slimgb' Singular's `slimgb` command

'libsingular:groebner' libSingular's `groebner` command

'libsingular:std' libSingular's `std` command

'libsingular:slimgb' libSingular's `slimgb` command

'libsingular:stdhilb' libSingular's `stdhib` command

'libsingular:stdfglm' libSingular's `stdfglm` command

'toy:buchberger' Sage's toy/educational buchberger without Buchberger criteria

'toy:buchberger2' Sage's toy/educational buchberger with Buchberger criteria

'toy:d_basis' Sage's toy/educational algorithm for computation over PIDs

'macaulay2:gb' Macaulay2's `gb` command (if available)

'magma:GroebnerBasis' Magma's `Groebnerbasis` command (if available)

'ginv:TQ', 'ginv:TQBlockHigh', 'ginv:TQBlockLow' and 'ginv:TQDegree' One of GINV's implementations (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. However, the libSingular interface does not support pretty printing of computation protocols.

EXAMPLES:

Consider Katsura-3 over **Q** with lexicographical term ordering. We compute the reduced Groebner basis using every available implementation and check their equality.

```
sage: P.<a,b,c> = PolynomialRing(QQ,3, order='lex')
sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis()
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c

sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('libsingular:groebner')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c
```

```
sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('libsingular:std')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c

sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('libsingular:stdhilb')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c

sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('libsingular:stdfglm')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c

sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('libsingular:slimgb')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c
```

Note that `toy:buchberger` does not return the reduced Groebner basis,

```
sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('toy:buchberger')
[a^2 - a + 2*b^2 + 2*c^2,
 a*b + b*c - 1/2*b, a + 2*b + 2*c - 1,
 b^2 + 3*b*c - 1/2*b + 3*c^2 - c,
 b*c - 1/10*b + 6/5*c^2 - 2/5*c,
 b + 30*c^3 - 79/7*c^2 + 3/7*c,
 c^6 - 79/210*c^5 - 229/2100*c^4 + 121/2520*c^3 + 1/3150*c^2 - 11/12600*c,
 c^4 - 10/21*c^3 + 1/84*c^2 + 1/84*c]
```

but that `toy:buchberger2` does.:

```
sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('toy:buchberger2')
[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

sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('macaulay2:gb') # optional - macaulay2
[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

sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('magma:GroebnerBasis') # optional - magma
[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
```

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)
[x^3 + y^2, x^2*y + 1]
sage: I.groebner_basis()
[x^3 + y^2, x^2*y + 1, y^3 - x]
sage: I.groebner_basis(deg_bound=2)
[x^3 + y^2, x^2*y + 1]
```

A protocol is printed, if the verbosity level is at least 2, or if the argument `prot` is provided. Historically, the protocol did not appear during doctests, so, we skip the examples with protocol output.

```
sage: set_verbose(2)
sage: I = R*[x^3+y^2,x^2*y+1]
sage: I.groebner_basis()  # not tested
std in (0),(x,y),(dp(2),C)
[...:2]3ss4s6
(S:2)--
product criterion:1 chain criterion:0
[x^3 + y^2, x^2*y + 1, y^3 - x]
sage: I.groebner_basis(prot=False)
std in (0),(x,y),(dp(2),C)
[...:2]3ss4s6
(S:2)--
product criterion:1 chain criterion:0
[x^3 + y^2, x^2*y + 1, y^3 - x]
sage: set_verbose(0)
sage: I.groebner_basis(prot=True)  # not tested
std in (0),(x,y),(dp(2),C)
[...:2]3ss4s6
(S:2)--
product criterion:1 chain criterion:0
[x^3 + y^2, x^2*y + 1, y^3 - x]
```

The list of available options is provided at `LibSingularOptions`.

Note that Groebner bases over $\mathbf{Z}$ can also be computed.:

```
sage: P.<a,b,c> = PolynomialRing(ZZ,3)
sage: I = P * (a + 2*b + 2*c - 1, a^2 - a + 2*b^2 + 2*c^2, 2*a*b + 2*b*c - b)
sage: I.groebner_basis()
[b^3 - 23*b*c^2 + 3*b^2 + 5*b*c, 2*b*c^2 - 6*c^3 - b^2 - b*c + 2*c^2,
 42*c^3 + 5*b^2 + 4*b*c - 14*c^2, 2*b^2 + 6*b*c + 6*c^2 - b - 2*c,
 10*b*c + 12*c^2 - b - 4*c, a + 2*b + 2*c - 1]

sage: I.groebner_basis('macaulay2') # optional - macaulay2
[b^3 + b*c^2 + 12*c^3 + b^2 + b*c - 4*c^2,
 2*b*c^2 - 6*c^3 + b^2 + 5*b*c + 8*c^2 - b - 2*c,
 42*c^3 + b^2 + 2*b*c - 14*c^2 + b,
 2*b^2 - 4*b*c - 6*c^2 + 2*c, 10*b*c + 12*c^2 - b - 4*c,
 a + 2*b + 2*c - 1]
```

Groebner bases over $\mathbf{Z}/n\mathbf{Z}$ are also supported:

```
sage: P.<a,b,c> = PolynomialRing(Zmod(1000),3)
sage: I = P * (a + 2*b + 2*c - 1, a^2 - a + 2*b^2 + 2*c^2, 2*a*b + 2*b*c - b)
sage: I.groebner_basis()
[b*c^2 + 992*b*c + 712*c^2 + 332*b + 96*c,
 2*c^3 + 589*b*c + 862*c^2 + 762*b + 268*c,
 b^2 + 438*b*c + 281*b,
 5*b*c + 156*c^2 + 112*b + 948*c,
 50*c^2 + 600*b + 650*c, a + 2*b + 2*c + 999, 125*b]
```

Sage also supports local orderings:

```
sage: P.<x,y,z> = PolynomialRing(QQ,3,order='negdegrevlex')
sage: I = P * (  x*y*z + z^5, 2*x^2 + y^3 + z^7, 3*z^5 +y ^5 )
sage: I.groebner_basis()
[x^2 + 1/2*y^3, x*y*z + z^5, y^5 + 3*z^5, y^4*z - 2*x*z^5, z^6]
```

We can represent every element in the ideal as a combination of the generators using the `lift()` method:

```
sage: P.<x,y,z> = PolynomialRing(QQ,3)
sage: I = P * ( x*y*z + z^5, 2*x^2 + y^3 + z^7, 3*z^5 +y ^5 )
sage: J = Ideal(I.groebner_basis())
sage: f = sum(P.random_element(terms=2)*f for f in I.gens())
sage: f
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 + 9
sage: f.lift(I.gens())
[2*x + 95*z, 1/2*y^2 - 1/4*y*z, 0]
sage: l = f.lift(J.gens()); l
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1/2*y^2 + 1/4*y*z, 1/2*y^2*z^2 - 1/4*y*z^3 + 2*x +
sage: sum(map(mul, zip(l,J.gens()))) == f
True
```

Groebner bases over fraction fields of polynomial rings are also supported:

```
sage: P.<t> = QQ[]
sage: F = Frac(P)
sage: R.<X,Y,Z> = F[]
sage: I = Ideal([f + P.random_element() for f in sage.rings.ideal.Katsura(R).gens()])
sage: I.groebner_basis()
[Z^3 + (79/105*t^2 - 79/105*t + 79/630)*Z^2 + (-11/105*t^4 + 22/105*t^3 - 17/45*t^2 + 197/63
Y^2 + (-3/5)*Z^2 + (2/5*t^2 - 2/5*t + 1/15)*Y + (-2/5*t^2 + 2/5*t - 1/15)*Z - 1/10*t^4 + 1/5
Y*Z + 6/5*Z^2 + (1/5*t^2 - 1/5*t + 1/30)*Y + (4/5*t^2 - 4/5*t + 2/15)*Z + 1/5*t^4 - 2/5*t^3
```

In cases where a characteristic cannot be determined, we use a toy implementation of Buchberger's algorithm (see trac ticket #6581):

```
sage: R.<a,b> = QQ[]; I = R.ideal(a^2+b^2-1)
sage: Q = QuotientRing(R,I); K = Frac(Q)
sage: R2.<x,y> = K[]; J = R2.ideal([(a^2+b^2)*x + y, x+y])
sage: J.groebner_basis()
verbose 0 (...: multi_polynomial_ideal.py, groebner_basis) Warning: falling back to very slo
[x + y]
```

ALGORITHM:

Uses Singular, Magma (if available), Macaulay2 (if available), or a toy implementation.

groebner_fan(*is_groebner_basis=False*, *symmetry=None*, *verbose=False*)

Return the Groebner fan of this ideal.

The base ring must be $\mathbf{Q}$ or a finite field $\mathbf{F}_p$ 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 generators of this ideal.

INPUT:

- `h` - variable name or variable in cover ring (default: 'h')

EXAMPLE:

```
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.

EXAMPLE:

```
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
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
Graphics object consisting of 1 graphics primitive

sage: I = R.ideal([y^2 + x^2*(1/4) - 1])
sage: I.plot()                                # ellipse
Graphics object consisting of 1 graphics primitive

sage: I = R.ideal([y^2-(x^2-1)*(x-2)])
sage: I.plot()                                # elliptic curve
Graphics object consisting of 1 graphics primitive

sage: f = ((x+3)^3 + 2*(x+3)^2 - y^2)*(x^3 - y^2)*((x-3)^3-2*(x-3)^2-y^2)
sage: I = R.ideal(f)
sage: I.plot()                                # the Singular logo
Graphics object consisting of 1 graphics primitive
```

This used to be trac ticket #5267:

```
sage: I = R.ideal([-x^2*y+1])
sage: I.plot()
Graphics object consisting of 1 graphics primitive
```

AUTHORS:

- Martin Albrecht (2008-09)

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()`.

EXAMPLE:

We compute a uniformly random element up to the provided degree.:

```
sage: P.<x,y,z> = GF(127)[]
sage: I = sage.rings.ideal.Katsura(P)
sage: I.random_element(degree=4, compute_gb=True, terms=infinity)
34*x^4 - 33*x^3*y + 45*x^2*y^2 - 51*x*y^3 - 55*y^4 + 43*x^3*z ... - 28*y - 33*z + 45
```

Note that sampling uniformly at random from the ideal at some large enough degree is equivalent to computing a Gröbner basis. We give an example showing how to compute a Gröbner basis if we can sample uniformly at random from an ideal:

```
sage: n = 3; d = 4
sage: P = PolynomialRing(GF(127), n, 'x')
sage: I = sage.rings.ideal.Cyclic(P)
```

1. We sample n^d uniformly random elements in the ideal:

   ```
   sage: F = Sequence(I.random_element(degree=d, compute_gb=True, terms=infinity) for _ in 
   ```

2. We linearize and compute the echelon form:

   ```
   sage: A,v = F.coefficient_matrix()
   sage: A.echelonize()
   ```

3. The result is the desired Gröbner basis:

   ```
   sage: G = Sequence((A*v).list())
   sage: G.is_groebner()
   True
   sage: Ideal(G) == I
   True
   ```

We return some element in the ideal with no guarantee on the distribution:

```
sage: P = PolynomialRing(GF(127), 10, 'x')
sage: I = sage.rings.ideal.Katsura(P)
sage: I.random_element(degree=3)
7*x0^2*x1 + 14*x1^3 + 57*x0*x1*x2 - 32*x1^2*x2 - ... + 49*x4 + 48*x5 - 40*x7 - 6*x8
```

We show that the default method does not sample uniformly at random from the ideal:

```
sage: P.<x,y,z> = GF(127)[]
sage: G = Sequence([x+7, y-2, z+110])
sage: I = Ideal([sum(P.random_element() * g for g in G) for _ in range(4)])
sage: all(I.random_element(degree=1) == 0 for _ in range(100))
True
```

If degree equals the degree of the generators a random linear combination of the generators is returned:

```
sage: P.<x,y> = QQ[]
sage: I = P.ideal([x^2,y^2])
sage: I.random_element(degree=2)
52*x^2 - 8/3*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:

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: I = (x^3 + y, y)*R
sage: I.reduce(y)
0
sage: I.reduce(x^3)
0
```

```
sage: I.reduce(x - y)
x

sage: I = (y^2 - (x^3 + x))*R
sage: I.reduce(x^3)
y^2 - x
sage: I.reduce(x^6)
y^4 - 2*x*y^2 + x^2
sage: (y^2 - x)^2
y^4 - 2*x*y^2 + x^2
```

Note: Requires computation of a Groebner basis, which can be a very expensive operation.

subs(*in_dict=None*, ***kwds*)

Substitute variables.

This method substitutes some variables in the polynomials that generate the ideal with given values. Variables that are not specified in the input remain unchanged.

INPUT:

- `in_dict` – (optional) dictionary of inputs
- `**kwds` – named parameters

OUTPUT:

A new ideal with modified generators. If possible, in the same polynomial ring. Raises a `TypeError` if no common polynomial ring of the substituted generators can be found.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(ZZ,2,'xy')
sage: I = R.ideal(x^5+y^5, x^2 + y + x^2*y^2 + 5); I
Ideal (x^5 + y^5, x^2*y^2 + x^2 + y + 5) of Multivariate Polynomial Ring in x, y over Intege
sage: I.subs(x=y)
Ideal (2*y^5, y^4 + y^2 + y + 5) of Multivariate Polynomial Ring in x, y over Integer Ring
sage: 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:

```
sage: R.<a,b> = PolynomialRing(QQ,2)
sage: S.<x,y> = PolynomialRing(QQ,2)
sage: I = R.ideal(a^2+b^2+a-b+2); I
Ideal (a^2 + b^2 + a - b + 2) of Multivariate Polynomial Ring in a, b over Rational Field
sage: 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 mulitvariate polynomial ring:

```
sage: T.<t> = PolynomialRing(QQ)
sage: I.subs(a=t, b=t)
Principal ideal (t^2 + 1) of Univariate Polynomial Ring in t over Rational Field
sage: var("z")
z
sage: I.subs(a=z, b=z)
Principal ideal (2*z^2 + 2) of Symbolic Ring
```

Variables that are not substituted remain unchanged:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: I = R.ideal(x^2+y^2+x-y+2); I
Ideal (x^2 + y^2 + x - y + 2) of Multivariate Polynomial Ring in x, y over Rational Field
sage: I.subs(x=1)
Ideal (y^2 - y + 4) of Multivariate Polynomial Ring in x, y over Rational Field
```

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 $Res_{L/k}(X)$, defined over k. It is useful for reducing questions about varieties over large fields to questions about more complicated varieties over smaller fields.

This function does not compute this Weil restriction directly but computes on generating sets of polynomial ideals:

Let d be the degree of the field extension L/k, let a a generator of L/k and p the minimal polynomial of L/k. Denote this ideal by I.

Specifically, this function first maps each variable x to its representation over k: $\sum_{i=0}^{d-1} a^i x_i$. Then each generator of I is evaluated over these representations and reduced modulo the minimal polynomial p. The result is interpreted as a univariate polynomial in a and its coefficients are the new generators of the returned ideal.

If the input and the output ideals are radical, this is equivalent to the statement about algebraic varieties above.

OUTPUT: MPolynomial Ideal

EXAMPLE:

```
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, 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, x1: 1, x0: 1, y0: 0}]

sage: J.weil_restriction() # returns J
Ideal (x0*y0 + x1*y1 + 1, x1*y0 + x0*y1 + x1*y1, x1 + 1, x0 + x1, x0^2 +
x0, x1^2 + x1, y0^2 + y0, y1^2 + y1) of Multivariate Polynomial Ring in
x0, x1, y0, y1 over Finite Field of size 2

sage: k.<a> = GF(3^5)
sage: P.<x,y,z> = PolynomialRing(k)
sage: I = sage.rings.ideal.Katsura(P)
sage: I.dimension()
0
sage: I.variety()
[{y: 0, z: 0, x: 1}]
```

```
sage: J = I.weil_restriction(); J
Ideal (x0 - y0 - z0 - 1, x1 - y1 - z1, x2 - y2 - z2, x3 - y3 - z3, x4 -
y4 - z4, x0^2 + x2*x3 + x1*x4 - y0^2 - y2*y3 - y1*y4 - z0^2 - z2*z3 -
z1*z4 - x0, -x0*x1 - x2*x3 - x3^2 - x1*x4 + x2*x4 + y0*y1 + y2*y3 + y3^2
+ y1*y4 - y2*y4 + z0*z1 + z2*z3 + z3^2 + z1*z4 - z2*z4 - x1, x1^2 -
x0*x2 + x3^2 - x2*x4 + x3*x4 - y1^2 + y0*y2 - y3^2 + y2*y4 - y3*y4 -
z1^2 + z0*z2 - z3^2 + z2*z4 - z3*z4 - x2, -x1*x2 - x0*x3 - x3*x4 - x4^2
+ y1*y2 + y0*y3 + y3*y4 + y4^2 + z1*z2 + z0*z3 + z3*z4 + z4^2 - x3, x2^2
- x1*x3 - x0*x4 + x4^2 - y2^2 + y1*y3 + y0*y4 - y4^2 - z2^2 + z1*z3 +
z0*z4 - z4^2 - x4, -x0*y0 + x4*y1 + x3*y2 + x2*y3 + x1*y4 - y0*z0 +
y4*z1 + y3*z2 + y2*z3 + y1*z4 - y0, -x1*y0 - x0*y1 - x4*y1 - x3*y2 +
x4*y2 - x2*y3 + x3*y3 - x1*y4 + x2*y4 - y1*z0 - y0*z1 - y4*z1 - y3*z2 +
y4*z2 - y2*z3 + y3*z3 - y1*z4 + y2*z4 - y1, -x2*y0 - x1*y1 - x0*y2 -
x4*y2 - x3*y3 + x4*y3 - x2*y4 + x3*y4 - y2*z0 - y1*z1 - y0*z2 - y4*z2 -
y3*z3 + y4*z3 - y2*z4 + y3*z4 - y2, -x3*y0 - x2*y1 - x1*y2 - x0*y3 -
x4*y3 - x3*y4 + x4*y4 - y3*z0 - y2*z1 - y1*z2 - y0*z3 - y4*z3 - y3*z4 +
y4*z4 - y3, -x4*y0 - x3*y1 - x2*y2 - x1*y3 - x0*y4 - x4*y4 - y4*z0 -
y3*z1 - y2*z2 - y1*z3 - y0*z4 - y4*z4 - y4) of Multivariate Polynomial
Ring in x0, x1, x2, x3, x4, y0, y1, y2, y3, y4, z0, z1, z2, z3, z4 over
Finite Field of size 3
sage: J += sage.rings.ideal.FieldIdeal(J.ring()) # ensure radical ideal
sage: J.variety()
[{y1: 0, y4: 0, x4: 0, y2: 0, y3: 0, y0: 0, x2: 0, z4: 0, z3: 0, z2: 0, x1: 0, z1: 0, z0: 0,
```

Weil restrictions are often used to study elliptic curves over extension fields so we give a simple example involving those:

```
sage: K.<a> = QuadraticField(1/3)
sage: E = EllipticCurve(K,[1,2,3,4,5])
```

We pick a point on E:

```
sage: p = E.lift_x(1); p
(1 : 2 : 1)

sage: I = E.defining_ideal(); I
Ideal (-x^3 - 2*x^2*z + x*y*z + y^2*z - 4*x*z^2 + 3*y*z^2 - 5*z^3)
of Multivariate Polynomial Ring in x, y, z over Number Field in a with defining polynomial x
```

Of course, the point p is a root of all generators of I:

```
sage: I.subs(x=1,y=2,z=1)
Ideal (0) of Multivariate Polynomial Ring in x, y, z over
Number Field in a with defining polynomial x^2 - 1/3
```

I is also radical:

```
sage: I.radical() == I
True
```

So we compute its Weil restriction:

```
sage: J = I.weil_restriction()
sage: J
Ideal (-x0^3 - x0*x1^2 - 2*x0^2*z0 - 2/3*x1^2*z0 + x0*y0*z0 + y0^2*z0 +
1/3*x1*y1*z0 + 1/3*y1^2*z0 - 4*x0*z0^2 + 3*y0*z0^2 - 5*z0^3 -
4/3*x0*x1*z1 + 1/3*x1*y0*z1 + 1/3*x0*y1*z1 + 2/3*y0*y1*z1 - 8/3*x1*z0*z1
+ 2*y1*z0*z1 - 4/3*x0*z1^2 + y0*z1^2 - 5*z0*z1^2, -3*x0^2*x1 - 1/3*x1^3
- 4*x0*x1*z0 + x1*y0*z0 + x0*y1*z0 + 2*y0*y1*z0 - 4*x1*z0^2 + 3*y1*z0^2
- 2*x0^2*z1 - 2/3*x1^2*z1 + x0*y0*z1 + y0^2*z1 + 1/3*x1*y1*z1 +
1/3*y1^2*z1 - 8*x0*z0*z1 + 6*y0*z0*z1 - 15*z0^2*z1 - 4/3*x1*z1^2 +
y1*z1^2 - 5/3*z1^3) of Multivariate Polynomial Ring in x0, x1, y0, y1,
z0, z1 over Rational Field
```

We can check that the point p is still a root of all generators of J:

```
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:

```
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**

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**

class sage.rings.polynomial.multi_polynomial_ideal.**MPolynomialIdeal_singular_base_repr**

syzygy_module()

Computes the first syzygy (i.e., the module of relations of the given generators) of the ideal.

EXAMPLE:

```
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: sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_base_repr

An ideal in a multivariate polynomial ring, which has an underlying Singular ring associated to it.

associated_primes(*algorithm='sy'*)

Return a list of the associated primes of primary ideals of which the intersection is I = `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 $n \in \mathbf{Z}$.

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 $n \in \mathbf{Z}$, is called the *associated prime* of Q.

If I is a proper ideal of the ring R then there exists a decomposition in primary ideals Q_i such that

- their intersection is I
- none of the Q_i contains the intersection of the rest, and
- the associated prime ideals of Q_i are pairwise different.

This method returns the associated primes of the Q_i.

INPUT:

- `algorithm` - string:
- `'sy'` - (default) use the shimoyama-yokoyama algorithm
- `'gtz'` - use the gianni-trager-zacharias algorithm

OUTPUT:

- `list` - a list of associated primes

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y-z^2)*R
sage: pd = I.associated_primes(); pd
[Ideal (z^3 + 2, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field,
 Ideal (z^2 + 1, y + 1) of Multivariate Polynomial Ring in x, y, z over Rational Field]
```

ALGORITHM:

Uses Singular.

REFERENCES:

- Thomas Becker and Volker Weispfenning. Groebner Bases - A Computational Approach To Commutative Algebra. Springer, New York 1993.

basis_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 $Syz(LM(I))$:

$$S * G = \sum_{i=0}^{m} h_i g_i ---- >_G 0.$$

ALGORITHM:

Uses Singular.

EXAMPLE:

```
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))
sage: l = [u6 + u5 + u4 + u3 + u2 - 3791*h, \
          U6 + U5 + U4 + U3 + U2 - 3791*h, \
          U2*u2 - h^2, U3*u3 - h^2, U4*u4 - h^2, \
          U5*u4 + U5*u3 + U4*u3 + U5*u2 + U4*u2 + U3*u2 - 3791*U5*h - 3791*U4*h - 3791*U3*h
          U4*u5 + U3*u5 + U2*u5 + U3*u4 + U2*u4 + U2*u3 - 3791*u5*h - 3791*u4*h - 3791*u3*h
          U5*u5 - h^2, U4*U2*u3 + U5*U3*u2 + U4*U3*u2 + U3^2*u2 - 3791*U5*U3*h - 3791*U4*U3
           - 3791*U4*U2*h + U3*U2*h - 3791*U2^2*h - 3791*U4*u3*h - 3791*U4*u2*h - 3791*U3*u
          U3*u5*u4 + U2*u5*u4 + U3*u4^2 + U2*u4^2 + U2*u4*u3 - 3791*u5*u4*h - 3791*u4^2*h -
          U2*u5*u4*u3 + U2*u4^2*u3 + U2*u4*u3^2 - 3791*u5*u4*u3*h - 3791*u4^2*u3*h - 3791*u
          U5^2*U4*u3 + U5*U4^2*u3 + U5^2*U4*u2 + U5*U4^2*u2 + U5^2*U3*u2 + 2*U5*U4*U3*u2 +
           + U5*U4*U3*h - 3791*U5*U3^2*h - 3791*U5^2*U2*h + U5*U4*U2*h + U5*U3*U2*h - 3791*
           - U4^2*h^2 - 947*U5*U3*h^2 - U4*U3*h^2 - 948*U5*U2*h^2 - U4*U2*h^2 - 1422*U5*h^3
          u5*u4*u3*u2*h + u4^2*u3*u2*h + u4*u3^2*u2*h + u4*u3*u2^2*h + 2*u5*u4*u3*h^2 + 2*u
           + 2*u5*u3*u2*h^2 + 1899*u4*u3*u2*h^2, \
          U5^2*U4*U3*u2 + U5*U4^2*U3*u2 + U5*U4*U3^2*u2 - 3791*U5^2*U4*U3*h - 3791*U5*U4^2*
           + 3791*U5*U4*U3*u2*h + U5^2*U4*h^2 + U5*U4^2*h^2 + U5^2*U3*h^2 - U4^2*U3*h^2 - U
           - U5*U3*U2*h^2 - U4*U3*U2*h^2 + 3791*U5*U4*h^3 + 3791*U5*U3*h^3 + 3791*U4*U3*h^3
          u4^2*u3*u2*h^2 + 1515*u5*u3^2*u2*h^2 + u4*u3^2*u2*h^2 + 1515*u5*u4*u2^2*h^2 + 151
           + 1521*u5*u4*u3*h^3 - 3028*u4^2*u3*h^3 - 3028*u4*u3^2*h^3 + 1521*u5*u4*u2*h^3 -
          U5^2*U4*U3*U2*h + U5*U4^2*U3*U2*h + U5*U4*U3^2*U2*h + U5*U4*U3*U2^2*h + 2*U5^2*U4
           + 2*U5^2*U4*U2*h^2 + 2*U5*U4^2*U2*h^2 + 2*U5^2*U3*U2*h^2 - 2*U4^2*U3*U2*h^2 - 2*
            - 2*U5*U4*U2^2*h^2 - 2*U5*U3*U2^2*h^2 - 2*U4*U3*U2^2*h^2 - U5*U4*U3*h^3 - U5*U4

sage: Ideal(l).basis_is_groebner()
False
sage: gb = Ideal(l).groebner_basis()
sage: Ideal(gb).basis_is_groebner()
True
```

Note: From the Singular Manual for the reduce function we use in this method: 'The result may have no meaning if the second argument (`self`) is not a standard basis'. I (malb) believe this refers to the mathematical fact that the results may have no meaning if self is 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(*algorithm='sy'*)

Return a list of primary ideals such that their intersection is `self`, together with the associated prime ideals.

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 $n \in \mathbf{Z}$.

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 $n \in \mathbf{Z}$), 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 P_i 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 pairs (Q_i, P_i), where the Q_i form a primary decomposition of `self` and P_i is the associated prime of Q_i.

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.complete_primary_decomposition(); pd
[(Ideal (z^6 + 4*z^3 + 4, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational
  Ideal (z^3 + 2, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field),
 (Ideal (z^2 + 1, y + 1) of Multivariate Polynomial Ring in x, y, z over Rational Field,
  Ideal (z^2 + 1, y + 1) of Multivariate Polynomial Ring in x, y, z over Rational Field)]

sage: I.primary_decomposition_complete(algorithm = 'gtz')
[(Ideal (z^6 + 4*z^3 + 4, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational
  Ideal (z^3 + 2, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field),
 (Ideal (z^2 + 1, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field,
  Ideal (z^2 + 1, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field)]

sage: from functools import reduce
sage: reduce(lambda Qi,Qj: Qi.intersection(Qj), [Qi for (Qi,radQi) in pd]) == I
True

sage: [Qi.radical() == radQi for (Qi,radQi) in pd]
[True, True]

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.complete_primary_decomposition()
Traceback (most recent call last):
...
ValueError: Coefficient ring must be a field for function 'complete_primary_decomposition'.
```

ALGORITHM:

Uses Singular.

Note: See [BW93] for an introduction to primary decomposition.

TESTS:

Check that trac ticket #15745 is fixed:

```
sage: R.<x,y>= QQ[]
sage: I = Ideal(R(1))
sage: I.complete_primary_decomposition()
```

```
[]
sage: I.is_prime()
False
```

dimension(*singular='singular_default'*)

The dimension of the ring modulo this ideal.

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(GF(32003),order='degrevlex')
sage: I = ideal(x^2-y,x^3)
sage: I.dimension()
1
```

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".

EXAMPLE:

```
sage: R.<x,y> = PolynomialRing(GF(2147483659),order='lex')
sage: I = R.ideal([x*y,x*y+1])
sage: I.dimension()
verbose 0 (...: multi_polynomial_ideal.py, dimension) Warning: falling back to very slow toy
0
sage: I=ideal([x*(x*y+1),y*(x*y+1)])
sage: I.dimension()
verbose 0 (...: multi_polynomial_ideal.py, dimension) Warning: falling back to very slow toy
1
sage: I = R.ideal([x^3*y,x*y^2])
sage: I.dimension()
verbose 0 (...: multi_polynomial_ideal.py, dimension) Warning: falling back to very slow toy
1
sage: R.<x,y> = PolynomialRing(GF(2147483659),order='lex')
sage: I = R.ideal(0)
sage: I.dimension()
verbose 0 (...: multi_polynomial_ideal.py, dimension) Warning: falling back to very slow toy
2
```

ALGORITHM:

Uses Singular, unless the characteristic is too large.

Note: Requires computation of a Groebner basis, which can be a very expensive operation.

elimination_ideal(*variables*)

Returns the elimination ideal this ideal with respect to the variables given in `variables`.

INPUT:

•`variables` - a list or tuple of variables in `self.ring()`

EXAMPLE:

```
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: I.elimination_ideal([t,s])
Ideal (y^2 - x*z, x*y - z, x^2 - y) of Multivariate
Polynomial Ring in x, y, t, s, z over Rational Field
```

ALGORITHM:

Uses Singular.

Note: Requires computation of a Groebner basis, which can be a very expensive operation.

genus()

Return the genus of the projective curve defined by this ideal, which must be 1 dimensional.

EXAMPLE:

Consider the hyperelliptic curve $y^2 = 4x^5 - 30x^3 + 45x - 22$ over $\mathbf{Q}$, it has genus 2:

```
sage: P, x = PolynomialRing(QQ,"x").objgen()
sage: f = 4*x^5 - 30*x^3 + 45*x - 22
sage: C = HyperellipticCurve(f); C
Hyperelliptic Curve over Rational Field defined by y^2 = 4*x^5 - 30*x^3 + 45*x - 22
sage: C.genus()
2

sage: P.<x,y> = PolynomialRing(QQ)
sage: f = y^2 - 4*x^5 - 30*x^3 + 45*x - 22
sage: I = Ideal([f])
sage: I.genus()
2
```

TESTS:

Check that the answer is correct for reducible curves:

```
sage: R.<x, y, z> = QQ[]
sage: C = Curve(x^2 - 2*y^2)
sage: C.is_singular()
True
sage: C.genus()
-1
sage: Ideal(x^4+y^2*x+x).genus()
0
sage: T.<t1,t2,u1,u2> = QQ[]
sage: TJ = Ideal([t1^2 + u1^2 - 1,t2^2 + u2^2 - 1, (t1-t2)^2 + (u1-u2)^2 -1])
sage: TJ.genus()
-1
```

hilbert_numerator(*singular='singular_default'*, *grading=None*)

Return the Hilbert numerator of this ideal.

Let I = `self` be a homogeneous ideal and R = `self.ring()` be a graded commutative algebra ($R = \oplus R_d$) over a field K. Then the Hilbert function is defined as $H(d) = dim_K R_d$ and the Hilbert series of I is defined as the formal power series $H(d) = dim_K R_d$ and the Hilbert series of I is defined as the formal power series $HS(t) = \sum_0^\infty H(d)t^d$.

This power series can be expressed as $HS(t) = Q(t)/(1-t)^n$ where $Q(t)$ is a polynomial over Z and n the number of variables in R. This method returns $Q(t)$, the numerator; hence the name, $hilbert_numerator$.

An optional `grading` can be given, in which case the graded (or weighted) Hilbert numerator is given.

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([x^3*y^2 + 3*x^2*y^2*z + y^3*z^2 + z^5])
sage: I.hilbert_numerator()
-t^5 + 1
sage: R.<a,b> = PolynomialRing(QQ)
sage: J = R.ideal([a^2*b,a*b^2])
```

```
sage: J.hilbert_numerator()
t^4 - 2*t^3 + 1
sage: J.hilbert_numerator(grading=(10,3))
t^26 - t^23 - t^16 + 1
```

hilbert_polynomial()

Return the Hilbert polynomial of this ideal.

Let I = `self` be a homogeneous ideal and R = `self.ring()` be a graded commutative algebra ($R = \oplus R_d$) 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.

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([x^3*y^2 + 3*x^2*y^2*z + y^3*z^2 + z^5])
sage: I.hilbert_polynomial()
5*t - 5
```

hilbert_series(*singular='singular_default'*, *grading=None*)

Return the Hilbert series of this ideal.

Let I = `self` be a homogeneous ideal and R = `self.ring()` be a graded commutative algebra ($R = \oplus R_d$) 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 $HS(t) = \sum_0^\infty H(d)t^d$.

This power series can be expressed as $HS(t) = Q(t)/(1-t)^n$ where $Q(t)$ is a polynomial over Z and n the number of variables in R. This method returns $Q(t)/(1-t)^n$.

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()
(-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()
(t^3 - t^2 - t - 1)/(t - 1)
sage: J.hilbert_series(grading=(10,3))
(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)

sage: J = R.ideal([a^2*b^3, a*b^4 + a^3*b^2])
sage: J.hilbert_series(grading=[1,2])
(t^11 + t^8 - t^6 - t^5 - t^4 - t^3 - t^2 - t - 1)/(t^2 - 1)
sage: J.hilbert_series(grading=[2,1])
(2*t^7 - t^6 - t^4 - t^2 - 1)/(t - 1)
```

TESTS:

```
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(grading=5)
Traceback (most recent call last):
...
TypeError: Grading must be a list or a tuple of integers.
```

integral_closure (*p=0*, *r=True*, *singular='singular_default'*)

Let I = `self`.

Returns 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`)

EXAMPLE:

```
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 $(f_1, ..., f_n)$ this method returns $(g_1, ..., g_s)$ such that:

- $(f_1, ..., f_n) = (g_1, ..., g_s)$
- $LT(g_i)! = LT(g_j)$ for all $i! = j$
- $LT(g_i)$ **does not divide** m **for all monomials** m **of** $\{g_1, ..., g_{i-1}, g_{i+1}, ..., g_s\}$
- $LC(g_i) == 1$ for all i if the coefficient ring is a field.

EXAMPLE:

```
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([z*x+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*x+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 `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); K
Ideal (y^2, x*y, x^2) of Multivariate Polynomial Ring in x, y over Rational Field
sage: IJ = I*J; IJ
Ideal (x^2*y^2, x^3, y^3, x*y) of Multivariate Polynomial Ring in x, y over Rational Field
sage: IJ == K
False
```

Intersection of several ideals:

```
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)
Ideal (x*y*z^20 - x*y*z^3, x*y^2 - x*y*z^3, x^2*y + x*y*z^4) of Multivariate Polynomial Ring
```

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')
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 trac ticket #5982. Note that the quotient ring is not recognized as being a field at this time, so the fraction field is not the quotient ring itself:

```
sage: Q = R.quotient(I); Q
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 - y^2
sage: Q.fraction_field()
Fraction Field of Quotient of Multivariate Polynomial Ring in x, y over Rational Field by th
```

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: I.minimal_associated_primes ()
[Ideal (z^2 + 1, -z^2 + y) of Multivariate Polynomial Ring
in x, y, z over Rational Field, Ideal (z^3 + 2, -z^2 + y)
of Multivariate Polynomial Ring in x, y, z over Rational
Field]
```

ALGORITHM:

Uses Singular.

normal_basis(*algorithm='libsingular'*, *singular='singular_default'*)

Returns 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.

INPUT:

`algorithm` - defaults to use libsingular, if it is anything else we will use the `kbase()` command

EXAMPLES:

```
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]
```

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 **Q**.

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
Graphics object consisting of 1 graphics primitive
sage: I = R.ideal([y^2 - x^2 - 1])
sage: I.plot()        # hyperbola
Graphics object consisting of 1 graphics primitive
sage: I = R.ideal([y^2 + x^2*(1/4) - 1])
sage: I.plot()        # ellipse
Graphics object consisting of 1 graphics primitive
sage: I = R.ideal([y^2-(x^2-1)*(x-2)])
sage: I.plot()        # elliptic curve
Graphics object consisting of 1 graphics primitive
```

Implicit plotting in 3-d:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: I = R.ideal([y^2 + x^2*(1/4) - z])
sage: I.plot()          # a cone; optional - surf
sage: I = R.ideal([y^2 + z^2*(1/4) - x])
sage: I.plot()          # same code, from a different angle; optional - surf
sage: I = R.ideal([x^2*y^2+x^2*z^2+y^2*z^2-16*x*y*z])
sage: I.plot()          # Steiner surface; optional - surf
```

AUTHORS:

- David Joyner (2006-02-12)

primary_decomposition(*algorithm='sy'*)

Return a list of primary ideals such that their intersection is `self`.

An ideal Q is called primary if it is a proper ideal of the ring R, and if whenever $ab \in Q$ and $a \notin Q$, then $b^n \in Q$ for some $n \in \mathbf{Z}$.

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 $n \in \mathbf{Z}$), 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: 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(); pd
[Ideal (z^6 + 4*z^3 + 4, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational F
 Ideal (z^2 + 1, y + 1) of Multivariate Polynomial Ring in x, y, z over Rational Field]

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 Commutative Algebra. Springer, New York 1993.

primary_decomposition_complete(*algorithm='sy'*)

Return a list of primary ideals such that their intersection is `self`, together with the associated prime ideals.

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 $n \in \mathbf{Z}$.

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 $n \in \mathbf{Z}$), 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 P_i 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 pairs (Q_i, P_i), where the Q_i form a primary decomposition of `self` and P_i is the associated prime of Q_i.

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.complete_primary_decomposition(); pd
[(Ideal (z^6 + 4*z^3 + 4, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational
  Ideal (z^3 + 2, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field),
 (Ideal (z^2 + 1, y + 1) of Multivariate Polynomial Ring in x, y, z over Rational Field,
  Ideal (z^2 + 1, y + 1) of Multivariate Polynomial Ring in x, y, z over Rational Field)]
```

```
sage: I.primary_decomposition_complete(algorithm = 'gtz')
[(Ideal (z^6 + 4*z^3 + 4, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational
  Ideal (z^3 + 2, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field),
 (Ideal (z^2 + 1, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field,
  Ideal (z^2 + 1, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field)]

sage: from functools import reduce
sage: reduce(lambda Qi,Qj: Qi.intersection(Qj), [Qi for (Qi,radQi) in pd]) == I
True

sage: [Qi.radical() == radQi for (Qi,radQi) in pd]
[True, True]

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.complete_primary_decomposition()
Traceback (most recent call last):
...
ValueError: Coefficient ring must be a field for function 'complete_primary_decomposition'.
```

ALGORITHM:

Uses Singular.

Note: See [BW93] for an introduction to primary decomposition.

TESTS:

Check that trac ticket #15745 is fixed:

```
sage: R.<x,y>= QQ[]
sage: I = Ideal(R(1))
sage: I.complete_primary_decomposition()
[]
sage: I.is_prime()
False
```

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

EXAMPLE:

```
sage: R.<x,y,z> = PolynomialRing(GF(181),3)
sage: I = Ideal([x^2+x*y*z,y^2-z^3*y,z^3+y^5*x*z])
sage: J = Ideal([x])
sage: Q = I.quotient(J)
sage: y*z + x in I
False
sage: x in J
True
sage: x * (y*z + x) in I
True
```

TEST:

This example checks trac ticket #16301:

```
sage: R.<x,y,z> = ZZ[]
sage: I = Ideal(R(2), x*y, x*z + x)
sage: eD = Ideal(x, z^2-1)
sage: I.quotient(eD).gens()
[2, x*z + x, x*y]
```

radical()

The radical of this ideal.

EXAMPLES:

This is an obviously not radical ideal:

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: I = (x^2, y^3, (x*z)^4 + y^3 + 10*x^2)*R
sage: I.radical()
Ideal (y, x) of Multivariate Polynomial Ring in x, y, z over Rational Field
```

That the radical is correct is clear from the Groebner basis.

```
sage: I.groebner_basis()
[y^3, x^2]
```

This is the example from the Singular manual:

```
sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y-z^2)*R
sage: I.radical()
Ideal (z^2 - y, y^2*z + y*z + 2*y + 2) of Multivariate Polynomial Ring in x, y, z over Ratio
```

Note: From the Singular manual: A combination of the algorithms of Krick/Logar and Kemper is used. Works also in positive characteristic (Kempers algorithm).

```
sage: R.<x,y,z> = PolynomialRing(GF(37), 3)
sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y - z^2)*R
sage: I.radical()
Ideal (z^2 - y, y^2*z + y*z + 2*y + 2) of Multivariate Polynomial Ring in x, y, z over Finit
```

saturation(*other*)

Returns 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.

EXAMPLE:

```
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.

transformed_basis(*algorithm='gwalk'*, *other_ring=None*, *singular='singular_default'*)

Returns a lex or `other_ring` Groebner Basis for this ideal.

INPUT:

- `algorithm` - see below for options.
- `other_ring` - only valid for algorithm 'fglm', if provided conversion will be performed to this ring. Otherwise a lex Groebner basis will be returned.

ALGORITHMS:

- `fglm` - FGLM algorithm. The input ideal must be given with a reduced Groebner Basis of a zero-dimensional ideal
- `gwalk` - Groebner Walk algorithm (*default*)
- `awalk1` - 'first alternative' algorithm
- `awalk2` - 'second alternative' algorithm
- `twalk` - Tran algorithm
- `fwalk` - Fractal Walk algorithm

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: I = Ideal([y^3+x^2,x^2*y+x^2, x^3-x^2, z^4-x^2-y])
sage: I = 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')
sage: I=Ideal([y^3+x*y*z+y^2*z+x*z^3,3+x*y+x^2*y+y^2*z])
sage: I.transformed_basis('gwalk')
[z*y^2 + y*x^2 + y*x + 3,
 z*x + 8297*y^8*x^2 + 8297*y^8*x + 3556*y^7 - 8297*y^6*x^4 + 15409*y^6*x^3 - 8297*y^6*x^2
 - 8297*y^5*x^5 + 15409*y^5*x^4 - 8297*y^5*x^3 + 3556*y^5*x^2 + 3556*y^5*x + 3556*y^4*x^3
 + 3556*y^4*x^2 - 10668*y^4 - 10668*y^3*x - 8297*y^2*x^9 - 1185*y^2*x^8 + 14224*y^2*x^7
 - 1185*y^2*x^6 - 8297*y^2*x^5 - 14223*y*x^7 - 10666*y*x^6 - 10666*y*x^5 - 14223*y*x^4
 + x^5 + 2*x^4 + x^3,
 y^9 - y^7*x^2 - y^7*x - y^6*x^3 - y^6*x^2 - 3*y^6 - 3*y^5*x - y^3*x^7 - 3*y^3*x^6
 - 3*y^3*x^5 - y^3*x^4 - 9*y^2*x^5 - 18*y^2*x^4 - 9*y^2*x^3 - 27*y*x^3 - 27*y*x^2 - 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` - decomp. of self into tri. 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.

EXAMPLE:

```
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,
Ideal (a - 1, b - 1, c^2 + 3*c + 1, d + c + 3, e - 1) of Multivariate Polynomial Ring in e,
Ideal (a - 1, b^2 + 3*b + 1, c + b + 3, d - 1, e - 1) of Multivariate Polynomial Ring in e,
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 Multiva
Ideal (a^2 + 3*a + 1, b - 1, c - 1, d - 1, e + a + 3) of Multivariate Polynomial Ring in e,
Ideal (a^2 + 3*a + 1, b + a + 3, c - 1, d - 1, e - 1) of Multivariate Polynomial Ring in e,
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
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 Multiva
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 Multivariat
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 Multivariat
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 Multivariat
Ideal (a^4 + a^3 + a^2 + a + 1, b^3 + b^2*a + b^2 + b*a^2 + b*a + b + a^3 + a^2 + a + 1, c +
Ideal (a^4 + a^3 + 6*a^2 - 4*a + 1, -11*b^2 + 6*b*a^3 + 10*b*a^2 + 39*b*a + 2*b + 16*a^3 + 2

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
```

TESTS:

```
sage: R.<x,y> = QQ[]
sage: J = Ideal(x^2+y^2-2, y^2-1)
sage: J.triangular_decomposition()
[Ideal (y^2 - 1, x^2 - 1) of Multivariate Polynomial Ring in x, y over Rational Field]
```

variety(*ring=None*)

Return the variety of this ideal.

Given a zero-dimensional ideal I (== `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`)
- `proof` - return a provably correct result (default: `True`)

EXAMPLE:

```
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(); V
[{y: w^2 + 2, x: 2*w}, {y: w^2 + w, x: 2*w + 1}, {y: w^2 + 2*w, x: 2*w + 2}]

sage: [f.subs(v) for f in I.gens() for v in V] # check that all polynomials vanish
[0, 0, 0, 0, 0, 0]
sage: [I.subs(v).is_zero() for v in V] # same test, but nicer syntax
[True, True, True]
```

However, we only account for solutions in the ground field and not in the algebraic closure:

```
sage: I.vector_space_dimension()
48
```

Here we compute the points of intersection of a hyperbola and a circle, in several fields:

```
sage: K.<x, y> = PolynomialRing(QQ, 2, order='lex')
sage: I = Ideal([ x*y - 1, (x-2)^2 + (y-1)^2 - 1])
sage: I = Ideal(I.groebner_basis()); I
Ideal (x + y^3 - 2*y^2 + 4*y - 4, y^4 - 2*y^3 + 4*y^2 - 4*y + 1)
of Multivariate Polynomial Ring in x, y over Rational Field
```

These two curves have one rational intersection:

```
sage: I.variety()
[{y: 1, x: 1}]
```

There are two real intersections:

```
sage: I.variety(ring=RR)
[{y: 0.361103080528647, x: 2.76929235423863},
 {y: 1.00000000000000, x: 1.00000000000000}]
sage: I.variety(ring=AA)
[{x: 2.769292354238632?, y: 0.3611030805286474?},
 {x: 1, y: 1}]
```

and a total of four intersections:

```
sage: I.variety(ring=CC)
[{y: 0.31944845973567... - 1.6331702409152...*I,
  x: 0.11535382288068... + 0.58974280502220...*I},
 {y: 0.31944845973567... + 1.6331702409152...*I,
  x: 0.11535382288068... - 0.58974280502220...*I},
 {y: 0.36110308052864..., x: 2.7692923542386...},
 {y: 1.00000000000000, x: 1.00000000000000}]
sage: I.variety(ring=QQbar)
[{y: 0.3194484597356763? - 1.633170240915238?*I,
  x: 0.1153538228806843? + 0.5897428050222055?*I},
 {y: 0.3194484597356763? + 1.633170240915238?*I,
  x: 0.1153538228806843? - 0.5897428050222055?*I},
 {y: 0.3611030805286474?, x: 2.769292354238632?},
 {y: 1, x: 1}]
```

Computation over floating point numbers may compute only a partial solution, or even none at all. Notice that x values are missing from the following variety:

```
sage: R.<x,y> = CC[]
sage: I = ideal([x^2+y^2-1,x*y-1])
sage: I.variety()
verbose 0 (...: multi_polynomial_ideal.py, variety) Warning: computations in the complex fie
verbose 0 (...: multi_polynomial_ideal.py, variety) Warning: falling back to very slow toy i
[{y: -0.86602540378443... - 0.500000000000000*I},
 {y: -0.86602540378443... + 0.500000000000000*I},
 {y: 0.86602540378443... - 0.500000000000000*I},
 {y: 0.86602540378443... + 0.500000000000000*I}]
```

This is due to precision error, which causes the computation of an intermediate Groebner basis to fail.

If the ground field's characteristic is too large for Singular, we resort to a toy implementation:

```
sage: R.<x,y> = PolynomialRing(GF(2147483659),order='lex')
sage: I=ideal([x^3-2*y^2,3*x+y^4])
sage: I.variety()
verbose 0 (...: multi_polynomial_ideal.py, groebner_basis) Warning: falling back to very slo
verbose 0 (...: multi_polynomial_ideal.py, dimension) Warning: falling back to very slow toy
verbose 0 (...: multi_polynomial_ideal.py, variety) Warning: falling back to very slow toy i
```

```
[{y: 0, x: 0}]
```

TESTS:

```
sage: K.<w> = GF(27)
sage: P.<x, y> = PolynomialRing(K, 2, order='lex')
sage: I = Ideal([ x^8 + y + 2, y^6 + x*y^5 + x^2 ])
```

Testing the robustness of the Singular interface:

```
sage: T = I.triangular_decomposition('singular:triangLfak')
sage: I.variety()
[{y: w^2 + 2, x: 2*w}, {y: w^2 + w, x: 2*w + 1}, {y: w^2 + 2*w, x: 2*w + 2}]
```

Testing that a bug is indeed fixed

```
sage: R = PolynomialRing(GF(2), 30, ['x%d'%(i+1) for i in range(30)], order='lex')
sage: R.inject_variables()
Defining...
sage: I = Ideal([x1 + 1, x2, x3 + 1, x5*x10 + x10 + x18, x5*x11 + x11, \
                 x5*x18, x6, x7 + 1, x9, x10*x11 + x10 + x18, x10*x18 + x18, \
                 x11*x18, x12, x13, x14, x15, x16 + 1, x17 + x18 + 1, x19, x20, \
                 x21 + 1, x22, x23, x24, x25 + 1, x28 + 1, x29 + 1, x30, x8, \
                 x26, x1^2 + x1, x2^2 + x2, x3^2 + x3, x4^2 + x4, x5^2 + x5, \
                 x6^2 + x6, x7^2 + x7, x8^2 + x8, x9^2 + x9, x10^2 + x10, \
                 x11^2 + x11, x12^2 + x12, x13^2 + x13, x14^2 + x14, x15^2 + x15, \
                 x16^2 + x16, x17^2 + x17, x18^2 + x18, x19^2 + x19, x20^2 + x20, \
                 x21^2 + x21, x22^2 + x22, x23^2 + x23, x24^2 + x24, x25^2 + x25, \
                 x26^2 + x26, x27^2 + x27, x28^2 + x28, x29^2 + x29, x30^2 + x30])
sage: I.basis_is_groebner()
True
sage: for V in I.variety(): print V  # long time (6s on sage.math, 2011)
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 0, x19: 0, x18: 1, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 1, x19: 0, x18: 1, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 0, x19: 0, x18: 1, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 1, x19: 0, x18: 1, x7: 1, x6: 0, x1
```

Check that the issue at trac ticket #7425 is fixed:

```
sage: R.<x, y, z> = QQ[]
sage: I = R.ideal([x^2-y^3*z, x+y*z])
sage: I.dimension()
1
sage: I.variety()
```

```
Traceback (most recent call last):
...
ValueError: The dimension of the ideal is 1, but it should be 0
```

Check that the issue at trac ticket #7425 is fixed:

```
sage: S.<t>=PolynomialRing(QQ)
sage: F.<q>=QQ.extension(t^4+1)
sage: R.<x,y>=PolynomialRing(F)
sage: I=R.ideal(x,y^4+1)
sage: I.variety()
[...{y: -q^3, x: 0}...]
```

Check that computing the variety of the `[1]` ideal is allowed (trac ticket #13977):

```
sage: R.<x,y> = QQ[]
sage: I = R.ideal(1)
sage: I.variety()
[]
```

Check that the issue at trac ticket #16485 is fixed:

```
sage: R.<a,b,c> = PolynomialRing(QQ, order='lex')
sage: I = R.ideal(c^2-2, b-c, a)
sage: I.variety(QQbar)
[...a: 0...]
```

ALGORITHM:

Uses triangular decomposition.

vector_space_dimension()

Return the vector space dimension of the ring modulo this ideal. If the ideal is not zero-dimensional, a TypeError is raised.

ALGORITHM:

Uses Singular.

EXAMPLE:

```
sage: R.<u,v> = PolynomialRing(QQ)
sage: g = u^4 + v^4 + u^3 + v^3
sage: I = ideal(g) + ideal(g.gradient())
sage: I.dimension()
0
sage: I.vector_space_dimension()
4
```

When the ideal is not zero-dimensional, we return infinity:

```
sage: R.<x,y> = PolynomialRing(QQ)
sage: I = R.ideal(x)
sage: I.dimension()
1
sage: I.vector_space_dimension()
+Infinity
```

class sage.rings.polynomial.multi_polynomial_ideal.**NCPolynomialIdeal**(*ring*, *gens*, *coerce=True*, *side='left'*)

Bases: sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr, sage.rings.noncommutative_ideals.Ideal_nc

Creates a non-commutative polynomial ideal.

INPUT:

- ring - the g-algebra to which this ideal belongs
- gens - the generators of this ideal
- coerce (optional - default True) - generators are coerced into the ring before creating the ideal
- side - optional string, either "left" (default) or "twosided"; defines whether this ideal is left of two-sided.

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: H.inject_variables()
Defining x, y, z
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False) # indirect doctest
sage: I
Left Ideal (y^2, x^2, z^2 - 1) of Noncommutative Multivariate Polynomial Ring in x, y, z over Ra
sage: H.ideal([y^2, x^2, z^2-H.one()], side="twosided")
Twosided Ideal (y^2, x^2, z^2 - 1) of Noncommutative Multivariate Polynomial Ring in x, y, z ove
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.
```

reduce (*p*)

Reduce an element modulo a Groebner basis for this ideal.

It returns 0 if and only if the element is in this ideal. In any case, this reduction is unique up to monomial orders.

NOTE:

There are left and two-sided ideals. Hence,

EXAMPLE:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False, side='twosided')
sage: Q = H.quotient(I); Q
Quotient of Noncommutative Multivariate Polynomial Ring in x, y, z
 over Rational Field, nc-relations: {z*x: x*z + 2*x,
 z*y: y*z - 2*y, y*x: x*y - z} by the ideal (y^2, x^2, z^2 - 1)
sage: Q.2^2 == Q.one()    # indirect doctest
True
```

Here, we see that the relation that we just found in the quotient is actually a consequence of the given relations:

```
sage: I.std()
Twosided Ideal (z^2 - 1, y*z - y, x*z + x, y^2, 2*x*y - z - 1, x^2)
of Noncommutative Multivariate Polynomial Ring in x, y, z over
Rational Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z}
```

Here is the corresponding direct test:

```
sage: I.reduce(z^2)
1
```

res(*length*)

Computes the resoltuion 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.

EXAMPLE:

```
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()

Computes a GB of the ideal. It is two-sided if and only if the ideal is two-sided.

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: H.inject_variables()
Defining x, y, z
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False)
sage: I.std()
Left Ideal (z^2 - 1, y*z - y, x*z + x, y^2, 2*x*y - z - 1, x^2) of Noncommutative Multivaria
```

If the ideal is a left ideal, then std returns a left Groebner basis. But if it is a two-sided ideal, then the output of std and `twostd()` coincide:

```
sage: JL = H.ideal([x^3, y^3, z^3 - 4*z])
sage: JL
Left Ideal (x^3, y^3, z^3 - 4*z) of Noncommutative Multivariate Polynomial Ring in x, y, z o
sage: JL.std()
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 Nonco
sage: JT = H.ideal([x^3, y^3, z^3 - 4*z], side='twosided')
sage: JT
Twosided Ideal (x^3, y^3, z^3 - 4*z) of Noncommutative Multivariate Polynomial Ring in x, y,
sage: JT.std()
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,
sage: JT.std() == JL.twostd()
True
```

ALGORITHM: Uses Singular's std command

syzygy_module()

Computes the first syzygy (i.e., the module of relations of the given generators) of the ideal.

NOTE:

Only left syzygies can be computed. So, even if the ideal is two-sided, then the syzygies are only one-sided. In that case, a warning is printed.

EXAMPLE:

```
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: G = vector(I.gens()); G
d...: UserWarning: You are constructing a free module
over a noncommutative ring. Sage does not have a concept
of left/right and both sided modules, so be careful.
It's also not guaranteed that all multiplications are
done from the right side.
d...: UserWarning: You are constructing a free module
over a noncommutative ring. Sage does not have a concept
of left/right and both sided modules, so be careful.
It's also not guaranteed that all multiplications are
done from the right side.
(y^2, x^2, z^2 - 1)
sage: M = I.syzygy_module(); M
[                                                                             -z^2 - 8*z - 15
[                                                                                           0
[                                                                  x^2*z^2 + 8*x^2*z + 15*x^2
[                 x^2*y*z^2 + 9*x^2*y*z - 6*x*z^3 + 20*x^2*y - 72*x*z^2 - 282*x*z - 360*x
[                                                                  x^3*z^2 + 7*x^3*z + 12*x^3
[  x^2*y^2*z + 4*x^2*y^2 - 8*x*y*z^2 - 48*x*y*z + 12*z^3 - 64*x*y + 108*z^2 + 312*z + 288
[                                                      2*x^3*y*z + 8*x^3*y + 9*x^2*z + 27*x^2
[                                                                                   x^4*z + 4*x^4
[x^3*y^2*z + 4*x^3*y^2 + 18*x^2*y*z - 36*x*z^3 + 66*x^2*y - 432*x*z^2 - 1656*x*z - 2052*x

sage: M*G
(0, 0, 0, 0, 0, 0, 0, 0, 0)
```

ALGORITHM: Uses Singular's syz command

twostd()

Computes a two-sided GB of the ideal (even if it is a left ideal).

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: H.inject_variables()
Defining x, y, z
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False)
sage: I.twostd()
Twosided Ideal (z^2 - 1, y*z - y, x*z + x, y^2, 2*x*y - z - 1, x^2) of Noncommutative Multiv
```

ALGORITHM: Uses Singular's twostd command

class sage.rings.polynomial.multi_polynomial_ideal.**RequireField**(*f*)

Bases: sage.misc.method_decorator.MethodDecorator

Decorator which throws an exception if a computation over a coefficient ring which is not a field is attempted.

Note: This decorator is used automatically internally so the user does not need to use it manually.

sage.rings.polynomial.multi_polynomial_ideal.**is_MPolynomialIdeal**(*x*)

Return True if the provided argument x is an ideal in the multivariate polynomial ring.

INPUT:

•x - an arbitrary object

EXAMPLES:

```
sage: from sage.rings.polynomial.multi_polynomial_ideal import is_MPolynomialIdeal
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = [x + 2*y + 2*z - 1, x^2 + 2*y^2 + 2*z^2 - x, 2*x*y + 2*y*z - y]
```

Sage distinguishes between a list of generators for an ideal and the ideal itself. This distinction is inconsistent with Singular but matches Magma's behavior.

```
sage: is_MPolynomialIdeal(I)
False

sage: I = Ideal(I)
sage: is_MPolynomialIdeal(I)
True
```

`sage.rings.polynomial.multi_polynomial_ideal.`**`require_field`**

alias of `RequireField`

3.7 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 [CMR05] (cf. `sage.crypto.mq.sr.SR()`) and Flurry/Curry [BPW06]. 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)
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: F,s = sr.polynomial_system()
sage: F
Polynomial Sequence with 112 Polynomials in 64 Variables

sage: r2 = F.part(2); r2
(w200 + k100 + x100 + x102 + x103,
 w201 + k101 + x100 + x101 + x103 + 1,
 w202 + k102 + x100 + x101 + x102 + 1,
 w203 + k103 + x101 + x102 + x103,
 w210 + k110 + x110 + x112 + x113,
 w211 + k111 + x110 + x111 + x113 + 1,
 w212 + k112 + x110 + x111 + x112 + 1,
 w213 + k113 + x111 + x112 + x113,
 x100*w100 + x100*w103 + x101*w102 + x102*w101 + x103*w100,
 x100*w100 + x100*w101 + x101*w100 + x101*w103 + x102*w102 + x103*w101,
 x100*w101 + x100*w102 + x101*w100 + x101*w101 + x102*w100 + x102*w103 + x103*w102,
 x100*w100 + x100*w102 + x100*w103 + x101*w100 + x101*w101 + x102*w102 + x103*w100 + x100,
 x100*w101 + x100*w103 + x101*w101 + x101*w102 + x102*w100 + x102*w103 + x103*w101 + x101,
 x100*w100 + x100*w102 + x101*w100 + x101*w102 + x101*w103 + x102*w100 + x102*w101 + x103*w102 + x102
 x100*w101 + x100*w102 + x101*w100 + x101*w103 + x102*w101 + x103*w103 + x103,
 x100*w100 + x100*w101 + x100*w103 + x101*w101 + x102*w100 + x102*w102 + x103*w100 + w100,
 x100*w102 + x101*w100 + x101*w101 + x101*w103 + x102*w101 + x103*w100 + x103*w102 + w101,
 x100*w100 + x100*w101 + x100*w102 + x101*w102 + x102*w100 + x102*w101 + x102*w103 + x103*w101 + w102
 x100*w101 + x101*w100 + x101*w102 + x102*w100 + x103*w101 + x103*w103 + w103,
 x100*w102 + x101*w101 + x102*w100 + x103*w103 + 1,
 x110*w110 + x110*w113 + x111*w112 + x112*w111 + x113*w110,
 x110*w110 + x110*w111 + x111*w110 + x111*w113 + x112*w112 + x113*w111,
 x110*w111 + x110*w112 + x111*w110 + x111*w111 + x112*w110 + x112*w113 + x113*w112,
 x110*w110 + x110*w112 + x110*w113 + x111*w110 + x111*w111 + x112*w112 + x113*w110 + x110,
 x110*w111 + x110*w113 + x111*w111 + x111*w112 + x112*w110 + x112*w113 + x113*w111 + x111,
 x110*w110 + x110*w112 + x111*w110 + x111*w112 + x111*w113 + x112*w110 + x112*w111 + x113*w112 + x112
 x110*w111 + x110*w112 + x111*w110 + x111*w113 + x112*w111 + x113*w113 + x113,
 x110*w110 + x110*w111 + x110*w113 + x111*w111 + x112*w110 + x112*w112 + x113*w110 + w110,
 x110*w112 + x111*w110 + x111*w111 + x111*w113 + x112*w111 + x113*w110 + x113*w112 + w111,
 x110*w110 + x110*w111 + x110*w112 + x111*w112 + x112*w110 + x112*w111 + x112*w113 + x113*w111 + w112
 x110*w111 + x111*w110 + x111*w112 + x112*w110 + x113*w111 + x113*w113 + w113,
 x110*w112 + x111*w111 + x112*w110 + x113*w113 + 1)
```

We separate the system in independent subsystems:

```
sage: C = Sequence(r2).connected_components(); C
[[w213 + k113 + x111 + x112 + x113,
 w212 + k112 + x110 + x111 + x112 + 1,
 w211 + k111 + x110 + x111 + x113 + 1,
 w210 + k110 + x110 + x112 + x113,
 x110*w112 + x111*w111 + x112*w110 + x113*w113 + 1,
 x110*w112 + x111*w110 + x111*w111 + x111*w113 + x112*w111 + x113*w110 + x113*w112 + w111,
 x110*w111 + x111*w110 + x111*w112 + x112*w110 + x113*w111 + x113*w113 + w113,
 x110*w111 + x110*w113 + x111*w111 + x111*w112 + x112*w110 + x112*w113 + x113*w111 + x111,
 x110*w111 + x110*w112 + x111*w110 + x111*w113 + x112*w111 + x113*w113 + x113,
 x110*w111 + x110*w112 + x111*w110 + x111*w111 + x112*w110 + x112*w113 + x113*w112,
 x110*w110 + x110*w113 + x111*w112 + x112*w111 + x113*w110,
 x110*w110 + x110*w112 + x111*w110 + x111*w112 + x111*w113 + x112*w110 + x112*w111 + x113*w112 + x112
 x110*w110 + x110*w112 + x110*w113 + x111*w110 + x111*w111 + x112*w112 + x113*w110 + x110,
 x110*w110 + x110*w111 + x111*w110 + x111*w113 + x112*w112 + x113*w111,
 x110*w110 + x110*w111 + x110*w113 + x111*w111 + x112*w110 + x112*w112 + x113*w110 + w110,
 x110*w110 + x110*w111 + x110*w112 + x111*w112 + x112*w110 + x112*w111 + x112*w113 + x113*w111 + w112
[w203 + k103 + x101 + x102 + x103,
w202 + k102 + x100 + x101 + x102 + 1,
w201 + k101 + x100 + x101 + x103 + 1,
w200 + k100 + x100 + x102 + x103,
x100*w102 + x101*w101 + x102*w100 + x103*w103 + 1,
x100*w102 + x101*w100 + x101*w101 + x101*w103 + x102*w101 + x103*w100 + x103*w102 + w101,
x100*w101 + x101*w100 + x101*w102 + x102*w100 + x103*w101 + x103*w103 + w103,
x100*w101 + x100*w103 + x101*w101 + x101*w102 + x102*w100 + x102*w103 + x103*w101 + x101,
x100*w101 + x100*w102 + x101*w100 + x101*w103 + x102*w101 + x103*w103 + x103, x100*w101 + x100*w102 
x100*w100 + x100*w103 + x101*w102 + x102*w101 + x103*w100,
x100*w100 + x100*w102 + x101*w100 + x101*w102 + x101*w103 + x102*w100 + x102*w101 + x103*w102 + x102,
x100*w100 + x100*w102 + x100*w103 + x101*w100 + x101*w101 + x102*w102 + x103*w100 + x100,
x100*w100 + x100*w101 + x101*w100 + x101*w103 + x102*w102 + x103*w101,
x100*w100 + x100*w101 + x100*w103 + x101*w101 + x102*w100 + x102*w102 + x103*w100 + w100,
x100*w100 + x100*w101 + x100*w102 + x101*w102 + x102*w100 + x102*w101 + x102*w103 + x103*w101 + w102
sage: C[0].groebner_basis()
Polynomial Sequence with 30 Polynomials in 16 Variables
```

and compute the coefficient matrix:

```
sage: A,v = Sequence(r2).coefficient_matrix()
sage: A.rank()
32
```

Using these building blocks we can implement a simple XL algorithm easily:

```
sage: sr = mq.SR(1,1,1,4, gf2=True, polybori=True, order='lex')
sage: F,s = sr.polynomial_system()

sage: monomials = [a*b for a in F.variables() for b in F.variables() if a<b]
sage: len(monomials)
190
sage: F2 = Sequence(map(mul, cartesian_product_iterator((monomials, F))))
sage: A,v = F2.coefficient_matrix(sparse=False)
sage: A.echelonize()
sage: A
6840 x 4474 dense matrix over Finite Field of size 2 (use the '.str()' method to see the entries)
sage: A.rank()
4056
sage: A[4055]*v
(k001*k003)
```

TEST:

```
sage: P.<x,y> = PolynomialRing(QQ)
sage: I = [[x^2 + y^2], [x^2 - y^2]]
sage: F = Sequence(I, P)
sage: loads(dumps(F)) == F
True
```

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.

3.7.1 Classes

`sage.rings.polynomial.multi_polynomial_sequence.`**`PolynomialSequence`**(*arg1*, *arg2=None*, *immutable=False*, *cr=False*, *cr_str=None*)

Construct a new polynomial sequence object.

INPUT:

- arg1 - a multivariate polynomial ring, an ideal or a matrix
- arg2 - an iterable object of parts or polynomials (default:None)
  - immutable - if True the sequence is immutable (default: False)
  - cr - print a line break after each element (default: False)
  - cr_str - print a line break after each element if 'str' is called (default: None)

EXAMPLES:

```
sage: P.<a,b,c,d> = PolynomialRing(GF(127),4)
sage: I = sage.rings.ideal.Katsura(P)
```

If a list of tuples is provided, those form the parts:

```
sage: F = Sequence([I.gens(),I.gens()], I.ring()); F # indirect doctest
[a + 2*b + 2*c + 2*d - 1,
 a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a,
 2*a*b + 2*b*c + 2*c*d - b,
 b^2 + 2*a*c + 2*b*d - c,
 a + 2*b + 2*c + 2*d - 1,
 a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a,
 2*a*b + 2*b*c + 2*c*d - b,
 b^2 + 2*a*c + 2*b*d - c]
sage: F.nparts()
2
```

If an ideal is provided, the generators are used:

```
sage: Sequence(I)
[a + 2*b + 2*c + 2*d - 1,
 a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a,
 2*a*b + 2*b*c + 2*c*d - b,
 b^2 + 2*a*c + 2*b*d - c]
```

If a list of polynomials is provided, the system has only one part:

```
sage: F = Sequence(I.gens(), I.ring()); F
[a + 2*b + 2*c + 2*d - 1,
 a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a,
 2*a*b + 2*b*c + 2*c*d - b,
 b^2 + 2*a*c + 2*b*d - c]
 sage: F.nparts()
 1
```

We test that the ring is inferred correctly:

```
sage: P.<x,y,z> = GF(2)[]
sage: from sage.rings.polynomial.multi_polynomial_sequence import PolynomialSequence
sage: PolynomialSequence([1,x,y]).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: `sage.structure.sequence.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:

```
sage: P.<a,b,c,d> = PolynomialRing(GF(127),4)
sage: I = sage.rings.ideal.Katsura(P)

sage: Sequence([I.gens()], I.ring()) # indirect doctest
[a + 2*b + 2*c + 2*d - 1, a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a, 2*a*b + 2*b*c + 2*c*d - b, b^2 + 2*a*
```

If an ideal is provided, the generators are used.:

```
sage: Sequence(I)
[a + 2*b + 2*c + 2*d - 1, a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a, 2*a*b + 2*b*c + 2*c*d - b, b^2 + 2*a*
```

If a list of polynomials is provided, the system has only one part.:

```
sage: Sequence(I.gens(), I.ring())
[a + 2*b + 2*c + 2*d - 1, a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a, 2*a*b + 2*b*c + 2*c*d - b, b^2 + 2*a*
```

`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:

```
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

sage: R.<x,y> = PolynomialRing(QQ)
sage: S = Sequence([x, (x^2 + y^2 - 1)^2, x*y - 2])
sage: I = S.algebraic_dependence(); I
Ideal (16 + 32*T2 - 8*T0^2 + 24*T2^2 - 8*T0^2*T2 + 8*T2^3 + 9*T0^4 - 2*T0^2*T2^2 + T2^4 - T0
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
Ideal (2 - 3*T2 - T0^2 + 3*T2^2 - T0^2*T2 + T2^3 + 2*T0^4 - 2*T0^2*T2^2 + T2^4 - T0^4*T1 + T
sage: [F(S) for F in I.gens()]
[0]
```

Note: This function's code also works for sequences of polynomials from a univariate polynomial ring, but i don't know where in the Sage codebase to put it to use it to that effect.

AUTHORS:

- Alex Raichev (2011-06-22)

coefficient_matrix(*sparse=True*)

Return tuple `(A, v)` where `A` is the coefficient matrix of this system and `v` the matching monomial vector.

Thus value of `A[i, j]` corresponds the coefficient of the monomial `v[j]` in the `i`-th polynomial in this system.

Monomials are order w.r.t. the term ordering of `self.ring()` in reverse order, i.e. such that the smallest entry comes last.

INPUT:

- `sparse` - construct a sparse matrix (default: `True`)

EXAMPLE:

```
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*c + 2*b*d - c]

sage: F = Sequence(I)
sage: A,v = F.coefficient_matrix()
sage: A
[  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]
[b*d]
[c*d]
[d^2]
[  a]
[  b]
[  c]
[  d]
[  1]

sage: A*v
[         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]
```

connected_components()

Split the polynomial system in systems which do not share any variables.

EXAMPLE:

As an example consider one part of AES, which naturally splits into four subsystems which are independent:

```
sage: sr = mq.SR(2,4,4,8,gf2=True,polybori=True)
sage: F,s = sr.polynomial_system()
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.

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing()
sage: F = Sequence([x*y + y + 1, z + 1])
sage: F.connection_graph()
Graph on 3 vertices
```

groebner_basis(**args*, ***kwargs*)

Compute and return a Groebner basis for the ideal spanned by the polynomials in this system.

INPUT:

•`args` - list of arguments passed to `MPolynomialIdeal.groebner_basis` call

•`kwargs` - dictionary of arguments passed to `MPolynomialIdeal.groebner_basis` call

EXAMPLE:

```
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.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: P = F.ring()
sage: I = F.ideal()
```

```
sage: I.elimination_ideal(P('s000*s001*s002*s003*w100*w101*w102*w103*x100*x101*x102*x103'))
Ideal (k002 + (a^3 + a + 1)*k003 + (a^2 + 1),
       k001 + (a^3)*k003, k000 + (a)*k003 + (a^2),
       k103 + k003 + (a^2 + a + 1),
       k102 + (a^3 + a + 1)*k003 + (a + 1),
       k101 + (a^3)*k003 + (a^2 + a + 1),
       k100 + (a)*k003 + (a),
       k003^2 + (a)*k003 + (a^2))
of Multivariate Polynomial Ring in k100, k101, k102, k103, x100, x101, x102, x103,
w100, w101, w102, w103, s000, s001, s002, s003, k000, k001, k002, k003 over Finite Field in
```

is_groebner(*singular=Singular*)

Returns `True` if the generators of this ideal (`self.gens()`) form a Grbner 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 $Syz(LM(I))$:

$$S * G = \sum_{i=0}^{m} h_i g_i - - - - >_G 0.$$

EXAMPLE:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: len(F.monomials())
49
```

nmonomials()

Return the number of monomials present in this system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: F.nmonomials()
49
```

nparts()

Return number of parts of this system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: F.nparts()
4
```

nvariables()

Return number of variables present in this system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: F.nvariables()
20
```

part(*i*)

Return `i`-th part of this system.

EXAMPLE:

```
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.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: l = F.parts()
sage: len(l)
4
```

reduced()

If this sequence is $(f_1, ..., f_n)$ then this method returns $(g_1, ..., g_s)$ such that:

- $(f_1, ..., f_n) = (g_1, ..., g_s)$
- $LT(g_i)! = LT(g_j)$ for all $i! = j$
- $LT(g_i)$ **does not divide** m **for all monomials** m **of** $\{g_1, ..., g_{i-1}, g_{i+1}, ..., g_s\}$
- $LC(g_i) == 1$ for all i if the coefficient ring is a field.

EXAMPLE:

```
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: F = Sequence([z*x+y^3,z+y^3,z+x*y])
sage: F.reduced()
[y^3 + z, x*y + z, x*z - z]
```

Note that tail reduction for local orderings is not well-defined:

```
sage: R.<x,y,z> = PolynomialRing(QQ,order='negdegrevlex')
sage: F = Sequence([z*x+y^3,z+y^3,z+x*y])
sage: F.reduced()
[z + x*y, x*y - y^3, x^2*y - y^3]
```

A fixed error with nonstandard base fields:

```
sage: R.<t>=QQ['t']
sage: K.<x,y>=R.fraction_field()['x,y']
sage: I=t*x*K
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.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True,gf2=True,order='block')
sage: F,s = sr.polynomial_system()
sage: print F.ring().repr_long()
Polynomial Ring
 Base Ring : Finite Field of size 2
      Size : 20 Variables
  Block  0 : Ordering : deglex
             Names    : k100, k101, k102, k103, x100, x101, x102, x103, w100, w101, w102, w1
  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`

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system(); F
Polynomial Sequence with 40 Polynomials in 20 Variables
sage: F = F.subs(s); F
Polynomial Sequence with 40 Polynomials in 16 Variables
```

universe()

Return the polynomial ring all elements live in.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True,gf2=True,order='block')
sage: F,s = sr.polynomial_system()
sage: print F.ring().repr_long()
Polynomial Ring
 Base Ring : Finite Field of size 2
      Size : 20 Variables
  Block  0 : Ordering : deglex
             Names    : k100, k101, k102, k103, x100, x101, x102, x103, w100, w101, w102, w1
  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.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: F.variables()[:10]
(k003, k002, k001, k000, s003, s002, s001, s000, w103, w102)
```

class `sage.rings.polynomial.multi_polynomial_sequence.`**`PolynomialSequence_gf2`**(*parts, ring, immutable=False, cr=False, cr_str=None*)

Bases: `sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic`

Polynomial Sequences over $\mathbb{F}_2$.

eliminate_linear_variables(*maxlength=+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 `maxlength`.

INPUT:

- `maxlength` - an optional upper bound on the number of monomials by which a variable is replaced. If `maxlength==+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 what is implemented here, it is less flexible (skip` is not supported) and may increase the degree (default: ``False)

OUTPUT:

When `return_reductors==True`, then a pair of sequences of boolean polynomials are returned, along with the promises that:

1. The union of the two sequences spans the same boolean ideal as the argument of the method
2. The second sequence only contains linear polynomials, and it forms a reduced groebner basis (they all have pairwise distinct leading variables, and the leading variable of a polynomial does not occur anywhere in other polynomials).
3. The leading variables of the second sequence do not occur anywhere in the first sequence (these variables have been eliminated).

When `return_reductors==False`, only the first sequence is returned.

EXAMPLE:

```
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: 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:

```
sage: R.<x,y,z> = BooleanPolynomialRing()
sage: S = Sequence([x*y*z+x*y+z*y+x*z, x+y+z+1, x+y+z])
sage: S.eliminate_linear_variables()
[1]

sage: R.<x,y,z> = BooleanPolynomialRing()
sage: S = Sequence([x*y*z+x*y+z*y+x*z, x+y+z+1, x+y+z])
sage: S.eliminate_linear_variables(return_reductors=True)
([1], [])
```

TESTS:

The function should really dispose of linear equations (trac ticket #13968):

```
sage: R.<x,y,z> = BooleanPolynomialRing()
sage: S = Sequence([x+y+z+1, y+z])
sage: S.eliminate_linear_variables(return_reductors=True)
([], [x + 1, y + z])
```

The function should take care of linear variables created by previous substitution of linear variables

```
sage: R.<x,y,z> = BooleanPolynomialRing()
sage: S = Sequence([x*y*z+x*y+z*y+x*z, x+y+z+1, x+y])
sage: S.eliminate_linear_variables(return_reductors=True)
([], [x + y, z + 1])
```

We test a case which would increase the degree with `polybori=True`:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: f = a*d + a + b*d + c*d + 1
sage: Sequence([f, a + b*c + c+d + 1]).eliminate_linear_variables()
[a*d + a + b*d + c*d + 1, a + b*c + c + d + 1]

sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: f = a*d + a + b*d + c*d + 1
sage: Sequence([f, a + b*c + c+d + 1]).eliminate_linear_variables(use_polybori=True)
[b*c*d + b*c + b*d + c + d]
```

Note: This is called "massaging" in [CBJ07].

REFERENCES:

reduced()

If this sequence is $(f_1, ..., f_n)$ this method returns $(g_1, ..., g_s)$ such that:

- $< f_1, ..., f_n > = < g_1, ..., g_s >$
- $LT(g_i)! = LT(g_j)$ for all $i! = j$'
- $LT(g_i)$ does not divide m for all monomials m of $g_1, ..., g_{i-1}, g_{i+1}, ..., g_s$

EXAMPLE:

```
sage: sr = mq.SR(1, 1, 1, 4, gf2=True, polybori=True)
sage: F,s = sr.polynomial_system()
sage: F.reduced()
[k100 + 1, k101 + k001 + 1, k102, k103 + 1, ..., s002, s003 + k001 + 1, k000 + 1, k002 + 1,
```

solve(*algorithm='polybori'*, *n=1*, *eliminate_linear_variables=True*, *verbose=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:

- `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 $n < \infty$ then n solutions are returned if the equations have at least n solutions. Otherwise, all the solutions are returned. (default: `1`)
- `eliminate_linear_variables` - whether to eliminate variables that appear linearly. This reduces the number of variables (makes solving faster a priori), but is likely to make the equations denser (may make solving slower depending on the method).
- `verbose` - whether to display progress and (potentially) useful information while the computation runs. (default: `False`)

EXAMPLES:

Without argument, a single arbitrary solution is returned:

```
sage: R.<x,y,z> = BooleanPolynomialRing()
sage: S = Sequence([x*y+z, y*z+x, x+y+z+1])
sage: sol = S.solve(); sol                      # random
[{y: 1, z: 0, x: 0}]
```

We check that it is actually a solution:

```
sage: S.subs( sol[0] )
[0, 0, 0]
```

We obtain all solutions:

```
sage: sols = S.solve(n=Infinity); sols          # random
[{x: 0, y: 1, z: 0}, {x: 1, y: 1, z: 1}]
sage: map( lambda x: S.subs(x), sols)
[[0, 0, 0], [0, 0, 0]]
```

We can force the use of exhaustive search if the optional package `FES` is present:

```
sage: sol = S.solve(algorithm='exhaustive_search'); sol  # random, optional - FES
[{x: 1, y: 1, z: 1}]
sage: S.subs( sol[0] )
[0, 0, 0]
```

And we may use SAT-solvers if they are available:

```
sage: sol = S.solve(algorithm='sat'); sol                    # random, optional - CryptoMin
[{y: 1, z: 0, x: 0}]
sage: S.subs( sol[0] )
[0, 0, 0]
```

TESTS:

Make sure that variables not occuring in the equations are no problem:

```
sage: R.<x,y,z,t> = BooleanPolynomialRing()
sage: S = Sequence([x*y+z, y*z+x, x+y+z+1])
sage: sols = S.solve(n=Infinity)
sage: map( lambda x: S.subs(x), sols)
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Not eliminating linear variables:

```
sage: sols = S.solve(n=Infinity, eliminate_linear_variables=False)
sage: map( lambda x: S.subs(x), sols)
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

A tricky case where the linear equations are insatisfiable:

```
sage: R.<x,y,z> = BooleanPolynomialRing()
sage: S = Sequence([x*y*z+x*y+z*y+x*z, x+y+z+1, x+y+z])
sage: S.solve()
[]
```

class `sage.rings.polynomial.multi_polynomial_sequence.`**`PolynomialSequence_gf2e`**(*parts*, *ring*, *immutable=False*, *cr=False*, *cr_str=None*)

Bases: `sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic`

PolynomialSequence over $\mathbb{F}_{2^e}$, i.e extensions over GF(2).

`weil_restriction`()

Project this polynomial system to $\mathbb{F}_2$.

That is, compute the Weil restriction of scalars for the variety corresponding to this polynomial system and express it as a polynomial system over $\mathbb{F}_2$.

EXAMPLE:

```
sage: k.<a> = GF(2^2)
sage: P.<x,y> = PolynomialRing(k,2)
sage: a = P.base_ring().gen()
sage: F = Sequence([x*y + 1, a*x + 1], P)
sage: F2 = F.weil_restriction()
sage: F2
[x0*y0 + x1*y1 + 1, x1*y0 + x0*y1 + x1*y1, x1 + 1, x0 + x1, x0^2 + x0,
x1^2 + x1, y0^2 + y0, y1^2 + y1]
```

Another bigger example for a small scale AES:

```
sage: sr = mq.SR(1,1,1,4,gf2=False)
sage: F,s = sr.polynomial_system(); F
Polynomial Sequence with 40 Polynomials in 20 Variables
sage: F2 = F.weil_restriction(); F2
Polynomial Sequence with 240 Polynomials in 80 Variables
```

`sage.rings.polynomial.multi_polynomial_sequence.`**`is_PolynomialSequence`**(*F*)

Return `True` if `F` is a `PolynomialSequence`.

INPUT:

- `F` - anything

EXAMPLE:

```
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]

sage: from sage.rings.polynomial.multi_polynomial_sequence import is_PolynomialSequence
sage: is_PolynomialSequence(F)
True
```

3.8 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 **Q**,

- the ring of integers $\mathbf{Z}$,
- $\mathbf{Z}/n\mathbf{Z}$ for any integer n,
- finite fields $\mathbf{F}_{p^n}$ for p prime and $n > 0$,
- and absolute number fields $\mathbf{Q}(a)$.

AUTHORS:

The libSINGULAR interface was implemented by

- Martin Albrecht (2007-01): initial implementation
- Joel Mohler (2008-01): misc improvements, polishing
- Martin Albrecht (2008-08): added $\mathbf{Q}(a)$ and $\mathbf{Z}$ support
- Simon King (2009-04): improved coercion
- Martin Albrecht (2009-05): added $\mathbf{Z}/n\mathbf{Z}$ 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.

TODO:

- implement Real, Complex coefficient rings via libSINGULAR

EXAMPLES:

We show how to construct various multivariate polynomial rings:

```
sage: P.<x,y,z> = QQ[]
sage: P
Multivariate Polynomial Ring in x, y, z over Rational Field

sage: f = 27/113 * x^2 + y*z + 1/2; f
27/113*x^2 + y*z + 1/2

sage: P.term_order()
Degree reverse lexicographic term order

sage: P = PolynomialRing(GF(127),3,names='abc', order='lex')
sage: P
Multivariate Polynomial Ring in a, b, c over Finite Field of size 127

sage: a,b,c = P.gens()
sage: f = 57 * a^2*b + 43 * c + 1; f
57*a^2*b + 43*c + 1

sage: P.term_order()
Lexicographic term order

sage: z = QQ['z'].0
sage: K.<s> = NumberField(z^2 - 2)
sage: P.<x,y> = PolynomialRing(K, 2)
sage: 1/2*s*x^2 + 3/4*s
(1/2*s)*x^2 + (3/4*s)

sage: P.<x,y,z> = ZZ[]; P
Multivariate Polynomial Ring in x, y, z over Integer Ring
```

```
sage: P.<x,y,z> = Zmod(2^10)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 1024

sage: P.<x,y,z> = Zmod(3^10)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 59049

sage: P.<x,y,z> = Zmod(2^100)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 1267650600228229401496703205376

sage: P.<x,y,z> = Zmod(2521352)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 2521352
sage: type(P)
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular'>

sage: P.<x,y,z> = Zmod(25213521351515232)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 25213521351515232
sage: type(P)
<class 'sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict_with_category'>
```

We construct the Frobenius morphism on $\mathbf{F}_5[x, y, z]$ over $\mathbf{F}_5$:

```
sage: R.<x,y,z> = PolynomialRing(GF(5), 3)
sage: frob = R.hom([x^5, y^5, z^5])
sage: frob(x^2 + 2*y - z^4)
-z^20 + x^10 + 2*y^5
sage: frob((x + 2*y)^3)
x^15 + x^10*y^5 + 2*x^5*y^10 - 2*y^15
sage: (x^5 + 2*y^5)^3
x^15 + x^10*y^5 + 2*x^5*y^10 - 2*y^15
```

We make a polynomial ring in one variable over a polynomial ring in two variables:

```
sage: R.<x, y> = PolynomialRing(QQ, 2)
sage: S.<t> = PowerSeriesRing(R)
sage: t*(x+y)
(x + y)*t
```

TESTS:

```
sage: P.<x,y,z> = QQ[]
sage: loads(dumps(P)) == P
True
sage: loads(dumps(x)) == x
True
sage: P.<x,y,z> = GF(2^8,'a')[]
sage: loads(dumps(P)) == P
True
sage: loads(dumps(x)) == x
True
sage: P.<x,y,z> = GF(127)[]
sage: loads(dumps(P)) == P
True
sage: loads(dumps(x)) == x
True
sage: P.<x,y,z> = GF(127)[]
sage: loads(dumps(P)) == P
True
sage: loads(dumps(x)) == x
```

```
True

sage: Rt.<t> = PolynomialRing(QQ,1)
sage: p = 1+t
sage: R.<u,v> = PolynomialRing(QQ, 2)
sage: p(u/v)
(u + v)/v
```

Check if #6160 is fixed:

```
sage: x=var('x')
sage: K.<j> = NumberField(x-1728)
sage: R.<b,c> = K[]
sage: b-j*c
b - 1728*c
```

class `sage.rings.polynomial.multi_polynomial_libsingular.`**`MPolynomialRing_libsingular`**

Bases: `sage.rings.polynomial.multi_polynomial_ring_generic.MPolynomialRing_generic`

Construct a multivariate polynomial ring subject to the following conditions:

INPUT:

- **`base_ring` - base ring (must be either GF(q), ZZ, ZZ/nZZ,** 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[]
sage: P
Multivariate Polynomial Ring in x, y, z over Rational Field

sage: f = 27/113 * x^2 + y*z + 1/2; f
27/113*x^2 + y*z + 1/2

sage: P.term_order()
Degree reverse lexicographic term order

sage: P = PolynomialRing(GF(127),3,names='abc', order='lex')
sage: P
Multivariate Polynomial Ring in a, b, c over Finite Field of size 127

sage: a,b,c = P.gens()
sage: f = 57 * a^2*b + 43 * c + 1; f
57*a^2*b + 43*c + 1

sage: P.term_order()
Lexicographic term order

sage: z = QQ['z'].0
sage: K.<s> = NumberField(z^2 - 2)
sage: P.<x,y> = PolynomialRing(K, 2)
sage: 1/2*s*x^2 + 3/4*s
(1/2*s)*x^2 + (3/4*s)

sage: P.<x,y,z> = ZZ[]; P
```

```
Multivariate Polynomial Ring in x, y, z over Integer Ring

sage: P.<x,y,z> = Zmod(2^10)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 1024

sage: P.<x,y,z> = Zmod(3^10)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 59049

sage: P.<x,y,z> = Zmod(2^100)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 12676506002282294014967032

sage: P.<x,y,z> = Zmod(2521352)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 2521352
sage: type(P)
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular'>

sage: P.<x,y,z> = Zmod(25213521351515232)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 25213521351515232
sage: type(P)
<class 'sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict_with_category'>

sage: P.<x,y,z> = PolynomialRing(Integers(2^32),order='lex')
sage: P(2^32-1)
4294967295
```

TEST:

Make sure that a faster coercion map from the base ring is used; see trac ticket #9944:

```
sage: R.<x,y> = PolynomialRing(ZZ)
sage: R.coerce_map_from(R.base_ring())
Polynomial base injection morphism:
  From: Integer Ring
  To:   Multivariate Polynomial Ring in x, y over Integer Ring
```

`Element`
alias of `MPolynomial_libsingular`

`gen`(*n=0*)
Returns the `n`-th generator of this multivariate polynomial ring.

INPUT:

- `n` – an integer `>= 0`

EXAMPLES:

```
sage: P.<x,y,z> = QQ[]
sage: P.gen(),P.gen(1)
(x, y)

sage: P = PolynomialRing(GF(127),1000,'x')
sage: P.gen(500)
x500

sage: P.<SAGE,SINGULAR> = QQ[] # weird names
sage: P.gen(1)
SINGULAR
```

`ideal`(**gens*, ***kwds*)

Create an ideal in this polynomial ring.

INPUT:

- `*gens` - list or tuple of generators (or several input arguments)
- `coerce` - bool (default: `True`); this must be a keyword argument. Only set it to `False` if you are certain that each generator is already in the ring.

EXAMPLES:

```
sage: P.<x,y,z> = QQ[]
sage: sage.rings.ideal.Katsura(P)
Ideal (x + 2*y + 2*z - 1, x^2 + 2*y^2 + 2*z^2 - x, 2*x*y + 2*y*z - y) of Multivariate Polyno

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 Polyno
```

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
```

TESTS:

```
sage: P.<x,y,z> = QQ[]
sage: P.monomial_divides(P(1), P(0))
True
sage: P.monomial_divides(P(1), x)
True
```

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
```

TESTS:

```
sage: R.<x,y,z> = QQ[]
sage: P.<x,y,z> = QQ[]
sage: P.monomial_lcm(x*y,R.gen())
x*y

sage: P.monomial_lcm(P(3/2),P(2/3))
1

sage: P.monomial_lcm(x,P(1))
x
```

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
```

TESTS:

```
sage: Q.<x,y,z> = QQ[]
sage: P.<x,y,z> = QQ[]
sage: P.monomial_pairwise_prime(x^2*z^3, Q('y^4'))
True

sage: P.monomial_pairwise_prime(1/2*x^3*y^2, Q(0))
True

sage: P.monomial_pairwise_prime(P(1/2),x)
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):
...
ArithmeticError: Cannot divide these coefficients.
```

TESTS:

```
sage: R.<x,y,z> = QQ[]
sage: P.<x,y,z> = QQ[]
sage: P.monomial_quotient(x*y,x)
y

sage: P.monomial_quotient(x*y,R.gen())
y

sage: P.monomial_quotient(P(0),P(1))
0

sage: P.monomial_quotient(P(1),P(0))
Traceback (most recent call last):
...
ZeroDivisionError

sage: P.monomial_quotient(P(3/2),P(2/3), coeff=True)
9/4

sage: P.monomial_quotient(x,y)  # Note the wrong result
x*y^1048575*z^1048575 # 64-bit
x*y^65535*z^65535 # 32-bit

sage: P.monomial_quotient(x,P(1))
x
```

Warning: Assumes that the head term of f is a multiple of the head term of g and return the multiplicant m. If this rule is violated, funny things may happen.

monomial_reduce (*f*, *G*)

Try to find a `g` in `G` where `g.lm()` divides `f`. If found `(flt,g)` is returned, `(0,0)` otherwise, where `flt` is `f/g.lm()`.

It is assumed that `G` is iterable and contains *only* elements in this polynomial ring.

Coefficients are ignored.

INPUT:

- `f` - monomial
- `G` - list/set of mpolynomials

EXAMPLES:

```
sage: 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)
```

TESTS:

```
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(P(0),G)
(0, 0)

sage: P.monomial_reduce(f,[P(0)])
(0, 0)
```

ngens ()

Returns the number of variables in this multivariate polynomial ring.

EXAMPLES:

```
sage: P.<x,y> = QQ[]
sage: P.ngens()
2

sage: k.<a> = GF(2^16)
sage: P = PolynomialRing(k,1000,'x')
sage: P.ngens()
1000
```

class `sage.rings.polynomial.multi_polynomial_libsingular.`**MPolynomial_libsingular**

Bases: `sage.rings.polynomial.multi_polynomial.MPolynomial`

A multivariate polynomial implemented using libSINGULAR.

add_m_mul_q (*m*, *q*)

Return `self + m*q`, where `m` must be a monomial and `q` a polynomial.

INPUT:

- `m` - a monomial
- `q` - a polynomial

EXAMPLES:

```
sage: P.<x,y,z>=PolynomialRing(QQ,3)
sage: x.add_m_mul_q(y,z)
y*z + x
```

TESTS:

```
sage: R.<x,y,z>=PolynomialRing(QQ,3)
sage: P.<x,y,z>=PolynomialRing(QQ,3)
sage: P(0).add_m_mul_q(P(0),P(1))
0
sage: x.add_m_mul_q(R.gen(),R.gen(1))
x*y + x
```

coefficient (*degrees*)

Return the coefficient of the variables with the degrees specified in the python dictionary `degrees`. Mathematically, this is the coefficient in the base ring adjoined by the variables of this ring not listed in `degrees`. However, the result has the same parent as this polynomial.

This function contrasts with the function `monomial_coefficient` which returns the coefficient in the base ring of a monomial.

INPUT:

- **`degrees` - Can be any of:**
  - a dictionary of degree restrictions
  - a list of degree restrictions (with None in the unrestricted variables)
  - a monomial (very fast, but not as flexible)

OUTPUT: element of the parent of this element.

Note: For coefficients of specific monomials, look at `monomial_coefficient()`.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: f=x*y+y+5
sage: f.coefficient({x:0,y:1})
1
sage: f.coefficient({x:0})
y + 5
sage: f=(1+y+y^2)*(1+x+x^2)
sage: f.coefficient({x:0})
y^2 + y + 1
sage: f.coefficient([0,None])
y^2 + y + 1
sage: f.coefficient(x)
y^2 + y + 1
```

Be aware that this may not be what you think! The physical appearance of the variable x is deceiving – particularly if the exponent would be a variable.

```
sage: f.coefficient(x^0) # outputs the full polynomial
x^2*y^2 + x^2*y + x*y^2 + x^2 + x*y + y^2 + x + y + 1
sage: R.<x,y> = GF(389)[]
sage: f=x*y+5
sage: c=f.coefficient({x:0,y:0}); c
5
```

```
sage: parent(c)
Multivariate Polynomial Ring in x, y over Finite Field of size 389
```

AUTHOR:

- Joel B. Mohler (2007.10.31)

coefficients()

Return the nonzero coefficients of this polynomial in a list. The returned list is decreasingly ordered by the term ordering of the parent.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ, order='degrevlex')
sage: f=23*x^6*y^7 + x^3*y+6*x^7*z
sage: f.coefficients()
[23, 6, 1]

sage: R.<x,y,z> = PolynomialRing(QQ, order='lex')
sage: f=23*x^6*y^7 + x^3*y+6*x^7*z
sage: f.coefficients()
[6, 23, 1]
```

AUTHOR:

- Didier Deshommes

constant_coefficient()

Return the constant coefficient of this multivariate polynomial.

EXAMPLES:

```
sage: 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 maximal degree of this polynomial in `x`, where `x` must be one of the generators for the parent of this polynomial.

INPUT:

- `x` - (default: `None`) a multivariate polynomial which is (or coerces to) a generator of the parent of self. If `x` is `None`, return the total degree, which is the maximum degree of any monomial. Note that a matrix term ordering alters the grading of the generators of the ring; see the tests below. To avoid this behavior, use either `exponents()` for the exponents themselves, or the optional argument `std_grading=False`.

OUTPUT: integer

EXAMPLES:

```
sage: R.<x, y> = QQ[]
sage: f = y^2 - x^9 - x
sage: f.degree(x)
9
sage: f.degree(y)
```

```
2
sage: (y^10*x - 7*x^2*y^5 + 5*x^3).degree(x)
3
sage: (y^10*x - 7*x^2*y^5 + 5*x^3).degree(y)
10
```

TESTS:

```
sage: P.<x, y> = QQ[]
sage: P(0).degree(x)
-1
sage: P(1).degree(x)
0
```

With a matrix term ordering, the grading of the generators is determined by the first row of the matrix. This affects the behavior of `degree()` when no variable is specified. To evaluate the degree with a standard grading, use the optional argument `std_grading=True`.

> sage: tord = TermOrder(matrix([3,0,1,1,1,0,1,0,0])) sage: R.<x,y,z> = PolynomialRing(QQ,'x',3,order=tord) sage: (x^3*y+x*z^4).degree() 9 sage: (x^3*y+x*z^4).degree(std_grading=True) 5 sage: x.degree(x), y.degree(y), z.degree(z) (1, 1, 1)

The following example is inspired by trac 11652:

```
sage: R.<p,q,t> = ZZ[]
sage: poly = p+q^2+t^3
sage: poly = poly.polynomial(t)[0]
sage: poly
q^2 + p
```

There is no canonical coercion from `R` to the parent of `poly`, so this doesn't work:

```
sage: poly.degree(q)
Traceback (most recent call last):
...
TypeError: argument must canonically coerce to parent
```

Using a non-canonical coercion does work, but we require this to be done explicitly, since it can lead to confusing results if done automatically:

```
sage: poly.degree(poly.parent()(q))
2
sage: poly.degree(poly.parent()(p))
1
sage: T.<x,y> = ZZ[]
sage: poly.degree(poly.parent()(x))  # noncanonical coercions can be confusing
1
```

The argument to degree has to be a generator:

```
sage: pp = poly.parent().gen(0)
sage: poly.degree(pp)
1
sage: poly.degree(pp+1)
Traceback (most recent call last):
...
TypeError: argument must be a generator
```

Canonical coercions are used:

```
sage: S = ZZ['p,q']
sage: poly.degree(S.0)
1
sage: poly.degree(S.1)
2
```

degrees()

Returns a tuple with the maximal degree of each variable in this polynomial. The list of degrees is ordered by the order of the generators.

EXAMPLES:

```
sage: R.<y0,y1,y2> = PolynomialRing(QQ,3)
sage: q = 3*y0*y1*y1*y2; q
3*y0*y1^2*y2
sage: q.degrees()
(1, 2, 1)
sage: (q + y0^5).degrees()
(5, 2, 1)
```

dict()

Return a dictionary representing self. This dictionary is in the same format as the generic MPolynomial: The dictionary consists of `ETuple:coefficient` pairs.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: f=2*x*y^3*z^2 + 1/7*x^2 + 2/3
sage: f.dict()
{(0, 0, 0): 2/3, (1, 3, 2): 2, (2, 0, 0): 1/7}
```

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*y^2 + 1/4*x*y*z + 3/2*x*z^2 - 1/2*z^2
sage: f.discriminant(x)
1
sage: f.discriminant(y)
-383/16*x^2*z^2 + 8*x*z^2
sage: f.discriminant(z)
-383/16*x^2*y^2 + 8*x*y^2
```

Note that, unlike the univariate case, the result lives in the same ring as the polynomial:

```
sage: R.<x,y>=QQ[]
sage: f=x^5*y+3*x^2*y^2-2*x+y-1
sage: f.discriminant(y)
x^10 + 2*x^5 + 24*x^3 + 12*x^2 + 1
sage: f.polynomial(y).discriminant()
x^10 + 2*x^5 + 24*x^3 + 12*x^2 + 1
```

```
sage: f.discriminant(y).parent()==f.polynomial(y).discriminant().parent()
False
```

AUTHOR: Miguel Marco

exponents(*as_ETuples=True*)

Return the exponents of the monomials appearing in this polynomial.

INPUT:

- **as_ETuples - (default: True) if true returns the result as an list of ETuples** otherwise returns a list of tuples

EXAMPLES:

```
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)]
```

factor(*proof=True*)

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()
sage: 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 # order is somewhat random
(-1) * x * (-x + y) * (x + y) * (x - 1)^2
```

Next we factor a polynomial, but over a finite field of order 9.:

```
sage: K.<a> = GF(3^2)
sage: R.<x, y> = K[]
sage: f = (x^3 + 2*a*y^2*x) * (x^2 + x + 1); f
x^5 + (-a)*x^3*y^2 + x^4 + (-a)*x^2*y^2 + x^3 + (-a)*x*y^2
sage: F = f.factor()
sage: F
((-a)) * x * (x - 1)^2 * ((-a + 1)*x^2 + y^2)
sage: f - F
0
```

Next we factor a polynomial over a number field.:

```
sage: p = var('p')
sage: K.<s> = NumberField(p^3-2)
sage: KXY.<x,y> = K[]
sage: factor(x^3 - 2*y^3)
(x + (-s)*y) * (x^2 + (s)*x*y + (s^2)*y^2)
sage: k = (x^3-2*y^3)^5*(x+s*y)^2*(2/3 + s^2)
sage: k.factor()
((s^2 + 2/3)) * (x + (s)*y)^2 * (x + (-s)*y)^5 * (x^2 + (s)*x*y + (s^2)*y^2)^5
```

This shows that ticket #2780 is fixed, i.e. that the unit part of the factorization is set correctly:

```
sage: x = var('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)[]
sage: f = 9*(x-1)^2*(y+z)
sage: f.factor()
(9) * (y + z) * (x - 1)^2

sage: R.<x,w,v,u> = QQ['x','w','v','u']
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[]
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[]
sage: P(2^3*7).factor()
2^3 * 7
```

Factorization for finite prime fields with characteristic $> 2^{29}$ is not supported either.

```
sage: q = 1073741789
sage: T.<aa, bb> = PolynomialRing(GF(q))
sage: f = aa^2 + 12124343*bb*aa + 32434598*bb^2
sage: f.factor()
Traceback (most recent call last):
...
NotImplementedError: Factorization of multivariate polynomials over prime fields with charac
```

Finally, factorization over the integers is not supported.

```
sage: P.<x,y> = PolynomialRing(ZZ)
sage: f = (3*x + 4)*(5*x - 2)
sage: f.factor()
Traceback (most recent call last):
```

```
...
NotImplementedError: Factorization of multivariate polynomials over non-fields is not implem
```

TESTS:

This shows that trac ticket #10270 is fixed:

```
sage: R.<x,y,z> = GF(3)[]
sage: f = x^2*z^2+x*y*z-y^2
sage: f.factor()
x^2*z^2 + x*y*z - y^2
```

This checks that trac ticket #11838 is fixed:

```
sage: K = GF(4,'a')
sage: a = K.gens()[0]
sage: R.<x,y> = K[]
sage: p=x^8*y^3 + x^2*y^9 + a*x^9 + a*x*y^4
sage: q=y^11 + (a)*y^10 + (a + 1)*x*y^3
sage: f = p*q
sage: f.factor()
x * y^3 * (y^8 + (a)*y^7 + (a + 1)*x) * (x^7*y^3 + x*y^9 + (a)*x^8 + (a)*y^4)
```

We test several examples which were known to return wrong results in the past (see trac ticket #10902):

```
sage: R.<x,y> = GF(2)[]
sage: p = x^3*y^7 + x^2*y^6 + x^2*y^3
sage: q = x^3*y^5
sage: f = p*q
sage: p.factor()*q.factor()
x^5 * y^8 * (x*y^4 + y^3 + 1)
sage: f.factor()
x^5 * y^8 * (x*y^4 + y^3 + 1)
sage: f.factor().expand() == f
True

sage: R.<x,y> = GF(2)[]
sage: p=x^8 + y^8; q=x^2*y^4 + x
sage: f=p*q
sage: lf = f.factor()
sage: f-lf
0

sage: R.<x,y> = GF(3)[]
sage: p = -x*y^9 + x
sage: q = -x^8*y^2
sage: f = p*q
sage: f
x^9*y^11 - x^9*y^2
sage: f.factor()
y^2 * (y - 1)^9 * x^9
sage: f - f.factor()
0

sage: R.<x,y> = GF(5)[]
sage: p=x^27*y^9 + x^32*y^3 + 2*x^20*y^10 - x^4*y^24 - 2*x^17*y
sage: q=-2*x^10*y^24 + x^9*y^24 - 2*x^3*y^30
sage: f=p*q; f-f.factor()
0
```

```
sage: R.<x,y> = GF(7)[]
sage: p=-3*x^47*y^24
sage: q=-3*x^47*y^37 - 3*x^24*y^49 + 2*x^56*y^8 + 3*x^29*y^15 - x^2*y^33
sage: f=p*q
sage: f-f.factor()
0
```

The following examples used to give a Segmentation Fault, see trac ticket #12918 and trac ticket #13129:

```
sage: R.<x,y> = GF(2)[]
sage: f = x^6 + x^5 + y^5 + y^4
sage: f.factor()
x^6 + x^5 + y^5 + y^4
sage: f = x^16*y + x^10*y + x^9*y + x^6*y + x^5 + x*y + y^2
sage: f.factor()
x^16*y + x^10*y + x^9*y + x^6*y + x^5 + x*y + y^2
```

Test trac ticket #12928:

```
sage: R.<x,y> = GF(2)[]
sage: p = x^2 + y^2 + x + 1
sage: q = x^4 + x^2*y^2 + y^4 + x*y^2 + x^2 + y^2 + 1
sage: factor(p*q)
(x^2 + y^2 + x + 1) * (x^4 + x^2*y^2 + y^4 + x*y^2 + x^2 + y^2 + 1)
```

Check that trac ticket #13770 is fixed:

```
sage: U.<y,t> = GF(2)[]
sage: f = y*t^8 + y^5*t^2 + y*t^6 + t^7 + y^6 + y^5*t + y^2*t^4 + y^2*t^2 + y^2*t + t^3 + y^
sage: l = f.factor()
sage: l[0][0]==t^2 + y + t + 1 or l[1][0]==t^2 + y + t + 1
True
```

The following used to sometimes take a very long time or get stuck, see trac ticket #12846. These 100 iterations should take less than 1 second:

```
sage: K.<a> = GF(4)
sage: R.<x,y> = K[]
sage: f = (a + 1)*x^145*y^84 + (a + 1)*x^205*y^17 + x^32*y^112 + x^92*y^45
sage: for i in range(100):
...       assert len(f.factor()) == 4
```

gcd(*right*, *algorithm=None*, ***kwds*)

Return the greatest common divisor of self and right.

INPUT:

- `right` - polynomial
- `algorithm` - `ezgcd` - EZGCD algorithm - `modular` - multi-modular algorithm (default)
- `**kwds` - ignored

EXAMPLES:

```
sage: P.<x,y,z> = QQ[]
sage: f = (x*y*z)^6 - 1
sage: g = (x*y*z)^4 - 1
sage: f.gcd(g)
x^2*y^2*z^2 - 1
sage: GCD([x^3 - 3*x + 2, x^4 - 1, x^6 -1])
x - 1
```

```
sage: R.<x,y> = QQ[]
sage: f = (x^3 + 2*y^2*x)^2
sage: g = x^2*y^2
sage: f.gcd(g)
x^2
```

We compute a gcd over a finite field:

```
sage: F.<u> = GF(31^2)
sage: R.<x,y,z> = F[]
sage: p = x^3 + (1+u)*y^3 + z^3
sage: q = p^3 * (x - y + z*u)
sage: gcd(p,q)
x^3 + (u + 1)*y^3 + z^3
sage: gcd(p,q)  # yes, twice -- tests that singular ring is properly set.
x^3 + (u + 1)*y^3 + z^3
```

We compute a gcd over a number field:

```
sage: 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
```

TESTS:

```
sage: Q.<x,y,z> = QQ[]
sage: P.<x,y,z> = QQ[]
sage: P(0).gcd(Q(0))
0
sage: x.gcd(1)
1

sage: k.<a> = GF(9)
sage: R.<x,y> = PolynomialRing(k)
sage: f = R.change_ring(GF(3)).gen()
sage: g = x+y
sage: g.gcd(f)
1
sage: x.gcd(R.change_ring(GF(3)).gen())
x
```

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]
```

integral(*var*)

Integrates this polynomial with respect to the provided variable.

One requires that $\mathbf{Q}$ is contained in the ring.

INPUT:

- `variable` - the integral is taken with respect to variable

EXAMPLES:

```
sage: R.<x, y> = PolynomialRing(QQ, 2)
sage: f = 3*x^3*y^2 + 5*y^2 + 3*x + 2
sage: f.integral(x)
3/4*x^4*y^2 + 5*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 trac ticket #15896 is solved:

```
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
```

TESTS:

```
sage: z, w = polygen(QQ, 'z, w')
sage: f.integral(z)
Traceback (most recent call last):
...
TypeError: the variable is not in the same ring as self

sage: f.integral(y**2)
Traceback (most recent call last):
...
TypeError: not a variable in the same ring as self

sage: x,y = polygen(ZZ,'x,y')
sage: y.integral(x)
Traceback (most recent call last):
...
TypeError: the ring must contain the rational numbers
```

inverse_of_unit()

Return the inverse of this polynomial if it is a unit.

EXAMPLES:

```
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:

```
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:

```
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
```

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_unit()

Return `True` if self is a unit.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
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

sage: R.<x,y> = ZZ[]
sage: R(1).is_unit()
True
sage: R(2).is_unit()
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)
sage: f.is_univariate()
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
```

lc()

Leading coefficient of this polynomial with respect to the term order of `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
```

lcm(*g*)

Return the least common multiple of `self` and 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: 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^4 + a^3 + a)*x^4*y + (a^5)*x^4

sage: w = var('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
```

lift(*I*)

given an ideal `I = (f_1,...,f_r)` and some `g (== self)` in `I`, find `s_1,...,s_r` such that `g = s_1 f_1 + ... + s_r f_r`.

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*y^13 + y^12
sage: M = f.lift(I)
sage: M
[y^7, x^7*y^2 + x^8 + x^5*y^3 + x^6*y + x^3*y^4 + x^4*y^2 + x*y^5 + x^2*y^3 + y^4]
sage: sum( map( mul , zip( M, I.gens() ) ) ) == f
True
```

Check that trac ticket #13671 is fixed:

```
sage: R.<x1,x2> = QQ[]
sage: I = R.ideal(x2**2 + x1 - 2, x1**2 - 1)
sage: f = I.gen(0) + x2*I.gen(1)
sage: f.lift(I)
[1, x2]
sage: (f+1).lift(I)
Traceback (most recent call last):
...
ValueError: polynomial is not in the ideal
sage: f.lift(I)
[1, x2]
```

TESTS:

Check that trac ticket #13714 is fixed:

```
sage: R.<x1,x2> = QQ[]
sage: I = R.ideal(x2**2 + x1 - 2, x1**2 - 1)
```

```
sage: R.one().lift(I)
Traceback (most recent call last):
...
ValueError: polynomial is not in the ideal
sage: foo = I.complete_primary_decomposition() # indirect doctest
sage: foo[0][0]
Ideal (x2 - 1, x1 - 1) of Multivariate Polynomial Ring in x1, x2 over Rational Field
```

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
sage: f.lm()
x^4*y^7*z
```

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)
1
sage: f.monomial_coefficient(x*y)
5
sage: f.monomial_coefficient(x^9)
-1
sage: f.monomial_coefficient(x^10)
0
```

monomials()

Return the list of monomials in self. The returned list is decreasingly ordered by the term ordering of `self.parent()`.

EXAMPLES:

```
sage: P.<x,y,z> = QQ[]
sage: f = x + 3/2*y*z^2 + 2/3
sage: f.monomials()
[y*z^2, x, 1]
sage: f = P(3/2)
sage: f.monomials()
[1]
```

TESTS:

```
sage: P.<x,y,z> = QQ[]
sage: f = x
sage: f.monomials()
[x]
```

Check if trac ticket #12706 is fixed:

```
sage: f = P(0)
sage: f.monomials()
[]
```

Check if trac ticket #7152 is fixed:

```
sage: x=var('x')
sage: K.<rho> = NumberField(x**2 + 1)
sage: R.<x,y> = QQ[]
```

```
sage: p = rho*x
sage: q = x
sage: p.monomials()
[x]
sage: q.monomials()
[x]
sage: p.monomials()
[x]
```

numerator()

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

If the base_field of self is the Rational Field then the numerator is a polynomial whose base_ring is the Integer Ring, this is done for compatibility to the univariate case.

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

EXAMPLES:

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

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

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

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

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

```
sage: K=QQ['x,y']
sage: f=K.random_element()
sage: f.numerator() / f.denominator() == f
True
```

The following tests against a bug that has been fixed in trac ticket #11780:

```
sage: P.<foo,bar> = ZZ[]
sage: Q.<foo,bar> = QQ[]
sage: f = Q.random_element()
sage: f.numerator().parent() is P
True
```

nvariables()

Return the number variables in this polynomial.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(GF(127))
sage: f = x*y + z
sage: f.nvariables()
3
sage: f = x + y
sage: f.nvariables()
2
```

quo_rem(*right*)

Returns quotient and remainder of self and right.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: f = y*x^2 + x + 1
sage: f.quo_rem(x)
(x*y + 1, 1)
sage: f.quo_rem(y)
(x^2, x + 1)

sage: R.<x,y> = ZZ[]
sage: f = 2*y*x^2 + x + 1
sage: f.quo_rem(x)
(2*x*y + 1, 1)
sage: f.quo_rem(y)
(2*x^2, x + 1)
sage: f.quo_rem(3*x)
(0, 2*x^2*y + x + 1)
```

TESTS:

```
sage: R.<x,y> = QQ[]
sage: R(0).quo_rem(R(1))
(0, 0)
sage: R(1).quo_rem(R(0))
Traceback (most recent call last):
...
ZeroDivisionError
```

reduce(*I*)

Return the normal form of self w.r.t. `I`, i.e. return the remainder of this polynomial with respect to the polynomials in `I`. If the polynomial set/list `I` is not a (strong) Groebner basis the result is not canonical.

A strong Groebner basis `G` of `I` implies that for every leading term `t` of `I` there exists an element `g` of `G`, such that the leading term of `g` divides `t`.

INPUT:

- `I` - a list/set of polynomials. If `I` is an ideal, the generators are used.

EXAMPLES:

```
sage: P.<x,y,z> = QQ[]
sage: f1 = -2 * x^2 + x^3
sage: f2 = -2 * y + x* y
sage: f3 = -x^2 + y^2
sage: F = Ideal([f1,f2,f3])
sage: g = x*y - 3*x*y^2
sage: g.reduce(F)
-6*y^2 + 2*y
```

```
sage: g.reduce(F.gens())
-6*y^2 + 2*y
```

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
sage: g.reduce(F)
-6*y^2 + 2*y
sage: g.reduce(F.gens())
-6*y^2 + 2*y

sage: f = 3*x
sage: f.reduce([2*x,y])
3*x
```

resultant (*other*, *variable=None*)

Compute the resultant of this polynomial and the first argument with respect to the variable given as the second argument.

If a second argument is not provide the first variable of the parent is chosen.

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)
sage: f = 3 * (x+2)^3 + y
sage: g = x+y+z
sage: f.resultant(g,x)
3*y^3 + 9*y^2*z + 9*y*z^2 + 3*z^3 - 18*y^2 - 36*y*z - 18*z^2 + 35*y + 36*z - 24
```

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*b*u - 4*y^4
```

TESTS:

```
sage: P.<x,y> = PolynomialRing(QQ, order='degrevlex')
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
```

```
sage: P.<x,y> = PolynomialRing(ZZ,2)
sage: f = x+y
sage: g=y^2+x
sage: f.resultant(g,y)
x^2 + x
```

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
```

TESTS:

```
sage: Q.<x,y,z>=PolynomialRing(QQ,3)
sage: P.<x,y,z>=PolynomialRing(QQ,3)
sage: P(0).sub_m_mul_q(P(0),P(1))
0
sage: x.sub_m_mul_q(Q.gen(1),Q.gen(2))
-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
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 subsituted 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
```

TESTS:

```
sage: P.<x,y,z> = QQ[]
sage: f = y
sage: f.subs({y:x}).subs({x:z})
z
```

We test that we change the ring even if there is nothing to do:

```
sage: P = QQ['x,y']
sage: x = var('x')
sage: parent(P.zero() / x)
Symbolic Ring
```

We are catching overflows:

```
sage: R.<x,y> = QQ[]
sage: n=1000; f = x^n
sage: try:
....:   f.subs(x = x^n)
....:   print "no overflow"
....: except OverflowError:
....:   print "overflow"
overflow    # 32-bit
```

```
x^1000000   # 64-bit
no overflow # 64-bit

sage: n=100000;
sage: try:
....:   f = x^n
....:   f.subs(x = x^n)
....:   print "no overflow"
....: except OverflowError:
....:   print "overflow"
overflow
```

total_degree(*std_grading=False*)

Return the total degree of `self`, which is the maximum degree of all monomials in `self`.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: f=2*x*y^3*z^2
sage: f.total_degree()
6
sage: f=4*x^2*y^2*z^3
sage: f.total_degree()
7
sage: f=99*x^6*y^3*z^9
sage: f.total_degree()
18
sage: f=x*y^3*z^6+3*x^2
sage: f.total_degree()
10
sage: f=z^3+8*x^4*y^5*z
sage: f.total_degree()
10
sage: f=z^9+10*x^4+y^8*x^2
sage: f.total_degree()
10
```

TESTS:

```
sage: R.<x,y,z> = QQ[]
sage: R(0).total_degree()
-1
sage: R(1).total_degree()
0
```

With a matrix term ordering, the grading changes. To evaluate the total degree using the standard grading, use the optional argument‘‘std_grading=True‘‘.

> sage: tord=TermOrder(matrix([3,0,1,1,1,0,1,0,0])) sage: R.<x,y,z> = PolynomialRing(QQ,'x',3,order=tord) sage: (x^2*y).total_degree() 6 sage: (x^2*y).total_degree(std_grading=True) 3

univariate_polynomial(*R=None*)

Returns a univariate polynomial associated to this multivariate polynomial.

INPUT:

- `R` - (default: `None`) PolynomialRing

If this polynomial is not in at most one variable, then a `ValueError` exception is raised. This is checked using the `is_univariate()` method. The new Polynomial is over the same base ring as the given `MPolynomial` and in the variable `x` if no ring `R` is provided.

EXAMPLES:

```
sage: R.<x, y> = QQ[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.univariate_polynomial()
Traceback (most recent call last):
...
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:

```
sage: g = R(1)
sage: h = g.univariate_polynomial(); h
1
sage: h.parent()
Univariate Polynomial Ring in x over Rational Field
```

variable(*i=0*)

Return the i-th variable occurring in self. The index i is the index in `self.variables()`.

EXAMPLES:

```
sage: P.<x,y,z> = GF(2)[]
sage: f = x*z^2 + z + 1
sage: f.variables()
(x, z)
sage: f.variable(1)
z
```

variables()

Return a tuple of all variables occurring in self.

EXAMPLES:

```
sage: P.<x,y,z> = GF(2)[]
sage: f = x*z^2 + z + 1
sage: f.variables()
(x, z)
```

sage.rings.polynomial.multi_polynomial_libsingular.**unpickle_MPolynomialRing_libsingular**(*bas nam term*

inverse function for `MPolynomialRing_libsingular.__reduce__`

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(QQ)
sage: loads(dumps(P)) is P # indirect doctest
True
```

sage.rings.polynomial.multi_polynomial_libsingular.**unpickle_MPolynomial_libsingular**(*R*, *d*)

Deserialize an `MPolynomial_libsingular` object

INPUT:

- R - the base ring
- d - a Python dictionary as returned by MPolynomial_libsingular.dict()

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(QQ)
sage: loads(dumps(x)) == x # indirect doctest
True
```

3.9 Direct low-level access to SINGULAR's Groebner basis engine via libSINGULAR.

AUTHOR:

- Martin Albrecht (2007-08-08): initial version

EXAMPLES:

```
sage: x,y,z = QQ['x,y,z'].gens()
sage: I = ideal(x^5 + y^4 + z^3 - 1,  x^3 + y^3 + z^2 - 1)
sage: I.groebner_basis('libsingular:std')
[y^6 + x*y^4 + 2*y^3*z^2 + x*z^3 + z^4 - 2*y^3 - 2*z^2 - x + 1,
 x^2*y^3 - y^4 + x^2*z^2 - z^3 - x^2 + 1, x^3 + y^3 + z^2 - 1]
```

We compute a Groebner basis for cyclic 6, which is a standard benchmark and test ideal:

```
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: 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: 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: 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^3 - x*y, z^3 - x, x^2 - 3*y, 9*y^2 - y*z + 1]

sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = ideal( x^2 - 3*y, y^3 - x*y, z^3 - x, x^4 - y*z + 1 )
sage: I.interreduced_basis()
[y*z^2 - 81*x*y - 9*y - z, z^3 - x, x^2 - 3*y, y^2 - 1/9*y*z + 1/9]
```

sage.rings.polynomial.multi_polynomial_ideal_libsingular.**kbase_libsingular**(*I*)

SINGULAR's `kbase()` algorithm.

INPUT:

- `I` – a groebner basis of an ideal

OUTPUT:

Computes a vector space basis (consisting of monomials) of the quotient ring by the ideal, resp. of a free module by the module, in case it is finite dimensional and if the input is a standard basis with respect to the ring ordering. If the input is not a standard basis, the leading terms of the input are used and the result may have no meaning.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: I = R.ideal(x^2-2*y^2, x*y-3)
sage: I.normal_basis()
[y^3, y^2, y, 1]
```

sage.rings.polynomial.multi_polynomial_ideal_libsingular.**slimgb_libsingular**(*I*)

SINGULAR's `slimgb()` algorithm.

INPUT:

- `I` – a Sage ideal

sage.rings.polynomial.multi_polynomial_ideal_libsingular.**std_libsingular**(*I*)

SINGULAR's `std()` algorithm.

INPUT:

- `I` – a Sage ideal

3.10 PolyDict engine for generic multivariate polynomial rings

This module provides an implementation of the underlying arithmetic for multi-variate polynomial rings using Python dicts.

This class is not meant for end users, but instead for implementing multivariate polynomial rings over a completely general base. It does not do strong type checking or have parents, etc. For speed, it has been implemented in Cython.

The functions in this file use the 'dictionary representation' of multivariate polynomials

`{(e1,...,er):c1,...} <-> c1*x1^e1*...*xr^er+...,`

which we call a polydict. The exponent tuple `(e1,...,er)` in this representation is an instance of the class `ETuple`. This class behaves like a normal Python tuple but also offers advanced access methods for sparse monomials like positions of non-zero exponents etc.

AUTHORS:

- William Stein
- David Joyner
- Martin Albrecht (ETuple)
- Joel B. Mohler (2008-03-17) – ETuple rewrite as sparse C array

class `sage.rings.polynomial.polydict.`**`ETuple`**

Bases: `object`

Representation of the exponents of a polydict monomial. If (0,0,3,0,5) is the exponent tuple of x_2^3*x_4^5 then this class only stores {2:3,4:5} instead of the full tuple. This sparse information may be obtained by provided methods.

The index/value data is all stored in the _data C int array member variable. For the example above, the C array would contain 2,3,4,5. The indices are interlaced with the values.

This data structure is very nice to work with for some functions implemented in this class, but tricky for others. One reason that I really like the format is that it requires a single memory allocation for all of the values. A hash table would require more allocations and presumably be slower. I didn't benchmark this question (although, there is no question that this is much faster than the prior use of python dicts).

`combine_to_positives`(*other*)

Given a pair of ETuples (self, other), returns a triple of ETuples (a, b, c) so that self = a + b,other = a + c and b and c have all positive entries.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([-2,1,-5, 3, 1,0])
sage: f = ETuple([1,-3,-3,4,0,2])
sage: 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
sage: e = ETuple([1,0,2])
sage: f = ETuple([0,0,1])
sage: e.common_nonzero_positions(f)
{0, 2}
sage: e.common_nonzero_positions(f,sort=True)
[0, 2]
```

`eadd`(*other*)

Vector addition of self with other.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
sage: f = ETuple([0,1,1])
sage: e.eadd(f)
(1, 1, 3)
```

Verify that trac 6428 has been addressed:

```
sage: R.<y,z> = Frac(QQ['x'])[]
sage: type(y)
<class 'sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict'>
sage: y^(2^32)
Traceback (most recent call last):
...
OverflowError: Exponent overflow (2147483648).
```

eadd_p (*other*, *pos*)

Adds other to self at position pos.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import ETuple
sage: 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
```

emax (*other*)

Vector of maximum of components of self and other.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
sage: f = ETuple([0,1,1])
sage: e.emax(f)
(1, 1, 2)
sage: e=ETuple((1,2,3,4))
sage: f=ETuple((4,0,2,1))
sage: f.emax(e)
(4, 2, 3, 4)
sage: e=ETuple((1,-2,-2,4))
sage: f=ETuple((4,0,0,0))
sage: f.emax(e)
(4, 0, 0, 4)
sage: f.emax(e).nonzero_positions()
[0, 3]
```

emin (*other*)

Vector of minimum of components of self and other.

EXAMPLES:

```
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:

```
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
sage: e.emul(2)
(2, 0, 4)
```

esub (*other*)

Vector subtraction 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.esub(f)
(1, -1, 1)
```

nonzero_positions (*sort=False*)

Returns the positions of non-zero exponents in the tuple.

INPUT:

- `sort` – if True a sorted list is returned. If False an unsorted list is returned. (default: False)

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
sage: e.nonzero_positions()
[0, 2]
```

nonzero_values (*sort=True*)

Returns the non-zero values of the tuple.

INPUT:

- `sort` – if True the values are sorted by their indices. Otherwise the values are returned unsorted. (default: True)

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([2,0,1])
sage: e.nonzero_values()
[2, 1]
sage: f = ETuple([0,-1,1])
sage: f.nonzero_values(sort=True)
[-1, 1]
```

reversed ()

Returns the reversed ETuple of self.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,2,3])
sage: e.reversed()
(3, 2, 1)
```

sparse_iter()

Iterator over the elements of self where the elements are returned as `(i,e)` where `i` is the position of `e` in the tuple.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2,0,3])
sage: list(e.sparse_iter())
[(0, 1), (2, 2), (4, 3)]
```

class sage.rings.polynomial.polydict.**ETupleIter**

Bases: `object`

next()

x.next() -> the next value, or raise StopIteration

class sage.rings.polynomial.polydict.**PolyDict**

Bases: `object`

INPUT:

- `pdict` – list, which represents a multi-variable polynomial with the distribute representation (a copy is not made)
- `zero` – (optional) zero in the base ring
- `force_int_exponents` – bool (optional) arithmetic with int exponents is much faster than some of the alternatives, so this is True by default.
- `force_etuples` – bool (optional) enforce that the exponent tuples are instances of ETuple class

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: PolyDict({(2,3):2, (1,2):3, (2,1):4})
PolyDict with representation {(1, 2): 3, (2, 3): 2, (2, 1): 4}

# I've removed fractional exponent support in ETuple when moving to a sparse C integer array
#sage: PolyDict({(2/3,3,5):2, (1,2,1):3, (2,1):4}, force_int_exponents=False)
#PolyDict with representation {(2, 1): 4, (1, 2, 1): 3, (2/3, 3, 5): 2}

sage: PolyDict({(2,3):0, (1,2):3, (2,1):4}, remove_zero=True)
PolyDict with representation {(1, 2): 3, (2, 1): 4}

sage: PolyDict({(0,0):RIF(-1,1)}, remove_zero=True)
PolyDict with representation {(0, 0): 0.?}
```

coefficient(*mon*)

Return a polydict that defines a polynomial in 1 less number of variables that gives the coefficient of mon in this polynomial.

The coefficient is defined as follows. If f is this polynomial, then the coefficient is the sum T/mon where the sum is over terms T in f that are exactly divisible by mon.

coefficients()

Return the coefficients of self.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
```

```
sage: f.coefficients()
[3, 2, 4]
```

compare(*other*, *fn=None*)

degree(*x=None*)

dict()

Return a copy of the dict that defines self. It is safe to change this. For a reference, use dictref.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: f.dict()
{(1, 2): 3, (2, 1): 4, (2, 3): 2}
```

exponents()

Return the exponents of self.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: f.exponents()
[(1, 2), (2, 3), (2, 1)]
```

homogenize(*var*)

is_homogeneous()

latex(*vars*, *atomic_exponents=True*, *atomic_coefficients=True*, *cmpfn=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: f.latex(['a','WW'])
'2 a^{2} WW^{3} + 4 a^{2} WW + 3 a WW^{2}'
```

When `atomic_exponents` is False, the exponents are surrounded in parenthesis, since ^ has such high precedence:

```
# I've removed fractional exponent support in ETuple when moving to a sparse C integer array
#sage: f = PolyDict({(2/3,3,5):2, (1,2,1):3, (2,1,1):4}, force_int_exponents=False)
#sage: f.latex(['a','b','c'], atomic_exponents=False)
#'4 a^{2}bc + 3 ab^{2}c + 2 a^{2/3}b^{3}c^{5}'
```

TESTS:

We check that the issue on Trac 9478 is resolved:

```
sage: R2.<a> = QQ[]
sage: R3.<xi, x> = R2[]
```

```
sage: print latex(xi*x)
\xi x
```

lcmt (*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

list ()

Return a list that defines self. It is safe to change this.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: f.list()
[[3, [1, 2]], [2, [2, 3]], [4, [2, 1]]]
```

max_exp ()

Returns an ETuple containing the maximum exponents appearing. If there are no terms at all in the PolyDict, it returns None.

The 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.max_exp()
(2, 3)
sage: PolyDict({}).max_exp() # returns None
```

min_exp ()

Returns an ETuple containing the minimum exponents appearing. If there are no terms at all in the Poly-Dict, it returns None.

The 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*)

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())
4
```

poly_repr (*vars*, *atomic_exponents=True*, *atomic_coefficients=True*, *cmpfn=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):2, (1,2):3, (2,1):4})
sage: f.poly_repr(['a','WW'])
'2*a^2*WW^3 + 4*a^2*WW + 3*a*WW^2'
```

When atomic_exponents is False, the exponents are surrounded in parenthesis, since ^ has such high precedence.

```
# I've removed fractional exponent support in ETuple when moving to a sparse C integer array
#sage: f = PolyDict({(2/3,3,5):2, (1,2,1):3, (2,1,1):4}, force_int_exponents=False)
#sage: f.poly_repr(['a','b','c'], atomic_exponents=False)
#'4*a^(2)*b*c + 3*a*b^(2)*c + 2*a^(2/3)*b^(3)*c^(5)'
```

We check to make sure that when we are in characteristic two, we don't put negative signs on the generators.

```
sage: 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)})
sage: f.poly_repr(['x','y'])
'1.?*x^2*y^3 + 0.?*x*y^2'
```

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:

```
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, 3): 2, (0, 1): 4}
sage: f = PolyDict({(0,3):2, (0,2):3, (2,1):4})
sage: f.polynomial_coefficient([0,None])
PolyDict with representation {(0, 3): 2, (0, 2): 3}
```

scalar_lmult(*s*)

Left Scalar Multiplication

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: x,y=FreeMonoid(2,'x,y').gens()  # a strange object to live in a polydict, but non-comm
sage: f = PolyDict({(2,3):x})
sage: f.scalar_lmult(y)
PolyDict with representation {(2, 3): y*x}
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
```

```
sage: f.scalar_lmult(-2)
PolyDict with representation {(1, 2): -6, (2, 3): -4, (2, 1): -8}
sage: f.scalar_lmult(RIF(-1,1))
PolyDict with representation {(1, 2): 0.?e1, (2, 3): 0.?e1, (2, 1): 0.?e1}
```

scalar_rmult(*s*)

Right Scalar Multiplication

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: x,y=FreeMonoid(2,'x,y').gens()   # a strange object to live in a polydict, but non-comm
sage: f = PolyDict({(2,3):x})
sage: f.scalar_rmult(y)
PolyDict with representation {(2, 3): x*y}
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: f.scalar_rmult(-2)
PolyDict with representation {(1, 2): -6, (2, 3): -4, (2, 1): -8}
sage: f.scalar_rmult(RIF(-1,1))
PolyDict with representation {(1, 2): 0.?e1, (2, 3): 0.?e1, (2, 1): 0.?e1}
```

total_degree()

valuation(*x=None*)

sage.rings.polynomial.polydict.**make_ETuple**(*data*, *length*)

sage.rings.polynomial.polydict.**make_PolyDict**(*data*)

CHAPTER

FOUR

INFINITE POLYNOMIAL RINGS

4.1 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 $x_*, y_*, ...$ and infinitely many variables of the form $x_0, x_1, x_2, ..., y_0, y_1, y_2, ..., ...$. We refer to the natural number n as the *index* of the variable x_n.

INPUT:

- `R`, the base ring. It has to be a commutative ring, and in some applications it must even be a field
- `names`, a list of generator names. Generator names must be alpha-numeric.
- `order` (optional string). The default order is `'lex'` (lexicographic). `'deglex'` is degree lexicographic, and `'degrevlex'` (degree reverse lexicographic) is possible but discouraged.

Each generator `x` produces an infinite sequence of variables `x[1], x[2], ...` which are printed on screen as `x_1, x_2, ...` and are latex typeset as x_1, x_2. Then, the Infinite Polynomial Ring is formed by polynomials in these variables.

By default, the monomials are ordered lexicographically. Alternatively, degree (reverse) lexicographic ordering is possible as well. However, we do not guarantee that the computation of Groebner bases will terminate in this case.

In either case, the variables of a Infinite Polynomial Ring X are ordered according to the following rule:

`X.gen(i)[m] > X.gen(j)[n]` if and only if `i<j or (i==j and m>n)`

We provide a 'dense' and a 'sparse' implementation. In the dense implementation, the Infinite Polynomial Ring carries a finite polynomial ring that comprises *all* variables up to the maximal index that has been used so far. This is potentially a very big ring and may also comprise many variables that are not used.

In the sparse implementation, we try to keep the underlying finite polynomial rings small, using only those variables that are really needed. By default, we use the dense implementation, since it usually is much faster.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(ZZ, implementation='sparse')
sage: A.<alpha,beta> = InfinitePolynomialRing(QQ, order='deglex')

sage: f = x[5] + 2; f
x_5 + 2
```

```
sage: g = 3*y[1]; g
3*y_1
```

It has some advantages to have an underlying ring that is not univariate. Hence, we always have at least two variables:

```
sage: g._p.parent()
Multivariate Polynomial Ring in y_1, y_0 over Integer Ring

sage: f2 = alpha[5] + 2; f2
alpha_5 + 2
sage: g2 = 3*beta[1]; g2
3*beta_1
sage: A.polynomial_ring()
Multivariate Polynomial Ring in alpha_5, alpha_4, alpha_3, alpha_2, alpha_1, alpha_0, beta_5, beta_4,
```

Of course, we provide the usual polynomial arithmetic:

```
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:

```
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:

```
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):
...
CoercionException: Overlapping variables (('y',),['y_1', 'y_2']) are incompatible
```

If the type of monomial orderings (e.g., 'degrevlex' versus 'lex') or if the implementations don't match, there is no simplified construction available:

```
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
```

TESTS:

Infinite Polynomial Rings are part of Sage's coercion system. Hence, we can do arithmetic, so that the result lives in a ring into which all constituents coerce.

```
sage: R.<a,b> = InfinitePolynomialRing(ZZ)
sage: X.<x> = InfinitePolynomialRing(R)
sage: x[2]/2+(5/3)*a[3]*x[4] + 1
5/3*a_3*x_4 + 1/2*x_2 + 1

sage: R.<a,b> = InfinitePolynomialRing(ZZ,implementation='sparse')
sage: X.<x> = InfinitePolynomialRing(R)
sage: x[2]/2+(5/3)*a[3]*x[4] + 1
5/3*a_3*x_4 + 1/2*x_2 + 1

sage: R.<a,b> = InfinitePolynomialRing(ZZ,implementation='sparse')
sage: X.<x> = InfinitePolynomialRing(R,implementation='sparse')
sage: x[2]/2+(5/3)*a[3]*x[4] + 1
5/3*a_3*x_4 + 1/2*x_2 + 1

sage: R.<a,b> = InfinitePolynomialRing(ZZ)
sage: X.<x> = InfinitePolynomialRing(R,implementation='sparse')
sage: x[2]/2+(5/3)*a[3]*x[4] + 1
5/3*a_3*x_4 + 1/2*x_2 + 1
```

class `sage.rings.polynomial.infinite_polynomial_ring.`**`GenDictWithBasering`**(*parent*, *start*)

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 interprete 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 Ration
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*)

A dictionary-like class that is suitable for usage in `sage_eval`.

The generators of an Infinite Polynomial Ring are not variables. Variables of an Infinite Polynomial Ring are returned by indexing a generator. The purpose of this class is to return a variable of an Infinite Polynomial Ring, given its string representation.

EXAMPLES:

```
sage: R.<a,b> = InfinitePolynomialRing(ZZ)
sage: D = R.gens_dict() # indirect doctest
sage: D._D
[InfiniteGenDict defined by ['a', 'b'], {'1': 1}]
sage: D._D[0]['a_15']
a_15
sage: type(_)
<class 'sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_dense'>
sage: sage_eval('3*a_3*b_5-1/2*a_7', D._D[0])
-1/2*a_7 + 3*a_3*b_5
```

class `sage.rings.polynomial.infinite_polynomial_ring.`**`InfinitePolynomialGen`**(*parent*, *name*)

Bases: `sage.structure.sage_object.SageObject`

This class provides the object which is responsible for returning variables in an infinite polynomial ring (implemented in `__getitem__()`).

EXAMPLES:

```
sage: X.<x1,x2> = InfinitePolynomialRing(RR)
sage: x1
x1_*
sage: x1[5]
x1_5
sage: x1 == loads(dumps(x1))
True
```

class `sage.rings.polynomial.infinite_polynomial_ring.`**`InfinitePolynomialRingFactory`**

Bases: `sage.structure.factory.UniqueFactory`

A factory for creating infinite polynomial ring elements. It handles making sure that they are unique as well as handling pickling. For more details, see `UniqueFactory` and `infinite_polynomial_ring`.

EXAMPLES:

```
sage: A.<a> = InfinitePolynomialRing(QQ)
sage: B.<b> = InfinitePolynomialRing(A)
sage: B.construction()
[InfPoly{[a,b], "lex", "dense"}, Rational Field]
sage: R.<a,b> = InfinitePolynomialRing(QQ)
sage: R is B
True
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: X2.<x> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: X is X2
False

sage: X is loads(dumps(X))
True
```

`create_key`(*R*, *names=('x',)*, *order='lex'*, *implementation='dense'*)

Creates a key which uniquely defines the infinite polynomial ring.

TESTS:

```
sage: InfinitePolynomialRing.create_key(QQ, ('y1',))
(InfPoly{[y1], "lex", "dense"}(FractionField(...)), Integer Ring)
sage: _[0].all
[FractionField, InfPoly{[y1], "lex", "dense"}]
sage: InfinitePolynomialRing.create_key(QQ, names=['beta'], order='deglex', implementation='
(InfPoly{[beta], "deglex", "sparse"}(FractionField(...)), Integer Ring)
sage: _[0].all
[FractionField, InfPoly{[beta], "deglex", "sparse"}]
sage: InfinitePolynomialRing.create_key(QQ, names=['x','y'], implementation='dense')
(InfPoly{[x,y], "lex", "dense"}(FractionField(...)), Integer Ring)
sage: _[0].all
[FractionField, InfPoly{[x,y], "lex", "dense"}]
```

If no generator name is provided, a generator named 'x', lexicographic order and the dense implementation are assumed:

```
sage: InfinitePolynomialRing.create_key(QQ)
(InfPoly{[x], "lex", "dense"}(FractionField(...)), Integer Ring)
sage: _[0].all
[FractionField, InfPoly{[x], "lex", "dense"}]
```

If it is attempted to use no generator, a ValueError is raised:

```
sage: InfinitePolynomialRing.create_key(ZZ, names=[])
Traceback (most recent call last):
...
ValueError: Infinite Polynomial Rings must have at least one generator
```

create_object (*version*, *key*)

Returns the infinite polynomial ring corresponding to the key `key`.

TESTS:

```
sage: InfinitePolynomialRing.create_object('1.0', InfinitePolynomialRing.create_key(ZZ, ('x3
Infinite polynomial ring in x3 over Integer Ring
```

class `sage.rings.polynomial.infinite_polynomial_ring.`**`InfinitePolynomialRing_dense`**(*R*, *names*, *order*)

Bases: `sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse`

Dense implementation of Infinite Polynomial Rings

Compared with `InfinitePolynomialRing_sparse`, from which this class inherits, it keeps a polynomial ring that comprises all elements that have been created so far.

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`.

EXAMPLE:

```
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:

```
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
```

tensor_with_ring(*R*)

Return the tensor product of `self` with another ring.

INPUT:

`R` - a ring.

OUTPUT:

An infinite polynomial ring that, mathematically, can be seen as the tensor product of `self` with `R`.

NOTE:

It is required that the underlying ring of self coerces into `R`. Hence, the tensor product is in fact merely an extension of the base ring.

EXAMPLES:

```
sage: R.<a,b> = InfinitePolynomialRing(ZZ, implementation='sparse')
sage: R.tensor_with_ring(QQ)
Infinite polynomial ring in a, b over Rational Field
sage: R
Infinite polynomial ring in a, b over Integer Ring
```

The following tests against a bug that was fixed at trac ticket #10468:

```
sage: R.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: R.tensor_with_ring(QQ) is R
True
```

class sage.rings.polynomial.infinite_polynomial_ring.**InfinitePolynomialRing_sparse**(*R*, *names*, *order*)

Bases: `sage.rings.ring.CommutativeRing`

Sparse implementation of Infinite Polynomial Rings.

An Infinite Polynomial Ring with generators $x_*, y_*, ...$ over a field F is a free commutative F-algebra generated by $x_0, x_1, x_2, ..., y_0, y_1, y_2, ..., ...$ and is equipped with a permutation action on the generators, namely $x_n^P = x_{P(n)}, y_n^P = y_{P(n)}, ...$ for any permutation P (note that variables of index zero are invariant under such permutation).

It is known that any permutation invariant ideal in an Infinite Polynomial Ring is finitely generated modulo the permutation action – see `SymmetricIdeal` for more details.

Usually, an instance of this class is created using `InfinitePolynomialRing` with the optional parameter `implementation='sparse'`. This takes care of uniqueness of parent structures. However, a direct construction is possible, in principle:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: Y.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: X is Y
True
sage: from sage.rings.polynomial.infinite_polynomial_ring import InfinitePolynomialRing_sparse
sage: Z = InfinitePolynomialRing_sparse(QQ, ['x','y'], 'lex')
```

Nevertheless, since infinite polynomial rings are supposed to be unique parent structures, they do not evaluate equal.

sage: Z == X False

The last parameter ('lex' in the above example) can also be 'deglex' or 'degrevlex'; this would result in an Infinite Polynomial Ring in degree lexicographic or degree reverse lexicographic order.

See `infinite_polynomial_ring` for more details.

characteristic()

Return the characteristic of the base field.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(GF(25,'a'))
sage: X
Infinite polynomial ring in x, y over Finite Field in a of size 5^2
sage: X.characteristic()
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`.

EXAMPLE:

```
sage: R.<x,y> = InfinitePolynomialRing(GF(5))
sage: R.construction()
[InfPoly{[x,y], "lex", "dense"}, Finite Field of size 5]
```

gen(*i=None*)

Returns the i^{th} 'generator' (see the description in `ngens()`) of this infinite polynomial ring.

EXAMPLES:

```
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
```

is_field(**args, **kwds*)

Return `False`: Since Infinite Polynomial Rings must have at least one generator, they have infinitely many variables and thus never are fields.

EXAMPLES:

```
sage: R.<x, y> = InfinitePolynomialRing(QQ)
sage: R.is_field()
False
```

TESTS:

```
sage: R = InfinitePolynomialRing(GF(2))
sage: R
Infinite polynomial ring in x over Finite Field of size 2
sage: R.is_field()
False
```

Ticket #9443:

```
sage: W = PowerSeriesRing(InfinitePolynomialRing(QQ,'a'),'x')
sage: W.is_field()
False
```

is_integral_domain(**args*, ***kwds*)

An infinite polynomial ring is an integral domain if and only if the base ring is. Arguments are passed to is_integral_domain method of base ring.

EXAMPLES:

```
sage: R.<x, y> = InfinitePolynomialRing(QQ)
sage: R.is_integral_domain()
True
```

TESTS:

Ticket #9443:

```
sage: W = PolynomialRing(InfinitePolynomialRing(QQ,'a'),2,'x,y')
sage: W.is_integral_domain()
True
```

is_noetherian(**args*, ***kwds*)

Return `False`, since polynomial rings in infinitely many variables are never Noetherian rings.

Note, however, that they are noetherian modules over the group ring of the symmetric group of the natural numbers

EXAMPLES:

```
sage: R.<x> = InfinitePolynomialRing(QQ)
sage: R.is_noetherian()
False
```

krull_dimension(**args*, ***kwds*)

Return `Infinity`, since polynomial rings in infinitely many variables have infinite Krull dimension.

EXAMPLES:

```
sage: R.<x, y> = InfinitePolynomialRing(QQ)
sage: R.krull_dimension()
+Infinity
```

ngens()

Returns the number of generators for this ring. Since there are countably infinitely many variables in this polynomial ring, by 'generators' we mean the number of infinite families of variables. See `infinite_polynomial_ring` for more details.

EXAMPLES:

```
sage: X.<x> = InfinitePolynomialRing(ZZ)
sage: X.ngens()
1

sage: X.<x1,x2> = InfinitePolynomialRing(QQ)
sage: X.ngens()
2
```

one()

TESTS:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: X.one()
1
```

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 trac ticket #10468:

```
sage: R.<x,y> = InfinitePolynomialRing(QQ)
sage: R.tensor_with_ring(QQ) is R
True
```

varname_cmp(*x*, *y*)

Comparison of two variable names.

INPUT:

`x,y` – two strings of the form `a+'_'+str(n)`, where a is the name of a generator, and n is an integer

RETURN:

-1,0,1 if x<y, x==y, x>y, respectively

THEORY:

The order is defined as follows: x<y $\iff$ 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_cmp('alpha_1','beta_10')
1
sage: X.varname_cmp('beta_1','alpha_10')
-1
sage: X.varname_cmp('alpha_1','alpha_10')
-1
```

4.2 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]` $\iff$ `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:

```
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`.

TESTS:

We test whether coercion works, even in complicated cases in which finite polynomial rings are merged with infinite polynomial rings:

```
sage: A.<a> = InfinitePolynomialRing(ZZ,implementation='sparse',order='degrevlex')
sage: B.<b_2,b_1> = A[]
sage: C.<b,c> = InfinitePolynomialRing(B,order='degrevlex')
sage: C
Infinite polynomial ring in b, c over Infinite polynomial ring in a over Integer Ring
sage: 1/2*b_1*a[4]+c[3]
1/2*a_4*b_1 + c_3
```

`sage.rings.polynomial.infinite_polynomial_element.`**`InfinitePolynomial`**(A, p)

Create an element of a Polynomial Ring with a Countably Infinite Number of Variables.

Usually, an InfinitePolynomial is obtained by using the generators of an Infinite Polynomial Ring (see `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:

```
sage: InfinitePolynomial(X, (alpha_1+alpha_2)^2)
Traceback (most recent call last):
...
TypeError: Could not find a mapping of the passed element to this ring.
```

When extending the underlying polynomial ring, the construction of an infinite polynomial works:

```
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:

```
sage: Y.<alpha,beta> = InfinitePolynomialRing(GF(2), implementation='sparse')
sage: a = (alpha_1+alpha_2)^2
sage: InfinitePolynomial(Y, a)
alpha_1^2 + 2*alpha_1*alpha_2 + alpha_2^2
```

However, it is checked when doing a conversion:

```
sage: Y(a)
alpha_2^2 + alpha_1^2
```

class `sage.rings.polynomial.infinite_polynomial_element.`**`InfinitePolynomial_dense`**`(`*A*, *p*`)`

Bases: `sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse`

Element of a dense Polynomial Ring with a Countably Infinite Number of Variables.

INPUT:

- `A` – an Infinite Polynomial Ring in dense implementation
- `p` – a *classical* polynomial that can be interpreted in `A`.

Of course, one should not directly invoke this class, but rather construct elements of `A` in the usual way.

This class inherits from `InfinitePolynomial_sparse`. See there for a description of the methods.

class `sage.rings.polynomial.infinite_polynomial_element.`**`InfinitePolynomial_sparse`**`(`*A*, *p*`)`

Bases: `sage.structure.element.RingElement`

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: 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 R
```

`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:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: a = 2*x[0]*x[1] + x[1] + x[2]
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]*x[1])
2
```

We can also pass in a dictionary:

```
sage: a.coefficient({x[0]:1, x[1]:1})
2
```

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: p = x[30]*y[1]^3*x[1]^2+2*x[10]*y[30]
sage: sorted(p.footprint().items())
[(1, [2, 3]), (30, [1, 0])]
```

TESTS:

This is a test whether it also works when the underlying polynomial ring is not implemented in libsingular:

```
sage: X.<x> = InfinitePolynomialRing(ZZ)
sage: Y.<y,z> = X[]
sage: Z.<a> = InfinitePolynomialRing(Y)
sage: Z
Infinite polynomial ring in a over Multivariate Polynomial Ring in y, z over Infinite polyno
sage: type(Z._P)
<class 'sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict_with_category'>
sage: p = a[12]^3*a[2]^7*a[4] + a[4]*a[2]
sage: sorted(p.footprint().items())
[(2, [7]), (4, [1]), (12, [3])]
```

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_unit()

Answers whether self is a unit

EXAMPLES:

```
sage: R1.<x,y> = InfinitePolynomialRing(ZZ)
sage: R2.<x,y> = InfinitePolynomialRing(QQ)
sage: p = 1 + x[2]
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
```

TESTS:

```
sage: R.<x> = InfinitePolynomialRing(ZZ.quotient_ring(8))
sage: [R(i).is_unit() for i in range(8)]
[False, True, False, True, False, True, False, True]
```

lc()

The coefficient of the leading term of self.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p = 2*x[10]*y[30]+3*x[10]*y[1]^3*x[1]^2
sage: p.lc()
3
```

lm()

The leading monomial of self.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p = 2*x[10]*y[30]+x[10]*y[1]^3*x[1]^2
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 = 2*x[10]*y[30]+3*x[10]*y[1]^3*x[1]^2
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: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p=x[1]^2+y[2]^2+x[1]*x[2]*y[3]+x[1]*y[4]
sage: p.max_index()
4
sage: x[0].max_index()
0
sage: X(10).max_index()
-1
```

polynomial()

Return the underlying polynomial.

EXAMPLES:

```
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 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: p = y[1]^2*y[3]+y[2]*x[3]^3
sage: p.reduce([y[2]*x[1]^2])
x_3^3*y_2 + y_3*y_1^2
```

The preceding is correct: If a permutation turns `y[2]*x[1]^2` into a factor of the leading monomial `y[2]*x[3]^3` of `p`, then it interchanges the variable indices 1 and 2; this is not allowed in a symmetric reduction. However, reduction by `y[1]*x[2]^2` works, since one can change variable index 1 into 2 and 2 into 3:

```
sage: p.reduce([y[1]*x[2]^2])
y_3*y_1^2
```

The next example shows that tail reduction is not done, unless it is explicitly advised. The input can also be a Symmetric Ideal:

```
sage: I = (y[3])*X
sage: p.reduce(I)
x_3^3*y_2 + y_3*y_1^2
sage: p.reduce(I, tailreduce=True)
x_3^3*y_2
```

Last, we demonstrate the `report` option:

```
sage: p=x[1]^2+y[2]^2+x[1]*x[2]*y[3]+x[1]*y[4]
sage: p.reduce(I, tailreduce=True, report=True)
:T[2]:>
>
x_1^2 + y_2^2
```

The output ':' means that there was one reduction of the leading monomial. 'T[2]' means that a tail reduction was performed on a polynomial with two terms. At '>', one round of the reduction process is finished (there could only be several non-trivial rounds if I was generated by more than one polynomial).

ring()

The ring which `self` belongs to.

This is the same as `self.parent()`.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(ZZ,implementation='sparse')
sage: p = x[100]*y[1]^3*x[1]^2+2*x[10]*y[30]
sage: p.ring()
Infinite polynomial ring in x, y over Integer Ring
```

squeezed()

Reduce the variable indices occurring in `self`.

OUTPUT:

Apply a permutation to `self` that does not change the order of the variable indices of `self` but squeezes them into the range 1,2,...

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ,implementation='sparse')
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 $v_{n \cdot k}$ for all generators v_* occurring in self.

EXAMPLES:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: a = x[0] + x[1] + x[2]
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
```

TESTS:

The following would hardly work in a dense implementation, because an underlying polynomial ring with 6001 variables would be created. This is avoided in the sparse implementation:

```
sage: X.<x> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: a = x[2] + x[3]
sage: a.stretch(2000)
x_6000 + x_4000
```

symmetric_cancellation_order(*other*)

Comparison of leading terms by Symmetric Cancellation Order, $<_{sc}$.

INPUT:

self, other – two Infinite Polynomials

ASSUMPTION:

Both Infinite Polynomials are non-zero.

OUTPUT:

`(c, 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
- sigma is a permutation witnessing `self` $<_{sc}$ `other` (resp. `self` $>_{sc}$ `other`) or is 1 if `self.lm()==other.lm()`
- w is 1 or is a term so that `w*self.lt()^sigma == other.lt()` if $c \leq 0$, and `w*other.lt()^sigma == 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: 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[1])
```

```
(None, 1, 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: 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: X.<x> = InfinitePolynomialRing(QQ)
sage: p = x[1] + x[2] - 2*x[1]*x[3]
sage: p.variables()
(x_3, x_2, x_1)
sage: x[1].variables()
(x_1,)
sage: X(1).variables()
()
```

4.3 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: R.<x> = InfinitePolynomialRing(QQ)
sage: I = R.ideal([x[1]*x[2] + x[3]])
```

Note that `I` is not a symmetric Groebner basis:

```
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: p = x[3]*x[1]+x[2]^2+3
sage: Q(p)
-2*x_1 + 3
```

By the second generator of `G`, variable x_n is equal to x_1 for any positive integer n. By the first generator of `G`, x_1^3 is equal to x_1 in `Q`. Indeed, we have

```
sage: Q(p)*x[2] == Q(p)*x[1]*x[3]*x[5]
True
```

class sage.rings.polynomial.symmetric_ideal.**SymmetricIdeal**(*ring*, *gens*, *coerce=True*)

Bases: sage.rings.ideal.Ideal_generic

Ideal in an Infinite Polynomial Ring, invariant under permutation of variable indices

THEORY:

An Infinite Polynomial Ring with finitely many generators $x_*, y_*, ...$ over a field F is a free commutative F-algebra generated by infinitely many 'variables' $x_0, x_1, x_2, ..., y_0, y_1, y_2, ...$. We refer to the natural number n as the *index* of the variable x_n. See more detailed description at infinite_polynomial_ring

Infinite Polynomial Rings are equipped with a permutation action by permuting positive variable indices, i.e., $x_n^P = x_{P(n)}, y_n^P = y_{P(n)}, ...$ for any permutation P. Note that the variables $x_0, y_0, ...$ of index zero are invariant under that action.

A *Symmetric Ideal* is an ideal in an infinite polynomial ring X that is invariant under the permutation action. In other words, if $\mathfrak{S}_\infty$ 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 $x_1 < x_2 < x_3...$ 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 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:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: I = [x[1]*y[2]*y[1] + 2*x[1]*y[2]]*X
sage: I == loads(dumps(I))
True
sage: latex(I)
\left(x_{1} y_{2} y_{1} + 2 x_{1} y_{2}\right)\Bold{Q}[x_{\ast}, y_{\ast}][\mathfrak{S}_{\infty}
```

The default ordering is lexicographic. We now compute a Groebner basis:

```
sage: J = I.groebner_basis() ; J    # about 3 seconds
[x_1*y_2*y_1 + 2*x_1*y_2, x_2*y_2*y_1 + 2*x_2*y_1, x_2*x_1*y_1^2 + 2*x_2*x_1*y_1, x_2*x_1*y_2 -
```

Note that even though the symmetric ideal can be generated by a single polynomial, its reduced symmetric Groebner basis comprises four elements. Ideal membership in I can now be tested by commuting symmetric reduction modulo J:

```
sage: I.reduce(J)
Symmetric Ideal (0) of Infinite polynomial ring in x, y over Rational Field
```

The Groebner basis is not point-wise invariant under permutation:

```
sage: P=Permutation([2, 1])
sage: J[2]
```

```
x_2*x_1*y_1^2 + 2*x_2*x_1*y_1
sage: J[2]^P
x_2*x_1*y_2^2 + 2*x_2*x_1*y_2
sage: J[2]^P in J
False
```

However, any element of `J` has symmetric reduction zero even after applying a permutation. This even holds when the permutations involve higher variable indices than the ones occuring in `J`:

```
sage: [[(p^P).reduce(J) for p in J] for P in Permutations(3)]
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Since `I` is not a Groebner basis, it is no surprise that it can not detect ideal membership:

```
sage: [p.reduce(I) for p in J]
[0, x_2*y_2*y_1 + 2*x_2*y_1, x_2*x_1*y_1^2 + 2*x_2*x_1*y_1, x_2*x_1*y_2 - x_2*x_1*y_1]
```

Note that we give no guarantee that the computation of a symmetric Groebner basis will terminate in any order different from lexicographic.

When multiplying Symmetric Ideals or raising them to some integer power, the permutation action is taken into account, so that the product is indeed the product of ideals in the mathematical sense.

```
sage: I=X*(x[1])
sage: I*I
Symmetric Ideal (x_1^2, x_2*x_1) of Infinite polynomial ring in x, y over Rational Field
sage: I^3
Symmetric Ideal (x_1^3, x_2*x_1^2, x_2^2*x_1, x_3*x_2*x_1) of Infinite polynomial ring in x, y o
sage: I*I == X*(x[1]^2)
False
```

groebner_basis(*tailreduce=False*, *reduced=True*, *algorithm=None*, *report=None*, *use_full_group=False*)

Return a symmetric Groebner basis (type `Sequence`) of `self`.

INPUT:

- `tailreduce` – (bool, default `False`) If True, use tail reduction in intermediate computations
- `reduced` – (bool, default `True`) If `True`, return the reduced normalised symmetric Groebner basis.
- `algorithm` – (string, default `None`) Determine the algorithm (see below for available algorithms).
- `report` – (object, default `None`) If not `None`, print information on the progress of computation.
- `use_full_group` – (bool, default `False`) If `True` then proceed as originally suggested by [AB2008]. Our default method should be faster; see `symmetrisation()` for more details.

The computation of symmetric Groebner bases also involves the computation of *classical* Groebner bases, i.e., of Groebner bases for ideals in polynomial rings with finitely many variables. For these computations, Sage provides the following ALGORITHMS:

'' autoselect (default)

'singular:groebner' Singular's `groebner` command

'singular:std' Singular's `std` command

'singular:stdhilb' Singular's `stdhib` 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()
[x_1]
sage: I2 = X*(y[1]^2*y[3]+y[1]*x[3])
sage: I2.groebner_basis()
[x_1*y_2 + y_2^2*y_1, x_2*y_1 + y_2*y_1^2]
```

Note that a symmetric Groebner basis of a principal ideal is not necessarily formed by a single polynomial.

When using the algorithm originally suggested by Aschenbrenner and Hillar, the result is the same, but the computation takes much longer:

```
sage: I2.groebner_basis(use_full_group=True)
[x_1*y_2 + y_2^2*y_1, x_2*y_1 + y_2*y_1^2]
```

Last, we demonstrate how the report on the progress of computations looks like:

```
sage: I1.groebner_basis(report=True, reduced=True)
Symmetric interreduction
[1/2]  >
[2/2] :>
[1/2]  >
[2/2]  >
Symmetrise 2 polynomials at level 2
Apply permutations
>
>
Symmetric interreduction
[1/3]  >
[2/3]  >
[3/3] :>
-> 0
[1/2]  >
[2/2]  >
Symmetrisation done
Classical Groebner basis
-> 2 generators
Symmetric interreduction
[1/2]  >
[2/2]  >
```

```
Symmetrise 2 polynomials at level 3
Apply permutations
>
>
:>
::>
:>
::>
Symmetric interreduction
[1/4]  >
[2/4] :>
-> 0
[3/4] ::>
-> 0
[4/4] :>
-> 0
[1/1]  >
Apply permutations
:>
:>
:>
Symmetric interreduction
[1/1]  >
Classical Groebner basis
-> 1 generators
Symmetric interreduction
[1/1]  >
Symmetrise 1 polynomials at level 4
Apply permutations
>
:>
:>
>
:>
:>
Symmetric interreduction
[1/2]  >
[2/2] :>
-> 0
[1/1]  >
Symmetric interreduction
[1/1]  >
[x_1]
```

The Aschenbrenner-Hillar algorithm is only guaranteed to work if the base ring is a field. So, we raise a TypeError if this is not the case:

```
sage: R.<x,y> = InfinitePolynomialRing(ZZ)
sage: I = R*[x[1]+x[2],y[1]]
sage: I.groebner_basis()
Traceback (most recent call last):
...
TypeError: The base ring (= Integer Ring) must be a field
```

TESTS:

In an earlier version, the following examples failed:

```
sage: X.<y,z> = InfinitePolynomialRing(GF(5),order='degrevlex')
sage: I = ['-2*y_0^2 + 2*z_0^2 + 1', '-y_0^2 + 2*y_0*z_0 - 2*z_0^2 - 2*z_0 - 1', 'y_0*z_0 +
sage: I.groebner_basis()
[1]

sage: Y.<x,y> = InfinitePolynomialRing(GF(3), order='degrevlex', implementation='sparse')
sage: I = ['-y_3']*Y
sage: I.groebner_basis()
[y_1]
```

interreduced_basis()

A fully symmetrically reduced generating set (type `Sequence`) of self.

This does essentially the same as `interreduction()` with the option 'tailreduce', but it returns a `Sequence` rather than a `SymmetricIdeal`.

EXAMPLES:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I=X*(x[1]+x[2],x[1]*x[2])
sage: I.interreduced_basis()
[-x_1^2, x_2 + x_1]
```

interreduction(*tailreduce=True*, *sorted=False*, *report=None*, *RStrat=None*)

Return symmetrically interreduced form of self

INPUT:

- `tailreduce` – (bool, default `True`) If True, the interreduction is also performed on the non-leading monomials.
- `sorted` – (bool, default `False`) If True, it is assumed that the generators of self are already increasingly sorted.
- `report` – (object, default `None`) If not None, some information on the progress of computation is printed
- `RStrat` – (`SymmetricReductionStrategy`, default `None`) A reduction strategy to which the polynomials resulting from the interreduction will be added. If `RStrat` already contains some polynomials, they will be used in the interreduction. The effect is to compute in a quotient ring.

OUTPUT:

A Symmetric Ideal J (sorted list of generators) coinciding with self as an ideal, so that any generator is symmetrically reduced w.r.t. the other generators. Note that the leading coefficients of the result are not necessarily 1.

EXAMPLES:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I=X*(x[1]+x[2],x[1]*x[2])
sage: I.interreduction()
Symmetric Ideal (-x_1^2, x_2 + x_1) of Infinite polynomial ring in x over Rational Field
```

Here, we show the `report` option:

```
sage: I.interreduction(report=True)
Symmetric interreduction
[1/2]  >
[2/2] :>
[1/2]  >
[2/2] T[1]>
```

```
>
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:

```
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: R = SymmetricReductionStrategy(X)
sage: R
Symmetric Reduction Strategy in Infinite polynomial ring in x over Rational Field
sage: I.interreduction(RStrat=R)
Symmetric Ideal (-x_1^2, x_2 + x_1) of Infinite polynomial ring in x over Rational Field
sage: R
Symmetric Reduction Strategy in Infinite polynomial ring in x over Rational Field, modulo
    x_1^2,
    x_2 + x_1
sage: R = SymmetricReductionStrategy(X,[x[1]^2])
sage: I.interreduction(RStrat=R)
Symmetric Ideal (x_2 + x_1) of Infinite polynomial ring in x over Rational Field
```

is_maximal()

Answers whether self is a maximal ideal.

ASSUMPTION:

self is defined by a symmetric Groebner basis.

NOTE:

It is not checked whether self is in fact a symmetric Groebner basis. A wrong answer can result if this assumption does not hold. A 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.ideal([x[1]+y[2], x[2]-y[1]])
sage: I = R*I.groebner_basis()
sage: I
Symmetric Ideal (y_1, x_1) of Infinite polynomial ring in x, y over Rational Field
sage: I = R.ideal([x[1]+y[2], x[2]-y[1]])
sage: I.is_maximal()
False
```

The preceding answer is wrong, since it is not the case that I is given by a symmetric Groebner basis:

```
sage: I = R*I.groebner_basis()
sage: I
Symmetric Ideal (y_1, x_1) of Infinite polynomial ring in x, y over Rational Field
sage: I.is_maximal()
True
```

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 Fiel
```

reduce (*I*, *tailreduce=False*)

Symmetric reduction of self by another Symmetric Ideal or list of Infinite Polynomials, or symmetric reduction of a given Infinite Polynomial by self.

INPUT:

- `I` – an Infinite Polynomial, or a Symmetric Ideal or a list of Infinite Polynomials.
- `tailreduce` – (bool, default `False`) If `True`, the non-leading terms will be reduced as well.

OUTPUT:

Symmetric reduction of `self` with respect to `I`.

THEORY:

Reduction of an element p of an Infinite Polynomial Ring X by some other element q means the following:

1. Let M and N be the leading terms of p and q.
2. Test whether there is a permutation P that does not does not diminish the variable indices occurring in N and preserves their order, so that there is some term $T \in X$ with $TN^P = M$. If there is no such permutation, return p
3. Replace p by $p - Tq^P$ and continue with step 1.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: I = X*(y[1]^2*y[3]+y[1]*x[3]^2)
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 Fi
```

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]])
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 Fi
sage: I.reduce(J, tailreduce=True)
Symmetric Ideal (x_3^2*y_1) of Infinite polynomial ring in x, y over Rational Field
```

squeezed ()

Reduce the variable indices occurring in `self`.

OUTPUT:

A Symmetric Ideal whose generators are the result of applying `squeezed()` to the generators of `self`.

NOTE:

The output describes the same Symmetric Ideal as `self`.

EXAMPLES:

```
sage: 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()
[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 self is symmetrically reducible, where by default `N` is the maximal variable index that occurs in the generators of `self.interreduction().squeezed()`.

NOTE:

If `I` is a symmetric ideal whose generators are monomials, then `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: X.<x> = InfinitePolynomialRing(QQ)
sage: I = X*(x[1]+x[2], x[1]*x[2])
sage: I.symmetrisation()
Symmetric Ideal (-x_1^2, x_2 + x_1) of Infinite polynomial ring in x over Rational Field
sage: I.symmetrisation(N=3)
Symmetric Ideal (-2*x_1) of Infinite polynomial ring in x over Rational Field
sage: I.symmetrisation(N=3, use_full_group=True)
Symmetric Ideal (-2*x_1) of Infinite polynomial ring in x over Rational Field
```

4.4 Symmetric Reduction of Infinite Polynomials

`SymmetricReductionStrategy` provides a framework for efficient symmetric reduction of Infinite Polynomials, see `infinite_polynomial_element`.

AUTHORS:

- Simon King <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 does not diminish the variable indices occurring in N and preserves their order, so that there is some term $T \in X$ with $TN^P = M$. If there is no such permutation, return p.
3. Replace p by $p - Tq^P$ and continue with step 1.

When reducing one polynomial p with respect to a list L of other polynomials, there usually is a choice of order on which the efficiency crucially depends. Also it helps to modify the polynomials on the list in order to simplify the basic reduction steps.

The preparation of L may be expensive. Hence, if the same list is used many times then it is reasonable to perform the preparation only once. This is the background of `SymmetricReductionStrategy`.

Our current strategy is to keep the number of terms in the polynomials as small as possible. For this, we sort L by increasing number of terms. If several elements of L allow for a reduction of p, we chose 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: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p = y[1]*y[3]+y[1]^2*x[3]
```

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: 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: 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)
y_3*y_1
```

The next example shows that tail reduction is not done, unless it is explicitly advised:

```
sage: S.reduce(x[3] + 2*x[2]*y[1]^2 + 3*y[2]^2*x[1])
x_3 + 2*x_2*y_1^2 + 3*x_1*y_2^2
sage: S.tailreduce(x[3] + 2*x[2]*y[1]^2 + 3*y[2]^2*x[1])
x_3
```

However, it is possible to ask for tailreduction already when the Symmetric Reduction Strategy is created:

```
sage: S2 = SymmetricReductionStrategy(X, [y[2]^2*x[1],y[1]^2*x[2]], tailreduce=True)
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
sage: S2.reduce(x[3] + 2*x[2]*y[1]^2 + 3*y[2]^2*x[1])
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
sage: S = SymmetricReductionStrategy(X, [y[2]^2*y[1],y[1]^2*y[2]], good_input=True)
sage: S.reduce(y[3] + 2*y[2]*y[1]^2 + 3*y[2]^2*y[1])
y_3 + 3*y_2^2*y_1 + 2*y_2*y_1^2
sage: S.tailreduce(y[3] + 2*y[2]*y[1]^2 + 3*y[2]^2*y[1])
y_3
```

`add_generator`(*p*, *good_input=None*)

Add another polynomial to `self`.

INPUT:

- `p` – An element of the underlying infinite polynomial ring.
- `good_input` – (bool, default `None`) If `True`, it is assumed that `p` is reduced with respect to `self`. Otherwise, this reduction will be done first (which may cost some time).

NOTE:

Previously added polynomials may be modified. All input is prepared in view of an efficient symmetric reduction.

EXAMPLES:

```
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: S = SymmetricReductionStrategy(X)
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field
sage: S.add_generator(y[3] + y[1]*(x[3]+x[1]))
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo
    x_3*y_1 + x_1*y_1 + y_3
```

Note that the first added polynomial will be simplified when adding a suitable second polynomial:

```
sage: S.add_generator(x[2]+x[1])
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo
    y_3,
    x_2 + x_1
```

By default, reduction is applied to any newly added polynomial. This can be avoided by specifying the optional parameter 'good_input':

```
sage: S.add_generator(y[2]+y[1]*x[2])
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo
    y_3,
    x_1*y_1 - y_2,
    x_2 + x_1
sage: S.reduce(x[3]+x[2])
-2*x_1
sage: S.add_generator(x[3]+x[2], good_input=True)
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo
    y_3,
    x_3 + x_2,
    x_1*y_1 - y_2,
    x_2 + x_1
```

In the previous example, `x[3] + x[2]` is added without being reduced to zero.

gens()

Return the list of Infinite Polynomials modulo which self reduces.

EXAMPLES:

```
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: S = SymmetricReductionStrategy(X, [y[2]^2*y[1],y[1]^2*y[2]])
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field, modulo
    y_2*y_1^2,
    y_2^2*y_1
sage: S.gens()
[y_2*y_1^2, y_2^2*y_1]
```

reduce (*p*, *notail=False*, *report=None*)

Symmetric reduction of an infinite polynomial.

INPUT:

- p – an element of the underlying infinite polynomial ring.
- notail – (bool, default False) If True, tail reduction is avoided (but there is no guarantee that there will be no tail reduction at all).
- report – (object, default None) If not None, print information on the progress of the computation.

OUTPUT:

Reduction of p with respect to self.

NOTE:

If tail reduction shall be forced, use tailreduce().

EXAMPLES:

```
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: S = SymmetricReductionStrategy(X, [y[3]], tailreduce=True)
sage: S.reduce(y[4]*x[1] + y[1]*x[4])
x_4*y_1
sage: S.reduce(y[4]*x[1] + y[1]*x[4], notail=True)
x_4*y_1 + x_1*y_4
```

Last, we demonstrate the 'report' option:

```
sage: S = SymmetricReductionStrategy(X, [x[2]+y[1],x[2]*y[3]+x[1]*y[2]+y[4],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_3*y_1
sage: S.reduce(x[3] + x[1]*y[3] + x[1]*y[1],report=True)
:::>
x_1*y_1 + y_4 - y_3*y_1 - y_1
```

Each ':' indicates that one reduction of the leading monomial was performed. Eventually, the '>' indicates that the computation is finished.

reset ()

Remove all polynomials from self.

EXAMPLES:

```
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: S = SymmetricReductionStrategy(X, [y[2]^2*y[1],y[1]^2*y[2]])
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field, modulo
    y_2*y_1^2,
    y_2^2*y_1
sage: S.reset()
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field
```

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:

```
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: S = SymmetricReductionStrategy(X, [y[2]^2*y[1],y[1]^2*y[2]])
sage: R = SymmetricReductionStrategy(X)
sage: R.setgens(S.gens())
sage: R
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field, modulo
    y_2*y_1^2,
    y_2^2*y_1
sage: R.gens() is S.gens()
False
sage: R.gens() == S.gens()
True
```

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:

```
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: S = SymmetricReductionStrategy(X, [y[3]])
sage: S.reduce(y[4]*x[1] + y[1]*x[4])
x_4*y_1 + x_1*y_4
sage: S.tailreduce(y[4]*x[1] + y[1]*x[4])
x_4*y_1
```

Last, we demonstrate the 'report' option:

```
sage: S = SymmetricReductionStrategy(X, [x[2]+y[1],x[2]*x[3]+x[1]*y[2]+y[4],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
sage: S.tailreduce(x[3] + x[1]*y[3] + x[1]*y[1],report=True)
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

FIVE

LAURENT POLYNOMIALS AND POLYNOMIAL RINGS

5.1 Ring of Laurent Polynomials

If R is a commutative ring, then the ring of Laurent polynomials in n variables over R is $R[x_1^{\pm 1}, x_2^{\pm 1}, \ldots, x_n^{\pm 1}]$. We implement it as a quotient ring

$$R[x_1, y_1, x_2, y_2, \ldots, x_n, y_n]/(x_1 y_1 - 1, x_2 y_2 - 1, \ldots, x_n y_n - 1).$$

TESTS:

```
sage: P.<q> = LaurentPolynomialRing(QQ)
sage: qi = q^(-1)
sage: qi in P
True
sage: P(qi)
q^-1

sage: A.<Y> = QQ[]
sage: R.<X> = LaurentPolynomialRing(A)
sage: matrix(R,2,2,[X,0,0,1])
[X 0]
[0 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, arg1=None, arg2=None, sparse=False, order='degrevlex', names=None, name=None*`)`

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

There are four ways to call the Laurent polynomial ring constructor:

1. `LaurentPolynomialRing(base_ring, name, sparse=False)`
2. `LaurentPolynomialRing(base_ring, names, order='degrevlex')`
3. `LaurentPolynomialRing(base_ring, name, n, order='degrevlex')`

4. `LaurentPolynomialRing(base_ring, n, name, order='degrevlex')`

The optional arguments sparse and order *must* be explicitly named, and the other arguments must be given positionally.

INPUT:

- `base_ring` – a commutative ring
- `name` – a string
- `names` – a list or tuple of names, or a comma separated string
- `n` – a positive integer
- `sparse` – bool (default: False), whether or not elements are sparse
- `order` – string or `TermOrder`, e.g.,
  - `'degrevlex'` (default) – degree reverse lexicographic
  - `'lex'` – lexicographic
  - `'deglex'` – degree lexicographic
  - `TermOrder('deglex',3) + TermOrder('deglex',3)` – block ordering

OUTPUT:

`LaurentPolynomialRing(base_ring, name, sparse=False)` returns a univariate Laurent polynomial ring; all other input formats return a multivariate Laurent polynomial ring.

UNIQUENESS and IMMUTABILITY: In Sage there is exactly one single-variate Laurent polynomial ring over each base ring in each choice of variable and sparseness. There is also exactly one multivariate Laurent polynomial ring over each base ring for each choice of names of variables and term order.

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

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

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

EXAMPLES:

1. `LaurentPolynomialRing(base_ring, name, sparse=False)`

   ```
   sage: LaurentPolynomialRing(QQ, 'w')
   Univariate Laurent Polynomial Ring in w over Rational Field
   ```

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

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

   You must specify a name:

```
sage: LaurentPolynomialRing(QQ)
Traceback (most recent call last):
...
TypeError: You must specify the names of the variables.

sage: R.<abc> = LaurentPolynomialRing(QQ, sparse=True); R
Univariate Laurent Polynomial Ring in abc over Rational Field

sage: R.<w> = LaurentPolynomialRing(PolynomialRing(GF(7),'k')); R
Univariate Laurent Polynomial Ring in w over Univariate Polynomial Ring in k over Finite Fie
```

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: 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.

```
sage: LaurentPolynomialRing(QQ, 'x', 10)
Multivariate Laurent Polynomial Ring in x0, x1, x2, x3, x4, x5, x6, x7, x8, x9 over Rational

sage: LaurentPolynomialRing(GF(7), 'y', 5)
```

```
Multivariate Laurent Polynomial Ring in y0, y1, y2, y3, y4 over Finite Field of size 7

sage: LaurentPolynomialRing(QQ, 'y', 3, sparse=True)
Multivariate Laurent Polynomial Ring in y0, y1, y2 over Rational Field
```

By calling the `inject_variables()` method, all those variable names are available for interactive use:

```
sage: R = LaurentPolynomialRing(GF(7),15,'w'); R
Multivariate Laurent Polynomial Ring in w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w1
sage: R.inject_variables()
Defining w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13, w14
sage: (w0 + 2*w8 + w13)^2
w0^2 + 4*w0*w8 + 4*w8^2 + 2*w0*w13 + 4*w8*w13 + w13^2
```

class `sage.rings.polynomial.laurent_polynomial_ring.`**`LaurentPolynomialRing_generic`**(*R*, *prepend_string*, *names*)

Bases: `sage.rings.ring.CommutativeRing`, `sage.structure.parent_gens.ParentWithGens`

Laurent polynomial ring (base class).

EXAMPLES:

This base class inherits from `CommutativeRing`. Since trac ticket #11900, it is also initialised as such:

```
sage: R.<x1,x2> = LaurentPolynomialRing(QQ)
sage: R.category()
Category of commutative rings
sage: TestSuite(R).run()
```

`change_ring`(*base_ring=None*, *names=None*, *sparse=False*, *order=None*)

EXAMPLES:

```
sage: R = LaurentPolynomialRing(QQ,2,'x')
sage: R.change_ring(ZZ)
Multivariate Laurent Polynomial Ring in x0, x1 over Integer Ring
```

`characteristic`()

Returns the characteristic of the base ring.

EXAMPLES:

```
sage: LaurentPolynomialRing(QQ,2,'x').characteristic()
0
sage: LaurentPolynomialRing(GF(3),2,'x').characteristic()
3
```

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

EXAMPLES:

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

```
-x^-1 + 1
sage: 1/PP(f)
-x - x^2 - x^3 - x^4 - x^5 - x^6 - x^7 - x^8 - x^9 - x^10 - x^11 - x^12 - x^13 - x^14 - x^15
```

construction()

Returns 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:

```
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.

EXAMPLES:

```
sage: LaurentPolynomialRing(QQ,2,'x').gen()
x0
sage: LaurentPolynomialRing(QQ,2,'x').gen(0)
x0
sage: LaurentPolynomialRing(QQ,2,'x').gen(1)
x1
```

TESTS:

```
sage: LaurentPolynomialRing(QQ,2,'x').gen(3)
Traceback (most recent call last):
...
ValueError: generator not defined
```

ideal()

EXAMPLES:

```
sage: LaurentPolynomialRing(QQ,2,'x').ideal()
Traceback (most recent call last):
...
NotImplementedError
```

is_exact()

Returns True if the base ring is exact.

EXAMPLES:

```
sage: LaurentPolynomialRing(QQ,2,'x').is_exact()
True
sage: LaurentPolynomialRing(RDF,2,'x').is_exact()
False
```

is_field(*proof=True*)

EXAMPLES:

```
sage: LaurentPolynomialRing(QQ,2,'x').is_field()
False
```

is_finite()

EXAMPLES:

```
sage: LaurentPolynomialRing(QQ,2,'x').is_finite()
False
```

is_integral_domain(*proof=True*)

Returns True if self is an integral domain.

EXAMPLES:

```
sage: LaurentPolynomialRing(QQ,2,'x').is_integral_domain()
True
```

The following used to fail; see #7530:

```
sage: L = LaurentPolynomialRing(ZZ, 'X')
sage: L['Y']
Univariate Polynomial Ring in Y over Univariate Laurent Polynomial Ring in X over Integer Ri
```

is_noetherian()

Returns 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()

Returns 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
```

class sage.rings.polynomial.laurent_polynomial_ring.**LaurentPolynomialRing_mpair** (*R*, *prepend_string*, *names*)

Bases: sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic

EXAMPLES:

```
sage: L = LaurentPolynomialRing(QQ,2,'x')
sage: type(L)
<class
'sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_mpair_with_category'>
sage: L == loads(dumps(L))
True
```

class sage.rings.polynomial.laurent_polynomial_ring.**LaurentPolynomialRing_univariate** (*R*, *names*)

Bases: sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic

EXAMPLES:

```
sage: L = LaurentPolynomialRing(QQ,'x')
sage: type(L)
<class 'sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_univariate_with_cate
sage: L == loads(dumps(L))
True
```

sage.rings.polynomial.laurent_polynomial_ring.**is_LaurentPolynomialRing** (*R*)

Returns True if and only if R is a Laurent polynomial ring.

EXAMPLES:

```
sage: from sage.rings.polynomial.laurent_polynomial_ring import is_LaurentPolynomialRing
sage: P = PolynomialRing(QQ,2,'x')
sage: is_LaurentPolynomialRing(P)
False
```

```
sage: R = LaurentPolynomialRing(QQ,3,'x')
sage: is_LaurentPolynomialRing(R)
True
```

5.2 Elements of Laurent polynomial rings

class `sage.rings.polynomial.laurent_polynomial.`**`LaurentPolynomial_generic`**

Bases: `sage.structure.element.CommutativeAlgebraElement`

A generic Laurent polynomial.

class `sage.rings.polynomial.laurent_polynomial.`**`LaurentPolynomial_mpair`**

Bases: `sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_generic`

Multivariate Laurent polynomials.

`coefficient`(*mon*)

Return the coefficient of `mon` in `self`, where `mon` must have the same parent as `self`.

The coefficient is defined as follows. If f is this polynomial, then the coefficient c_m is sum:

$$c_m := \sum_T \frac{T}{m}$$

where the sum is over terms T in f that are exactly divisible by m.

A monomial $m(x,y)$ 'exactly divides' $f(x,y)$ if $m(x,y)|f(x,y)$ and neither $x \cdot m(x,y)$ nor $y \cdot m(x,y)$ divides $f(x,y)$.

INPUT:

- `mon` – a monomial

OUTPUT:

Element of the parent of `self`.

Note: To get the constant coefficient, call `constant_coefficient()`.

EXAMPLES:

```
sage: P.<x,y> = LaurentPolynomialRing(QQ)
```

The coefficient returned is an element of the parent of `self`; in this case, `P`.

```
sage: f = 2 * x * y
sage: c = f.coefficient(x*y); c
2
sage: c.parent()
Multivariate Laurent Polynomial Ring in x, y over Rational Field

sage: P.<x,y> = LaurentPolynomialRing(QQ)
sage: f = (y^2 - x^9 - 7*x*y^2 + 5*x*y)*x^-3; f
-x^6 - 7*x^-2*y^2 + 5*x^-2*y + x^-3*y^2
sage: f.coefficient(y)
5*x^-2
sage: f.coefficient(y^2)
-7*x^-2 + x^-3
sage: f.coefficient(x*y)
```

```
0
sage: f.coefficient(x^-2)
-7*y^2 + 5*y
sage: f.coefficient(x^-2*y^2)
-7
sage: f.coefficient(1)
-x^6 - 7*x^-2*y^2 + 5*x^-2*y + x^-3*y^2
```

coefficients()

Return the nonzero coefficients of this polynomial in a list. The returned list is decreasingly ordered by the term ordering of `self.parent()`.

EXAMPLES:

```
sage: L.<x,y,z> = LaurentPolynomialRing(QQ,order='degrevlex')
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.coefficients()
[4, 3, 2, 1]
sage: L.<x,y,z> = LaurentPolynomialRing(QQ,order='lex')
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.coefficients()
[4, 1, 2, 3]
```

constant_coefficient()

Return the constant coefficient of `self`.

EXAMPLES:

```
sage: P.<x,y> = LaurentPolynomialRing(QQ)
sage: f = (y^2 - x^9 - 7*x*y^2 + 5*x*y)*x^-3; f
-x^6 - 7*x^-2*y^2 + 5*x^-2*y + x^-3*y^2
sage: f.constant_coefficient()
0
sage: f = (x^3 + 2*x^-2*y+y^3)*y^-3; f
x^3*y^-3 + 1 + 2*x^-2*y^-2
sage: f.constant_coefficient()
1
```

degree(*x=None*)

Returns the degree of x in self

EXAMPLES:

```
sage: R.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.degree(x)
7
sage: f.degree(y)
1
sage: f.degree(z)
0
```

derivative(**args*)

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

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

See also:

`_derivative()`

EXAMPLES:

```
sage: R = LaurentPolynomialRing(ZZ,'x, y')
sage: x, y = R.gens()
sage: t = x**4*y+x*y+y+x**(-1)+y**(-3)
sage: t.derivative(x, x)
12*x^2*y + 2*x^-3
sage: t.derivative(y, 2)
12*y^-5
```

dict()

EXAMPLES:

```
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: list(sorted(f.dict().iteritems()))
[((3, 1, 0), 3), ((4, 0, -2), 2), ((6, -7, 0), 1), ((7, 0, -1), 4)]
```

diff(**args*)

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

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

See also:

`_derivative()`

EXAMPLES:

```
sage: R = LaurentPolynomialRing(ZZ,'x, y')
sage: x, y = R.gens()
sage: t = x**4*y+x*y+y+x**(-1)+y**(-3)
sage: t.derivative(x, x)
12*x^2*y + 2*x^-3
sage: t.derivative(y, 2)
12*y^-5
```

differentiate(**args*)

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

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

See also:

`_derivative()`

EXAMPLES:

```
sage: R = LaurentPolynomialRing(ZZ,'x, y')
sage: x, y = R.gens()
sage: t = x**4*y+x*y+y+x**(-1)+y**(-3)
sage: t.derivative(x, x)
12*x^2*y + 2*x^-3
sage: t.derivative(y, 2)
12*y^-5
```

exponents()

Returns a list of the exponents of self.

EXAMPLES:

```
sage: L.<w,z> = LaurentPolynomialRing(QQ)
sage: a = w^2*z^-1+3; a
w^2*z^-1 + 3
sage: e = a.exponents()
sage: e.sort(); e
[(0, 0), (2, -1)]
```

factor()

Returns a Laurent monomial (the unit part of the factorization) and a factored multi-polynomial.

EXAMPLES:

```
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.factor()
(x^3*y^-7*z^-2) * (4*x^4*y^7*z + 3*y^8*z^2 + 2*x*y^7 + x^3*z^2)
```

has_any_inverse()

Returns True if self contains any monomials with a negative exponent, False otherwise.

EXAMPLES:

```
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.has_any_inverse()
True
sage: g = x^2 + y^2
sage: g.has_any_inverse()
False
```

has_inverse_of(*i*)

INPUT:

- `i` – The index of a generator of `self.parent()`

OUTPUT:

Returns True if self contains a monomial including the inverse of `self.parent().gen(i)`, False otherwise.

EXAMPLES:

```
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.has_inverse_of(0)
False
sage: f.has_inverse_of(1)
True
sage: f.has_inverse_of(2)
True
```

is_monomial()

Return True if this element is a monomial.

EXAMPLES:

```
sage: k.<y,z> = LaurentPolynomialRing(QQ)
sage: z.is_monomial()
True
sage: k(1).is_monomial()
True
sage: (z+1).is_monomial()
```

```
False
sage: (z^-2909).is_monomial()
True
sage: (38*z^-2909).is_monomial()
False
```

is_univariate()

Return `True` if this is a univariate or constant Laurent polynomial, and `False` otherwise.

EXAMPLES:

```
sage: R.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = (x^3 + y^-3)*z
sage: f.is_univariate()
False
sage: g = f(1,y,4)
sage: g.is_univariate()
True
sage: R(1).is_univariate()
True
```

monomial_coefficient(*mon*)

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

This function contrasts with the function `coefficient()` which returns the coefficient of a monomial viewing this polynomial in a polynomial ring over a base ring having fewer variables.

INPUT:

- `mon` - a monomial

See also:

For coefficients in a base ring of fewer variables, see `coefficient()`.

EXAMPLES:

```
sage: P.<x,y> = LaurentPolynomialRing(QQ)
sage: f = (y^2 - x^9 - 7*x*y^3 + 5*x*y)*x^-3
sage: f.monomial_coefficient(x^-2*y^3)
-7
sage: f.monomial_coefficient(x^2)
0
```

monomials()

Return the list of monomials in `self`.

EXAMPLES:

```
sage: P.<x,y> = LaurentPolynomialRing(QQ)
sage: f = (y^2 - x^9 - 7*x*y^3 + 5*x*y)*x^-3
sage: f.monomials()
[x^6, x^-3*y^2, x^-2*y, x^-2*y^3]
```

subs(*in_dict=None*, ***kwds*)

Note that this is a very unsophisticated implementation.

EXAMPLES:

```
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = x + 2*y + 3*z
```

```
sage: f.subs(x=1)
2*y + 3*z + 1
sage: f.subs(y=1)
x + 3*z + 2
sage: f.subs(z=1)
x + 2*y + 3
sage: f.subs(x=1,y=1,z=1)
6

sage: f = x^-1
sage: f.subs(x=2)
1/2
sage: f.subs({x:2})
1/2

sage: f = x + 2*y + 3*z
sage: f.subs({x:1,y:1,z:1})
6
sage: f.substitute(x=1,y=1,z=1)
6
```

TESTS:

```
sage: f = x + 2*y + 3*z
sage: f(q=10)
x + 2*y + 3*z
```

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. The new polynomial is over the same base ring as the given LaurentPolynomial and in the variable x if no ring R is provided.

EXAMPLES:

```
sage: R.<x, y> = LaurentPolynomialRing(ZZ)
sage: f = 3*x^2 - 2*y^-1 + 7*x^2*y^2 + 5
sage: f.univariate_polynomial()
Traceback (most recent call last):
...
TypeError: polynomial must involve at most one variable
sage: g = f(10,y); g
700*y^2 + 305 - 2*y^-1
sage: h = g.univariate_polynomial(); h
-2*y^-1 + 305 + 700*y^2
sage: h.parent()
Univariate Laurent Polynomial Ring in y over Integer Ring
sage: g.univariate_polynomial(LaurentPolynomialRing(QQ,'z'))
-2*z^-1 + 305 + 700*z^2
```

Here's an example with a constant multivariate polynomial:

```
sage: g = R(1)
sage: h = g.univariate_polynomial(); h
1
```

```
sage: h.parent()
Univariate Laurent Polynomial Ring in x over Integer Ring
```

variables(*sort=True*)

Return a tuple of all variables occurring in self.

INPUT:

- `sort` – specifies whether the indices shall be sorted

EXAMPLES:

```
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.variables()
(z, y, x)
sage: f.variables(sort=False) #random
(y, z, x)
```

class `sage.rings.polynomial.laurent_polynomial.`**LaurentPolynomial_univariate**

Bases: `sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_generic`

A univariate Laurent polynomial in the form of $t^n \cdot f$ where f is a polynomial in t.

INPUT:

- `parent` – a Laurent polynomial ring
- `f` – a polynomial (or something can be coerced to one)
- `n` – (default: 0) an integer

AUTHORS:

- Tom Boothby (2011) copied this class almost verbatim from `laurent_series_ring_element.pyx`, so most of the credit goes to William Stein, David Joyner, and Robert Bradshaw
- Travis Scrimshaw (09-2013): Cleaned-up and added a few extra methods

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
```

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 this polynomial.

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 :func‘derivative()‘ function for more details.

See also:

`_derivative()`

EXAMPLES:

```
sage: R.<x> = LaurentPolynomialRing(QQ)
sage: g = 1/x^10 - x + x^2 - x^4
sage: g.derivative()
-10*x^-11 - 1 + 2*x - 4*x^3
sage: g.derivative(x)
-10*x^-11 - 1 + 2*x - 4*x^3

sage: R.<t> = PolynomialRing(ZZ)
sage: S.<x> = LaurentPolynomialRing(R)
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:: 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:

```
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:

```
sage: R.<t> = LaurentPolynomialRing(ZZ)
sage: f = 4*t^-7 + 3*t^3 + 2*t^4 + t^-6
sage: f.factor()
(t^-7) * (4 + t + 3*t^10 + 2*t^11)
```

gcd(*right*)

Return the gcd of `self` with `right` where the common divisor `d` makes both `self` and `right` into polynomials with the lowest possible degree.

EXAMPLES:

```
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 coeffic
```

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 `True` if `self` 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
```

is_monomial()

Return True if this element 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_unit()

Return `True` if this Laurent polynomial is a unit in this ring.

EXAMPLES:

```
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: (2+t).is_unit()
False
sage: f = 2*t
sage: f.is_unit()
```

```
True
sage: 1/f
1/2*t^-1
sage: R(0).is_unit()
False
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:

```
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
```

polynomial_construction()
Return the polynomial and the shift in power used to construct the Laurent polynomial $t^n u$.

OUTPUT:

A tuple `(u, n)` where `u` is the underlying polynomial and `n` is the power of the exponent shift.

EXAMPLES:

```
sage: R.<x> = LaurentPolynomialRing(QQ)
sage: f = 1/x + x^2 + 3*x^4
sage: f.polynomial_construction()
(3*x^5 + x^3 + 1, -1)
```

quo_rem(*right_r*)
Attempts to divide `self` by `right` and returns a quotient and a remainder.

EXAMPLES:

```
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: (t^-3 - t^3).quo_rem(t^-1 - t)
(t^-2 + 1 + t^2, 0)
sage: (t^-2 + 3 + t).quo_rem(t^-4)
(t^2 + 3*t^4 + t^5, 0)
sage: (t^-2 + 3 + t).quo_rem(t^-4 + t)
(0, 1 + 3*t^2 + t^3)
```

residue()
Return the residue of `self`.

The residue is the coefficient of t^-1.

EXAMPLES:

```
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: f = 3*t^-2 - t^-1 + 3 + t^2
sage: f.residue()
-1
sage: g = -2*t^-2 + 4 + 3*t
sage: g.residue()
0
sage: f.residue().parent()
Rational Field
```

shift(*k*)

Return this Laurent polynomial multiplied by the power t^n. Does not change this polynomial.

EXAMPLES:

```
sage: R.<t> = LaurentPolynomialRing(QQ['y'])
sage: f = (t+t^-1)^4; f
t^-4 + 4*t^-2 + 6 + 4*t^2 + t^4
sage: f.shift(10)
t^6 + 4*t^8 + 6*t^10 + 4*t^12 + t^14
sage: f >> 10
t^-14 + 4*t^-12 + 6*t^-10 + 4*t^-8 + t^-6
sage: f << 4
1 + 4*t^2 + 6*t^4 + 4*t^6 + t^8
```

truncate(*n*)

Return a polynomial with degree at most $n-1$ whose j-th coefficients agree with `self` for all $j < n$.

EXAMPLES:

```
sage: R.<x> = LaurentPolynomialRing(QQ)
sage: f = 1/x^12 + x^3 + x^5 + x^9
sage: f.truncate(10)
x^-12 + x^3 + x^5 + x^9
sage: f.truncate(5)
x^-12 + x^3
sage: f.truncate(-16)
0
```

valuation(*p=None*)

Return the valuation of `self`.

The valuation of a Laurent polynomial $t^n u$ is n plus the valuation of u.

EXAMPLES:

```
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 occuring 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()
()
```

CHAPTER SIX

BOOLEAN POLYNOMIALS

Elements of the quotient ring

$$\mathbf{F}_2[x_1, ..., x_n]/ < x_1^2 + x_1, ..., x_n^2 + x_n > .$$

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." - [BD07]

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

$$< ab + cd + 1, ace + de, abe + ce, bc + cde + 1 > .$$

First, we compute the lexicographical Groebner basis in the polynomial ring

$$R = \mathbf{F}_2[a, b, c, d, e].$$

```
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 $\mathbf{F}_2$ then this Groebner basis was the one to consider. If one wants solutions over $\mathbf{F}_2$ only then one adds the field polynomials to the ideal to force the solutions in $\mathbf{F}_2$.

```
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 $\mathbf{F}_2$ 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 $\mathbf{F}_2$ by considering the quotient ring. If I is an ideal in $\mathbb{F}[x_1, ..., x_n]$ then the ideals in the quotient ring $\mathbb{F}[x_1, ..., x_n]/I$ are in one-to-one correspondence with the ideals of $\mathbb{F}[x_0, ..., x_n]$ 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
abar + dbar + 1
bbar + 1
cbar + 1
ebar
```

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.

6.1 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.

6.2 Access to the original PolyBoRi interface

The re-implementation PolyBoRi's native wrapper is available to the user too:

```
sage: from polybori import *
sage: declare_ring([Block('x',2),Block('y',3)],globals())
Boolean PolynomialRing in x0, x1, y0, y1, y2
sage: r
Boolean PolynomialRing in x0, x1, y0, y1, y2

sage: [Variable(i, r) for i in xrange(r.ngens())]
[x(0), x(1), y(0), y(1), 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.

REFERENCES:

class `sage.rings.polynomial.pbori.`**`BooleConstant`**

Bases: `object`

Construct a boolean constant (modulo 2) from integer value:

INPUT:

- `i` - an integer

EXAMPLE:

```
sage: from polybori import BooleConstant
sage: [BooleConstant(i) for i in range(5)]
[0, 1, 0, 1, 0]
```

`deg``()`

Get degree of boolean constant.

EXAMPLE:

```
sage: from polybori import BooleConstant
sage: BooleConstant(0).deg()
-1
sage: BooleConstant(1).deg()
0
```

`has_constant_part``()`

This is true for for $BooleConstant(1)$.

EXAMPLE:

```
sage: from polybori 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.

EXAMPLE:

```
sage: from polybori import BooleConstant
sage: BooleConstant(1).is_constant()
True
sage: BooleConstant(0).is_constant()
True
```

is_one()

Check whether boolean constant is one.

EXAMPLE:

```
sage: from polybori import BooleConstant
sage: BooleConstant(0).is_one()
False
sage: BooleConstant(1).is_one()
True
```

is_zero()

Check whether boolean constant is zero.

EXAMPLE:

```
sage: from polybori import BooleConstant
sage: BooleConstant(1).is_zero()
False
sage: BooleConstant(0).is_zero()
True
```

variables()

Get variables (return always and empty tuple).

EXAMPLE:

```
sage: from polybori import BooleConstant
sage: BooleConstant(0).variables()
()
sage: BooleConstant(1).variables()
()
```

class sage.rings.polynomial.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.

EXAMPLE:

```
sage: from polybori 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 polybori import *
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.

$$X \times Y = \{(x, y) | x \in X \text{ and } y \in Y\}.$$

EXAMPLE:

```
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.

EXAMPLE:

```
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:

$$X\ Y = \{x | x \in X \text{ and } x \notin Y\}.$$

EXAMPLE:

```
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.

EXAMPLE:

```
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

EXAMPLE:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
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:

$$X \cap Y = \{x | x \in X \text{ and } x \in Y\}.$$

EXAMPLE:

```
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.intersect(t)
{{x2,x3}}
```

minimal_elements()

Return a new set containing a divisor of all elements of this set.

EXAMPLE:

```
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

EXAMPLE:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
sage: from polybori import BooleSet
sage: B = BooleanPolynomialRing(5,'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2+x2*x3*x4+x2*x4+x3+x4+1
sage: s = f.set(); s
{{x1,x2}, {x2,x3,x4}, {x2,x4}, {x3}, {x4}, {}}

sage: nav = s.navigation()
sage: BooleSet(nav,s.ring())
{{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.

EXAMPLE:

```
sage: B = BooleanPolynomialRing(5,'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2+x2*x3*x4+x2*x4+x3+x4+1
sage: f.set().ring() is B
True
```

set()

Return `self`.

EXAMPLE:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
sage: B.<x,y> = BooleanPolynomialRing()
sage: s = x.set()
sage: s.stable_hash()
-845955105                 # 32-bit
173100285919               # 64-bit
```

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

EXAMPLE:

```
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

EXAMPLE:

```
sage: BooleanPolynomialRing(5,'x')
Boolean PolynomialRing in x0, x1, x2, x3, x4
sage: B = BooleanPolynomialRing(5,'x')
sage: B.inject_variables()
Defining x0, x1, x2, x3, x4
sage: f = x1*x2+x2*x3
sage: s = f.set(); s
{{x1,x2}, {x2,x3}}
sage: s.subset1(1)
{{x2}}
```

union(*rhs*)

Return the set theoretic union of this set and the set `rhs`.

The union of two sets X and Y is defined as:

$$X \cup Y = \{x | x \in X \text{ or } x \in Y\}.$$

EXAMPLE:

```
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.

EXAMPLE:

```
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.`**BooleSetIterator**

Bases: `object`

Helper class to iterate over boolean sets.

next()

x.next() -> the next value, or raise StopIteration

class `sage.rings.polynomial.pbori.`**BooleanMonomial**

Bases: `sage.structure.element.MonoidElement`

Construct a boolean monomial.

INPUT:

- `parent` - parent monoid this element lives in

EXAMPLE:

```
sage: from polybori 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.

EXAMPLES:

```
sage: from polybori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M(x*y).deg()
2

sage: M(x*x*y*z).deg()
3
```

Note: This function is part of the upstream PolyBoRi interface.

degree(*x=None*)

Return the degree of this monomial in `x`, where `x` must be one of the generators of the polynomial ring.

INPUT:

- `x` - boolean multivariate polynomial (a generator of the polynomial ring). If `x` is not specified (or is `None`), return the total degree of this monomial.

EXAMPLES:

```
sage: from polybori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M(x*y).degree()
2
sage: M(x*y).degree(x)
1
sage: M(x*y).degree(z)
0
```

divisors()

Return a set of boolean monomials with all divisors of this monomial.

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y
sage: m = f.lm()
sage: m.divisors()
{{x,y}, {x}, {y}, {}}
```

gcd(*rhs*)

Return the greatest common divisor of this boolean monomial and `rhs`.

INPUT:

- `rhs` - a boolean monomial

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: a,b,c,d = a.lm(), b.lm(), c.lm(), d.lm()
sage: (a*b).gcd(b*c)
b
sage: (a*b*c).gcd(d)
1
```

index()

Return the variable index of the first variable in this monomial.

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y
sage: m = f.lm()
sage: m.index()
0

# Check that Ticket #13133 is resolved:
sage: B(1).lm().index()
Traceback (most recent call last):
...
ValueError: no variables in constant monomial ; cannot take index()
```

Note: This function is part of the upstream PolyBoRi interface.

iterindex()

Return an iterator over the indicies of the variables in self.

EXAMPLES:

```
sage: from polybori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(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

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(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}}
```

Note: The returned set always contains `self` even if the bound `rhs` is smaller than `self`.

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.

EXAMPLE:

```
sage: from polybori 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

EXAMPLE:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
sage: B.<x,y> = BooleanPolynomialRing()
sage: m = x.lm()
sage: m.stable_hash()
-845955105                 # 32-bit
173100285919               # 64-bit
```

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.

EXAMPLE:

```
sage: from polybori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M(x*z).variables() # indirect doctest
(x, z)
```

class sage.rings.polynomial.pbori.**BooleanMonomialIterator**

Bases: object

An iterator over the variable indices of a monomial.

next()

x.next() -> the next value, or raise StopIteration

class sage.rings.polynomial.pbori.**BooleanMonomialMonoid**(*polring*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.monoids.monoid.Monoid_class

Construct a boolean monomial monoid given a boolean polynomial ring.

This object provides a parent for boolean monomials.

INPUT:

- polring - the polynomial ring our monomials lie in

EXAMPLES:

```
sage: from polybori import BooleanMonomialMonoid
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: M = BooleanMonomialMonoid(P)
sage: M
MonomialMonoid of Boolean PolynomialRing in x, y

sage: M.gens()
(x, y)
sage: type(M.gen(0))
<type 'sage.rings.polynomial.pbori.BooleanMonomial'>
```

Since trac ticket #9138, boolean monomial monoids are unique parents and are fit into the category framework:

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

gen (*i=0*)

Return the i-th generator of self.

INPUT:

- `i` - an integer

EXAMPLES:

```
sage: from polybori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M.gen(0)
x
sage: M.gen(2)
z

sage: P = BooleanPolynomialRing(1000, 'x')
sage: M = BooleanMonomialMonoid(P)
sage: M.gen(50)
x50
```

gens ()

Return the tuple of generators of this monoid.

EXAMPLES:

```
sage: from polybori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M.gens()
(x, y, z)
```

ngens ()

Returns the number of variables in this monoid.

EXAMPLES:

```
sage: from polybori import BooleanMonomialMonoid
sage: P = BooleanPolynomialRing(100, 'x')
sage: M = BooleanMonomialMonoid(P)
sage: M.ngens()
100
```

class `sage.rings.polynomial.pbori.`**BooleanMonomialVariableIterator**

Bases: `object`

x.__init__(...) initializes x; see help(type(x)) for signature

next ()

x.next() -> the next value, or raise StopIteration

class `sage.rings.polynomial.pbori.`**BooleanMulAction**

Bases: `sage.categories.action.Action`

class `sage.rings.polynomial.pbori.`**BooleanPolynomial**

Bases: `sage.rings.polynomial.multi_polynomial.MPolynomial`

Construct a boolean polynomial object in the given boolean polynomial ring.

INPUT:

- `parent` - a boolean polynomial ring

TEST:

```
sage: from polybori import BooleanPolynomial
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: BooleanPolynomial(B)
0
```

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.

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: x.constant()
False

sage: B(1).constant()
True
```

Note: This function is part of the upstream PolyBoRi interface.

constant_coefficient()

Returns the constant coefficient of this boolean polynomial.

EXAMPLE:

```
sage: B.<a,b> = BooleanPolynomialRing()
sage: a.constant_coefficient()
0
sage: (a+1).constant_coefficient()
1
```

deg()

Return the degree of `self`. This is usually equivalent to the total degree except for weighted term orderings which are not implemented yet.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: (x+y).degree()
1

sage: P(1).degree()
0

sage: (x*y + x + y + 1).degree()
2
```

Note: This function is part of the upstream PolyBoRi interface.

degree(*x=None*)

Return the maximal degree of this polynomial in `x`, where `x` must be one of the generators for the parent of this polynomial.

If x is not specified (or is `None`), return the total degree, which is the maximum degree of any monomial.

EXAMPLES:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
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

EXAMPLE:

```
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
```

TESTS:

```
sage: f.graded_part(-1)
0
```

has_constant_part()

Return `True` if this boolean polynomial has a constant part, i.e. if `1` is a term.

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + c*d + a*b + 1
sage: f.has_constant_part()
True

sage: f = a*b*c + c*d + a*b
sage: f.has_constant_part()
False
```

is_constant()

Check if `self` is constant.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P(1).is_constant()
True

sage: P(0).is_constant()
True

sage: x.is_constant()
False

sage: (x*y).is_constant()
False
```

is_equal(*right*)

EXAMPLE:

```
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_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_singleton()

Check if `self` has at most one term.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P(0).is_singleton()
True

sage: x.is_singleton()
True

sage: P(1).is_singleton()
True

sage: (x*y).is_singleton()
True

sage: (x + y).is_singleton()
False

sage: (x + 1).is_singleton()
False

sage: (x*y + 1).is_singleton()
False

sage: (x + y + 1).is_singleton()
False

sage: ((x + 1)*(y + 1)).is_singleton()
False
```

is_singleton_or_pair()

Check if `self` has at most two terms.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P(0).is_singleton_or_pair()
True

sage: x.is_singleton_or_pair()
True

sage: P(1).is_singleton_or_pair()
True

sage: (x*y).is_singleton_or_pair()
True

sage: (x + y).is_singleton_or_pair()
True

sage: (x + 1).is_singleton_or_pair()
True

sage: (x*y + 1).is_singleton_or_pair()
True

sage: (x + y + 1).is_singleton_or_pair()
False
```

```
sage: ((x + 1)*(y + 1)).is_singleton_or_pair()
False
```

is_unit()

Check if `self` is invertible in the parent ring.

Note that this condition is equivalent to being 1 for boolean polynomials.

EXAMPLE:

```
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, that is if self contains only 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 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()

Returns the total degree of the leading monomial of `self`.

EXAMPLES:

```
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.

EXAMPLE:

```
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:

```
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.

EXAMPLE:

```
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:

```
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:

```
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`.

EXAMPLE:

```
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

EXAMPLE:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
sage: B = BooleanPolynomialRing(5,'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2 + x2*x3 + 1
sage: f.n_nodes()
4
```

Note: This function is part of the upstream PolyBoRi interface.

n_vars()

Return the number of variables used to form this boolean polynomial.

EXAMPLE:

```
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.

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.

EXAMPLE:

```
sage: from polybori 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
```

Note: This function is part of the upstream PolyBoRi interface.

nvariables()

Return the number of variables used to form this boolean polynomial.

EXAMPLE:

```
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.

EXAMPLE:

```
sage: B.<x0,x1,x2,x3> = BooleanPolynomialRing(4)
sage: I = B.ideal((x0 + x1 + x2 + x3, \
                   x0*x1 + x1*x2 + x0*x3 + x2*x3, \
                   x0*x1*x2 + x0*x1*x3 + x0*x2*x3 + x1*x2*x3, \
                   x0*x1*x2*x3 + 1))
sage: gb = I.groebner_basis()
sage: f,g,h,i = I.gens()
```

```
sage: f.reduce(gb)
0
sage: p = f*g + x0*h + x2*i
sage: p.reduce(gb)
0
sage: p.reduce(I)
x1*x2*x3 + x2
sage: p.reduce([])
x0*x1*x2 + x0*x1*x3 + x0*x2*x3 + x2
```

Note: If this function is called repeatedly with the same I then it is advised to use PolyBoRi's `GroebnerStrategy` object directly, since that will be faster. See the source code of this function for details.

TESTS:

```
sage: R=BooleanPolynomialRing(20,'x','lex')
sage: a=R.random_element()
sage: a.reduce([None,None])
Traceback (most recent call last):
...
TypeError: argument must be a BooleanPolynomial.
```

reducible_by(*rhs*)

Return `True` if this boolean polynomial is reducible by the polynomial `rhs`.

INPUT:

- `rhs` - a boolean polynomial

EXAMPLE:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
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`.

EXAMPLE:

```
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.

EXAMPLE:

```
sage: B.<x,y> = BooleanPolynomialRing()
sage: x.stable_hash()
-845955105                 # 32-bit
173100285919               # 64-bit
```

Note: This function is part of the upstream PolyBoRi interface. In Sage all hashes are stable.

subs (*in_dict=None*, ***kwds*)

Fixes some given variables in a given boolean polynomial and returns the changed boolean polynomials. The polynomial itself is not affected. The variable,value pairs for fixing are to be provided as dictionary of the form {variable:value} or named parameters (see examples below).

INPUT:

- `in_dict` - (optional) dict with variable:value pairs
- `**kwds` - names parameters

EXAMPLE:

```
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'))
a*b + b*c + c + 1
sage: f.subs({'x':var('a'),'y':var('b'),'z':var('c')})
a*b + b*c + c + 1
```

terms()

Return a list of monomials appearing in `self` ordered largest to smallest.

EXAMPLE:

```
sage: P.<a,b,c> = BooleanPolynomialRing(3,order='lex')
sage: f = a + c*b
sage: f.terms()
[a, b*c]

sage: P.<a,b,c> = BooleanPolynomialRing(3,order='deglex')
sage: f = a + c*b
sage: f.terms()
[b*c, a]
```

total_degree()

Return the total degree of `self`.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: (x+y).total_degree()
1

sage: P(1).total_degree()
0

sage: (x*y + x + y + 1).total_degree()
2
```

univariate_polynomial(*R=None*)

Returns 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.

> sage: R.<x, y> = BooleanPolynomialRing() sage: f = x - y + x*y + 1 sage: f.univariate_polynomial() Traceback (most recent call last): ... ValueError: polynomial must involve at most one variable sage: g = f.subs({x:0}); g y + 1 sage: g.univariate_polynomial () y + 1 sage: g.univariate_polynomial(GF(2)['foo']) foo + 1

Here's an example with a constant multivariate polynomial:

```
sage: g = R(1)
sage: h = g.univariate_polynomial(); h
1
sage: h.parent()
Univariate Polynomial Ring in x over Finite Field of size 2 (using NTL)
```

variable(*i=0*)

Return the i-th variable occurring in self. The index i is the index in `self.variables()`

EXAMPLES:

```
sage: P.<x,y,z> = 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`.

EXAMPLE:

```
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 the 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
```

TESTS:

```
sage: R = BooleanPolynomialRing(1, 'y')
sage: y.vars_as_monomial()
y
sage: R
Boolean PolynomialRing in y
```

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

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b + c + d + 1
```

Now we create a set of points:

```
sage: s = a*b + a*b*c + c*d + 1
sage: s = s.set(); s
{{a,b,c}, {a,b}, {c,d}, {}}
```

This encodes the points (1,1,1,0), (1,1,0,0), (0,0,1,1) and (0,0,0,0). But of these only (1,1,0,0) evaluates to zero.

```
sage: f.zeros_in(s)
{{a,b}}

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.`**`BooleanPolynomialEntry`**

Bases: `object`

`p`

class `sage.rings.polynomial.pbori.`**`BooleanPolynomialIdeal`**(*ring*, *gens=[ ]*, *coerce=True*)

Bases: `sage.rings.polynomial.multi_polynomial_ideal.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: `True`)

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
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.

TESTS:

Check that trac ticket #13155 is solved:

```
sage: R = BooleanPolynomialRing(11, 'x')
sage: R2 = PolynomialRing(GF(2), 11, 'x')
sage: I = ideal([ R(f) for f in sage.rings.ideal.Cyclic(R2, 11).gens() ])
sage: I.dimension()
0
```

`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`)
- `faugere` - turn off or on the linear algebra (default: `False`)
- `linear_algebra_in_last_block` - this affects the last block of block orderings and degree orderings. If it is set to `True` linear algebra takes affect in this block. (default: `True`)
- **`gauss_on_linear` - perform Gaussian elimination on linear** polynomials (default: `True`)
- `selection_size` - maximum number of polynomials for parallel reductions (default: `1000`)
- `heuristic` - Turn off heuristic by setting `heuristic=False` (default: `True`)
- `lazy` - (default: `True`)
- `invert` - setting `invert=True` input and output get a transformation `x+1` for each variable `x`, which shouldn't effect the calculated GB, but the algorithm.
- `other_ordering_first` - possible values are `False` or an ordering code. In practice, many Boolean examples have very few solutions and a very easy Groebner basis. So, a complex walk algorithm (which cannot be implemented using the data structures) seems unnecessary, as such Groebner bases can be converted quite fast by the normal Buchberger algorithm from one ordering into another ordering. (default: `False`)
- `prot` - show protocol (default: `False`)
- `full_prot` - show full protocol (default: `False`)

EXAMPLES:

```
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: F,s = sr.polynomial_system()
sage: I = F.ideal()
sage: I.groebner_basis()
Polynomial Sequence with 36 Polynomials in 36 Variables
```

We compute the same example with Magma:

```
sage: sr = mq.SR(2,1,1,4,gf2=True, polybori=True)
sage: F,s = sr.polynomial_system()
sage: I = F.ideal()
sage: I.groebner_basis(algorithm='magma', prot='sage') # optional - magma
Leading term degree:  1. Critical pairs: 148.
Leading term degree:  2. Critical pairs: 144.
Leading term degree:  3. Critical pairs: 462.
Leading term degree:  1. Critical pairs: 167.
Leading term degree:  2. Critical pairs: 147.
Leading term degree:  3. Critical pairs: 101 (all pairs of current degree eliminated by crit

Highest degree reached during computation:  3.
Polynomial Sequence with 35 Polynomials in 36 Variables
```

TESTS:

This example shows, that a bug in our variable indices was indeed fixed:

```
sage: R.<a111,a112,a121,a122,b111,b112,b211,b212,c111,c112> = BooleanPolynomialRing(order='l
sage: I = (a111 * b111 * c111 + a112 * b112 * c112 - 1, a111 * b211 * c111 + a112 * b212 * c
...        a121 * b111 * c111 + a122 * b112 * c112, a121 * b211 * c111 + a122 * b212 * c112
sage: I.groebner_basis()
[a111 + b212, a112 + b211, a121 + b112, a122 + b111, b111*b112 + b111 + b112 + 1,
 b111*b211 + b111 + b211 + 1, b111*b212 + b112*b211 + 1, b112*b212 + b112 + b212 + 1,
 b211*b212 + b211 + b212 + 1, c111 + 1, c112 + 1]
```

The following example shows whether boolean constants are handled correctly:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: I = Ideal([x*z + y*z + z, x*y + x*z + x + y*z + y + z])
sage: I.groebner_basis()
[x, y, z]
```

Check that this no longer crash (trac ticket #12792):

```
sage: names = [ "s{0}s{1}".format(i,j) for i in range(4) for j in range(8)]
sage: R = BooleanPolynomialRing( 32, names)
sage: R.inject_variables()
Defining s0s0, ...
sage: problem = [s1s0*s1s1, s0s0*s0s1 + s0s0 + s0s1 + s2s0 + s3s0*s3s1 + s3s0 + s3s1,
...             s1s1 + s2s0 + s3s0 + s3s1 + 1, s0s0*s0s1 + s1s1 + s3s0*s3s1 + s3s0,
...             s0s1 + s1s0 + s1s1 + s3s0, s0s0*s0s1 + s0s0 + s0s1 + s1s1 + s2s0 + s3s1,
...             s0s1 + s1s0, s0s0*s0s1 + s0s0 + s0s1 + s1s0 + s2s0 + s3s1,
...             s0s0 + s2s0 + s3s0*s3s1 + s3s0 + 1, s0s0 + s1s1]
sage: ideal(problem).groebner_basis()
[1]
```

interreduced_basis()

If this ideal is spanned by `(f_1, ..., f_n)` this method returns `(g_1, ..., g_s)` such that:

- `<f_1,...,f_n> = <g_1,...,g_s>`
- `LT(g_i) != LT(g_j)` for all `i != j`‘
- `LT(g_i)` does not divide `m` for all monomials `m` of `{g_1,...,g_{i-1},g_{i+1},...,g_s}`

EXAMPLE:

```
sage: sr = mq.SR(1, 1, 1, 4, gf2=True, polybori=True)
sage: F,s = sr.polynomial_system()
sage: I = F.ideal()
sage: I.interreduced_basis()
[k100 + 1, k101 + k001 + 1, k102, k103 + 1, x100 + k001 + 1, x101 + k001, x102, x103 + k001,
```

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.

EXAMPLE:

```
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)
1
sage: I.reduce(gb[0]*gb[1])
0
sage: I.reduce(gb[0]*B.gen(1))
0
```

variety(***kwds*)

Return the variety associated to this boolean ideal.

EXAMPLE:

A Simple example:

```
sage: R.<x,y,z> = BooleanPolynomialRing()
sage: I = ideal( [ x*y*z + x*z + y + 1, x+y+z+1 ] )
sage: I.variety()
[{z: 0, y: 1, x: 0}, {z: 1, y: 1, x: 1}]
```

TESTS:

BooleanIdeal and regular (quotient) Ideal should coincide:

```
sage: R = BooleanPolynomialRing(6, ['x%d'%(i+1) for i in range(6)], order='lex')
sage: R.inject_variables()
Defining...
sage: polys = [\
       x1*x2 + x1*x4 + x1*x5 + x1*x6 + x1 + x2 + x3*x4 + x3*x5 + x3 + x4*x5 + x4*x6 + x4
       x1*x2 + x1*x3 + x1*x4 + x1*x6 + x2*x3 + x2*x6 + x2 + x3*x4 + x5*x6, \
       x1*x3 + x1*x4 + x1*x6 + x1 + x2*x5 + x2*x6 + x3*x4 + x3 + x4*x6 + x4 + x5*x6 + x5
       x1*x2 + x1*x3 + x1*x4 + x1*x5 + x2 + x3*x5 + x3*x6 + x3 + x5 + x6, \
       x1*x2 + x1*x4 + x1*x5 + x1*x6 + x2*x3 + x2*x4 + x2*x5 + x3*x5 + x5*x6 + x5 + x6,
       x1*x2 + x1*x6 + x2*x4 + x2*x5 + x2*x6 + x3*x6 + x4*x6 + x5*x6 + x5]
sage: I = R.ideal( polys )
sage: I.variety()
[{x6: 0, x5: 0, x4: 0, x3: 0, x2: 0, x1: 0},
 {x6: 1, x5: 0, x4: 0, x3: 1, x2: 1, x1: 1}]

sage: R = PolynomialRing(GF(2), 6, ['x%d'%(i+1) for i in range(6)], order='lex')
sage: I = R.ideal( polys )
sage: (I + sage.rings.ideal.FieldIdeal(R)).variety()
[{x2: 0, x5: 0, x4: 0, x1: 0, x6: 0, x3: 0}, {x2: 1, x5: 0, x4: 0, x1: 1, x6: 1, x3: 1}]
```

Check that trac ticket #13976 is fixed:

```
sage: R.<x,y,z> = BooleanPolynomialRing()
sage: I = ideal( [ x*y*z + x*z + y + 1, x+y+z+1 ] )
sage: sols = I.variety()
```

```
sage: sols[0][y]
1
```

class sage.rings.polynomial.pbori.**BooleanPolynomialIterator**

Bases: object

Iterator over the monomials of a boolean polynomial.

next()

x.next() -> the next value, or raise StopIteration

class sage.rings.polynomial.pbori.**BooleanPolynomialRing**

Bases: sage.rings.polynomial.multi_polynomial_ring_generic.MPolynomialRing_generic

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
```

TESTS:

```
sage: P.<x0, x1, x2, x3> = BooleanPolynomialRing(4,order='deglex(2),deglex(2)')
sage: x0 > x1
True
sage: x2 > x3
True
sage: TestSuite(P).run()
```

Boolean polynomial rings are unique parent structures. We thus have:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: R.<x,y> = BooleanPolynomialRing(2)
sage: P is R
True
```

```
sage: Q.<x,z> = BooleanPolynomialRing(2)
sage: P == Q
False

sage: S.<x,y> = BooleanPolynomialRing(2, order='deglex')
sage: P == S
False
```

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.

EXAMPLE:

```
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.

cover_ring ()

Return $R = \mathbf{F}_2[x_1, x_2, ..., x_n]$ if `x_1,x_2,...,x_n` is the ordered list of variable names of this ring. `R` also has the same term ordering as this ring.

EXAMPLE:

```
sage: B.<x,y> = BooleanPolynomialRing(2)
sage: R = B.cover_ring(); R
Multivariate Polynomial Ring in x, y over Finite Field of size 2

sage: B.term_order() == R.term_order()
True
```

The cover ring is cached:

```
sage: B.cover_ring() is B.cover_ring()
True
```

defining_ideal()

Return $I =< x_i^2 + x_i >\subset R$ where `R = self.cover_ring()`, and x_i any element in the set of variables of this ring.

EXAMPLE:

```
sage: B.<x,y> = BooleanPolynomialRing(2)
sage: I = B.defining_ideal(); I
Ideal (x^2 + x, y^2 + y) of Multivariate Polynomial Ring
in x, y over Finite Field of size 2
```

gen(*i=0*)

Returns 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
```

TESTS:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: P.gen(0)
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)
```

gens_dict()

Return a dictionary whose entries are `{var_name:variable,...}`.

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: B.gens_dict()
{'a': a, 'b': b, 'c': c, 'd': d}
```

get_base_order_code()

EXAMPLE:

```
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: B.get_base_order_code()
0

sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing(order='deglex')
sage: B.get_base_order_code()
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()
1
```

Note: This function which is part of the PolyBoRi upstream API works with a current global ring. This notion is avoided in Sage.

get_order_code()

EXAMPLE:

```
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: B.get_order_code()
0

sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing(order='deglex')
sage: B.get_order_code()
1
```

Note: This function which is part of the PolyBoRi upstream API works with a current global ring. This notion is avoided in Sage.

has_degree_order()

Returns checks whether the order code corresponds to a degree ordering.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P.has_degree_order()
False
```

id()

Returns a unique identifiert for this boolean polynomial ring.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: print "id: ", P.id()
id: ...

sage: P = BooleanPolynomialRing(10, 'x')
sage: Q = BooleanPolynomialRing(20, 'x')

sage: P.id() != Q.id()
```

```
True
```

ideal(**gens*, ***kwds*)

Create an ideal in this ring.

INPUT:

- `gens` - list or tuple of generators
- `coerce` - bool (default: True) automatically coerce the given polynomials to this ring to form the ideal

EXAMPLES:

```
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

EXAMPLE:

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, 0, 1, 0), \
                    (0, 0, 1, 1, 1, 1), (1, 0, 1, 1, 1, 1), \
                    (0, 0, 0, 0, 1, 0), (0, 1, 1, 1, 1, 0), \
                    (1, 1, 0, 0, 0, 1), (1, 1, 0, 1, 0, 1)])
sage: ones = set([(0, 0, 0, 0, 0, 0), (1, 0, 1, 0, 1, 0), \
                  (0, 0, 0, 1, 1, 1), (1, 0, 0, 1, 0, 1), \
                  (0, 0, 0, 0, 1, 1), (0, 1, 1, 0, 1, 1), \
                  (0, 1, 1, 1, 1, 1), (1, 1, 1, 0, 1, 0)])
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
```

```
sage: [g(*p) for p in zeros]
[0, 0, 0, 0, 0, 0, 0, 0]
sage: [g(*p) for p in ones]
[1, 1, 1, 1, 1, 1, 1, 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`()

Returns 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`()

Returns 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 random
- `vars_set` - list of integer indicies of generators of self to use in the generated polynomial

EXAMPLES:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.random_element(degree=3, terms=4)
x*y*z + x*z + x + y*z

sage: P.random_element(degree=1, terms=2)
z + 1
```

In corner cases this function will return fewer terms by default:

```
sage: P = BooleanPolynomialRing(2,'y')
sage: P.random_element()
y0*y1 + y0

sage: P = BooleanPolynomialRing(1,'y')
sage: P.random_element()
y
```

We return uniformly random polynomials up to degree 2:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: B.random_element(terms=Infinity)
a*b + a*c + a*d + b*c + b*d + d
```

TESTS:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.random_element(degree=4)
Traceback (most recent call last):
...
ValueError: Given degree should be less than or equal to number of variables (3)

sage: P.random_element(degree=1, terms=5)
y + 1

sage: P.random_element(degree=2, terms=5, vars_set=(0,1))
x*y + y
```

We test that trac ticket #13845 is fixed:

```
sage: n = 10
sage: B = BooleanPolynomialRing(n, 'x')
sage: r = B.random_element(terms=(n/2)**2)
```

remove_var(*order=None*, **var*)

Remove a variable or sequence of variables from this ring.

If `order` is not specified, then the subring inherits the term order of the original ring, if possible.

EXAMPLES:

```
sage: R.<x,y,z,w> = BooleanPolynomialRing()
sage: R.remove_var(z)
Boolean PolynomialRing in x, y, w
sage: R.remove_var(z,x)
Boolean PolynomialRing in y, w
sage: R.remove_var(y,z,x)
Boolean PolynomialRing in w
```

Removing all variables results in the base ring:

```
sage: R.remove_var(y,z,x,w)
Finite Field of size 2
```

If possible, the term order is kept:

> sage: R.<x,y,z,w> = BooleanPolynomialRing(order='deglex') sage: R.remove_var(y).term_order() Degree lexicographic term order
>
> sage: R.<x,y,z,w> = BooleanPolynomialRing(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> = BooleanPolynomialRing(order='deglex(2),deglex(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 or
sage: R.remove_var(x,y,z, order='deglex')
Boolean PolynomialRing in u, v
```

variable (*i=0*)

Returns 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.variable()
x
sage: P.variable(2)
z
sage: m = x.monomials()[0]
sage: P.variable(m)
x
```

TESTS:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: P.variable(0)
x
```

zero ()

EXAMPLES:

```
sage: P.<x0,x1> = BooleanPolynomialRing(2)
sage: P.zero()
0
```

class `sage.rings.polynomial.pbori.`**`BooleanPolynomialVector`**

Bases: `object`

A vector of boolean polynomials.

EXAMPLE:

```
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: from polybori import BooleanPolynomialVector
sage: l = [B.random_element() for _ in range(3)]
sage: v = BooleanPolynomialVector(l)
sage: len(v)
3
sage: v[0]
a*b + a + b*e + c*d + e*f
sage: list(v)
[a*b + a + b*e + c*d + e*f, a*d + c*d + d*f + e + f, a*c + a*e + b*c + c*f + f]
```

`append`(*el*)

Append the element `el` to this vector.

EXAMPLE:

```
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: from polybori import BooleanPolynomialVector
sage: v = BooleanPolynomialVector()
sage: for i in range(5):
...     v.append(B.random_element())

sage: list(v)
[a*b + a + b*e + c*d + e*f, a*d + c*d + d*f + e + f, a*c + a*e + b*c + c*f + f, a*c + a*d +
```

class `sage.rings.polynomial.pbori.`**`BooleanPolynomialVectorIterator`**

Bases: `object`

x.__init__(...) initializes x; see help(type(x)) for signature

`next`()

x.next() -> the next value, or raise StopIteration

class `sage.rings.polynomial.pbori.`**`CCuddNavigator`**

Bases: `object`

x.__init__(...) initializes x; see help(type(x)) for signature

`constant`()

`else_branch`()

`terminal_one`()

`then_branch`()

`value`()

class `sage.rings.polynomial.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.

EXAMPLE:

```
sage: from polybori 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.`**`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

EXAMPLE:

```
sage: from polybori 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 immediatly 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

EXAMPLE:

```
sage: from polybori 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 immediatly.

INPUT:

- p - a polynomial

EXAMPLE:

```
sage: from polybori 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()

EXAMPLE:

```
sage: from polybori 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)

sage: list(gbs)
[a + b]

sage: list(gbs.all_generators())
[a + b, a + c]
```

all_spolys_in_next_degree()

clean_top_by_chain_criterion()

contains_one()

Return True if 1 is in the generating system.

EXAMPLE:

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 polybori 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 polybori 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

EXAMPLE:

```
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: from polybori import GroebnerStrategy
sage: gb = GroebnerStrategy(B)
sage: gb.add_generator(a*c + a*f + d*f + d + f)
sage: gb.add_generator(b*c + b*e + c + d + 1)
sage: gb.add_generator(a*f + a + c + d + 1)
sage: gb.add_generator(a*d + a*e + b*e + c + f)
sage: gb.add_generator(b*d + c + d*f + e + f)
sage: gb.add_generator(a*b + b + c*e + e + 1)
sage: gb.add_generator(a + b + c*d + c*e + 1)

sage: from polybori import BooleanPolynomialVector
sage: V= BooleanPolynomialVector([b*d, a*b])
sage: list(gb.faugere_step_dense(V))
[b + c*e + e + 1, c + d*f + e + f]
```

implications(*i*)

Compute "useful" implied polynomials of `i`-th generator, and add them to the strategy, if it finds any.

INPUT:

- `i` - an index

ll_reduce_all()

Use the built-in ll-encoded `BooleSet` of polynomials with linear lexicographical leading term, which coincides with leading term in current ordering, to reduce the tails of all polynomials in the strategy.

minimalize()

Return a vector of all polynomials with minimal leading terms.

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

EXAMPLE:

```
sage: P = PolynomialRing(GF(2),10, 'x')
sage: B = BooleanPolynomialRing(10,'x')
sage: I = sage.rings.ideal.Cyclic(P)
sage: I = B.ideal([B(f) for f in I.gens()])
sage: gb = I.groebner_basis()

sage: from polybori import GroebnerStrategy

sage: G = GroebnerStrategy(B)
sage: _ = [G.add_generator(f) for f in gb]
sage: G.nf(gb[0])
0
sage: G.nf(gb[0] + 1)
1
sage: G.nf(gb[0]*gb[1])
0
sage: G.nf(gb[0]*B.gen(1))
0
```

Note: The result is only canonical if the generating set is a Groebner basis.

npairs ()

reduction_strategy

select (*m*)

Return the index of the generator which can reduce the monomial m.

INPUT:

- m - a BooleanMonomial

EXAMPLE:

```
sage: B.<a,b,c,d,e> = BooleanPolynomialRing()
sage: f = B.random_element()
sage: g = B.random_element()
sage: from polybori 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_spolys_in_next_degree (*f*, *n*)

some_spolys_in_next_degree (*n*)

suggest_plugin_variable ()

symmGB_F2()

Compute a Groebner basis for the generating system.

Note: This implementation is out of date, but it will revived at some point in time. Use the `groebner_basis()` function instead.

top_sugar()

variable_has_value(*v*)

Computes, whether there exists some polynomial of the form $v + c$ in the Strategy – where `c` is a constant – in the list of generators.

INPUT:

- `v` - the index of a variable

EXAMPLE:: sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing() sage: from polybori import GroebnerStrategy sage: gb = GroebnerStrategy(B) sage: gb.add_generator(a*c + a*f + d*f + d + f) sage: gb.add_generator(b*c + b*e + c + d + 1) sage: gb.add_generator(a*f + a + c + d + 1) sage: gb.add_generator(a*d + a*e + b*e + c + f) sage: gb.add_generator(b*d + c + d*f + e + f) sage: gb.add_generator(a*b + b + c*e + e + 1) sage: gb.variable_has_value(0) False

sage: from polybori 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.`**`MonomialConstruct`**

Bases: `object`

Implements PolyBoRi's `Monomial()` constructor.

class `sage.rings.polynomial.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.

EXAMPLE:

```
sage: from polybori import *
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: fac = MonomialFactory()
sage: fac = MonomialFactory(B)
```

class `sage.rings.polynomial.pbori.`**`PolynomialConstruct`**

Bases: `object`

Implements PolyBoRi's `Polynomial()` constructor.

lead(*x*)

Return the leading monomial of boolean polynomial `x`, with respect to to the order of parent ring.

EXAMPLE:

```
sage: from polybori import *
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: PolynomialConstruct().lead(a)
a
```

class `sage.rings.polynomial.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 to the order of parent ring.

EXAMPLE:

```
sage: from polybori import *
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: PolynomialFactory().lead(a)
a
```

class `sage.rings.polynomial.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.

EXAMPLE:

```
sage: from polybori import *
sage: B.<x,y,z> = BooleanPolynomialRing()
sage: red = ReductionStrategy(B)
sage: red.add_generator(x)
sage: list([f.p for f in red])
[x]
```

TESTS:

Check if #8966 is fixed:

```
sage: red = ReductionStrategy(B)
sage: red.add_generator(None)
Traceback (most recent call last):
...
TypeError: argument must be a BooleanPolynomial.
```

`can_rewrite`(*p*)

Return `True` if `p` can be reduced by the generators of this strategy.

EXAMPLE:

```
sage: from polybori import *
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: red = ReductionStrategy(B)
sage: red.add_generator(a*b + c + 1)
sage: red.add_generator(b*c + d + 1)
sage: red.can_rewrite(a*b + a)
True
sage: red.can_rewrite(b + c)
False
sage: red.can_rewrite(a*d + b*c + d + 1)
True
```

cheap_reductions(*p*)

Peform 'cheap' reductions on p.

INPUT:

•p - a boolean polynomial

EXAMPLE:

```
sage: from polybori import *
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: red = ReductionStrategy(B)
sage: red.add_generator(a*b + c + 1)
sage: red.add_generator(b*c + d + 1)
sage: red.add_generator(a)
sage: red.cheap_reductions(a*b + a)
0
sage: red.cheap_reductions(b + c)
b + c
sage: red.cheap_reductions(a*d + b*c + d + 1)
b*c + d + 1
```

head_normal_form(*p*)

Compute the normal form of p with respect to the generators of this strategy but do not perform tail any reductions.

INPUT:

•p - a polynomial

EXAMPLE:

```
sage: from polybori import *
sage: B.<x,y,z> = BooleanPolynomialRing()
sage: red = ReductionStrategy(B)
sage: red.opt_red_tail = True
sage: red.add_generator(x + y + 1)
sage: red.add_generator(y*z + z)

sage: red.head_normal_form(x + y*z)
y + z + 1

sage; red.nf(x + y*z)
y + z + 1
```

nf(*p*)

Compute the normal form of p w.r.t. to the generators of this reduction strategy object.

EXAMPLE:

```
sage: from polybori import *
sage: B.<x,y,z> = BooleanPolynomialRing()
sage: red = ReductionStrategy(B)
sage: red.add_generator(x + y + 1)
sage: red.add_generator(y*z + z)
sage: red.nf(x)
y + 1

sage: red.nf(y*z + x)
y + z + 1
```

reduced_normal_form(*p*)

Compute the normal form of `p` with respect to the generators of this strategy and perform tail reductions.

INPUT:

- `p` - a polynomial

EXAMPLE:

```
sage: from polybori 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.`**TermOrder_from_pb_order**(*n*, *order*, *blocks*)

class `sage.rings.polynomial.pbori.`**VariableBlock**

Bases: `object`

class `sage.rings.polynomial.pbori.`**VariableConstruct**

Bases: `object`

Implements PolyBoRi's `Variable()` constructor.

class `sage.rings.polynomial.pbori.`**VariableFactory**

Bases: `object`

a variable factory for given ring

`sage.rings.polynomial.pbori.`**add_up_polynomials**(*v*, *init*)

Add up all entries in the vector `v`.

INPUT:

- `v` - a vector of boolean polynomials

EXAMPLE:

```
sage: from polybori import *
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: v = BooleanPolynomialVector()
sage: l = [B.random_element() for _ in range(5)]
sage: _ = [v.append(e) for e in l]
sage: add_up_polynomials(v, B.zero())
a*d + b*c + b*d + c + 1
sage: sum(l)
a*d + b*c + b*d + c + 1
```

`sage.rings.polynomial.pbori.`**contained_vars**(*m*)

`sage.rings.polynomial.pbori.`**easy_linear_factors**(*p*)

`sage.rings.polynomial.pbori.`**gauss_on_polys**(*inp*)

Perform Gaussian elimination on the input list of polynomials.

INPUT:

- `inp` - an iterable

EXAMPLE:

```
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: from polybori import *
sage: l = [B.random_element() for _ in range(B.ngens())]
sage: A,v = Sequence(l,B).coefficient_matrix()
sage: A
[1 0 0 0 0 1 0 0 1 1 0 0 0 0 1 0 0 0]
[0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0]
[0 1 0 1 0 0 1 0 0 0 1 0 0 0 0 0 1 0]
[0 1 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 1 0 1 0 1 0 0 0 0 0 1]
[0 1 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 1]

sage: e = gauss_on_polys(l)
sage: E,v = Sequence(e,B).coefficient_matrix()
sage: E
[1 0 0 0 0 1 0 0 1 1 0 0 0 0 1 0 0 0]
[0 1 0 0 0 0 0 0 1 1 1 0 1 1 0 1 0 1]
[0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0]
[0 0 0 1 0 0 0 1 1 0 0 1 1 1 0 1 1 0]
[0 0 0 0 1 0 0 1 1 0 1 1 0 1 0 1 0 1]
[0 0 0 0 0 0 1 1 0 1 0 1 0 0 0 0 0 1]

sage: A.echelon_form()
[1 0 0 0 0 1 0 0 1 1 0 0 0 0 1 0 0 0]
[0 1 0 0 0 0 0 0 1 1 1 0 1 1 0 1 0 1]
[0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0]
[0 0 0 1 0 0 0 1 1 0 0 1 1 1 0 1 1 0]
[0 0 0 0 1 0 0 1 1 0 1 1 0 1 0 1 0 1]
[0 0 0 0 0 0 1 1 0 1 0 1 0 0 0 0 0 1]
```

`sage.rings.polynomial.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,y> = QQ[]
sage: sage.rings.polynomial.pbori.get_var_mapping(P,R)
[z, y]
sage: sage.rings.polynomial.pbori.get_var_mapping(P, z^2)
[z, None]

sage: R.<z,x> = BooleanPolynomialRing(2)
sage: sage.rings.polynomial.pbori.get_var_mapping(P,R)
[z, x]
sage: sage.rings.polynomial.pbori.get_var_mapping(P, x^2)
[None, x]
```

`sage.rings.polynomial.pbori.`**`if_then_else`**(*root*, *a*, *b*)

The opposite of navigating down a ZDD using navigators is to construct new ZDDs in the same way, namely giving their else- and then-branch as well as the index value of the new node.

INPUT:

- `root` - a variable

- a - the if branch, a BooleSet or a BoolePolynomial
- b - the else branch, a BooleSet or a BoolePolynomial

EXAMPLE:

```
sage: from polybori 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}}

sage: if_then_else(x1.lm().index(),f0,f1)
{{x1,x2,x3}, {x1,x3}, {x4}}

sage: if_then_else(x5, f0, f1)
Traceback (most recent call last):
...
IndexError: index of root must be less than the values of roots of the branches.
```

sage.rings.polynomial.pbori.**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

EXAMPLE:

```
sage: B = BooleanPolynomialRing(4,"x0,x1,x2,x3")
sage: x = B.gen
sage: from polybori.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.**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

EXAMPLE:

Let V be a set of points in $\mathbf{F}_2^n$ 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:

```
sage: B = BooleanPolynomialRing(4,"x0,x1,x2,x3")
sage: x = B.gen
sage: from polybori.interpolate import *
sage: V=(x(0)+x(1)+x(2)+x(3)+1).set()
```

We take V = {e0,e1,e2,e3,0}, where ei describes the i-th unit vector. For our considerations it does not play any role, if we suppose V to be embedded in $\mathbf{F}_2^4$ or a vector space of higher dimension:

```
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:

```
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.`**`ll_red_nf_noredsb`**`(`*p*, *reductors*`)`

Redude 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.

EXAMPLE:

```
sage: from polybori 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.rings.polynomial.pbori.**ll_red_nf_noredsb_single_recursive_call**(*p*, *reductors*)

Redude the polynomial `p` by the set of `reductors` with linear leading terms.

`ll_red_nf_noredsb_single_recursive()` call has the same specification as `ll_red_nf_noredsb()`, but a different implementation: It is very sensitive to the ordering of variables, however it has the property, that it needs just one recursive call.

INPUT:

- `p` - a boolean polynomial
- `reductors` - a boolean set encoding a Groebner basis with linear leading terms.

EXAMPLE:

```
sage: from polybori import ll_red_nf_noredsb_single_recursive_call
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: p = a*b + c + d + 1
sage: f,g  = a + c + 1, b + d + 1;
sage: reductors = f.set().union( g.set() )
sage: ll_red_nf_noredsb_single_recursive_call(p, reductors)
b*c + b*d + c + d + 1
```

sage.rings.polynomial.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.

EXAMPLE:

```
sage: from polybori import ll_red_nf_redsb
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_redsb(p, reductors)
b*c + b*d + c + d + 1
```

sage.rings.polynomial.pbori.**map_every_x_to_x_plus_one**(*p*)

Map every variable `x_i` in this polynomial to `x_i + 1`.

EXAMPLE:

```
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: f = a*b + z + 1; f
a*b + z + 1
sage: from polybori 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.rings.polynomial.pbori.**mod_mon_set**(*a_s*, *v_s*)

sage.rings.polynomial.pbori.**mod_var_set**(*a*, *v*)

sage.rings.polynomial.pbori.**mult_fact_sim_C**(*v*, *ring*) sage.rings.polynomial.pbori.**nf3**(*s*, *p*, *m*)

sage.rings.polynomial.pbori.**parallel_reduce**(*inp*, *strat*, *average_steps*, *delay_f*)

sage.rings.polynomial.pbori.**random_set**(*variables*, *length*)

Return a random set of monomials with `length` elements with each element in the variables `variables`.

EXAMPLE:

```
sage: from polybori import random_set, set_random_seed
sage: B.<a,b,c,d,e> = BooleanPolynomialRing()
sage: (a*b*c*d).lm()
a*b*c*d
sage: set_random_seed(1337)
sage: random_set((a*b*c*d).lm(),10)
{{a,b,c,d}, {a,b}, {a,c,d}, {a,c}, {b,c,d}, {b,d}, {b}, {c,d}, {c}, {d}}
```

sage.rings.polynomial.pbori.**recursively_insert**(*n*, *ind*, *m*)

sage.rings.polynomial.pbori.**red_tail**(*s*, *p*)

Perform tail reduction on `p` using the generators of `s`.

INPUT:

- `s` - a reduction strategy
- `p` - a polynomial

EXAMPLE:

```
sage: from polybori import *
sage: B.<x,y,z> = BooleanPolynomialRing()
sage: red = ReductionStrategy(B)
sage: red.add_generator(x + y + 1)
sage: red.add_generator(y*z + z)
sage: red_tail(red,x)
x
sage: red_tail(red,x*y + x)
x*y + y + 1
```

sage.rings.polynomial.pbori.**set_random_seed**(*seed*)

The the PolyBoRi random seed to `seed`

EXAMPLE:

```
sage: from polybori import random_set, set_random_seed
sage: B.<a,b,c,d,e> = BooleanPolynomialRing()
sage: (a*b*c*d).lm()
a*b*c*d
sage: set_random_seed(1337)
sage: random_set((a*b*c*d).lm(),2)
{{b}, {c}}
sage: random_set((a*b*c*d).lm(),2)
{{a,c,d}, {c}}

sage: set_random_seed(1337)
sage: random_set((a*b*c*d).lm(),2)
{{b}, {c}}
sage: random_set((a*b*c*d).lm(),2)
{{a,c,d}, {c}}
```

sage.rings.polynomial.pbori.**substitute_variables**(*parent*, *vec*, *poly*)

`var(i)` is replaced by `vec[i]` in `poly`.

EXAMPLE:

```
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: f = a*b + c + 1
sage: from polybori import substitute_variables
sage: substitute_variables(B, [a,b,c],f)
a*b + c + 1
sage: substitute_variables(B, [a+1,b,c],f)
a*b + b + c + 1
sage: substitute_variables(B, [a+1,b+1,c],f)
a*b + a + b + c
sage: substitute_variables(B, [a+1,b+1,B(0)],f)
a*b + a + b
```

Substitution is also allowed with different rings:

```
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: f = a*b + c + 1
sage: B.<w,x,y,z> = BooleanPolynomialRing(order='deglex')

sage: from polybori import substitute_variables
sage: substitute_variables(B, [x,y,z], f) * w
w*x*y + w*z + w
```

sage.rings.polynomial.pbori.**top_index**(*s*)

Return the highest index in the parameter `s`.

INPUT:

- `s` - `BooleSet`, `BooleMonomial`, `BoolePolynomial`

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: from polybori 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.**unpickle_BooleanPolynomial**(*ring*, *string*)

Unpickle boolean polynomials

EXAMPLE:

```
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.**unpickle_BooleanPolynomial0**(*ring*, *l*)

Unpickle boolean polynomials

EXAMPLE:

```
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.`**`unpickle_BooleanPolynomialRing`**(*n*, *names*, *order*)

Unpickle boolean polynomial rings.

EXAMPLE:

```
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.`**`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`

EXAMPLE:

```
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 polybori 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),)
```

CHAPTER

SEVEN

EDUCATIONAL VERSIONS OF GROEBNER BASIS AND RELATED ALGORITHMS

7.1 Educational Versions of Groebner Basis Algorithms.

Following [BW93] the original Buchberger algorithm (c.f. algorithm GROEBNER in [BW93]) and an improved version of Buchberger's algorithm (c.g. algorithm GROEBNERNEW2 in [BW93]) 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 in Sage efficiently use the `sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal.groebner_basis()` method on multivariate polynomial objects.

Note: The notion of 'term' and 'monomial' in [BW93] 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 [BW93] 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 [BW93].

EXAMPLES:

Consider Katsura-6 w.r.t. a `degrevlex` ordering.:

```
sage: from sage.rings.polynomial.toy_buchberger import *
sage: P.<a,b,c,e,f,g,h,i,j,k> = PolynomialRing(GF(32003),10)
sage: I = sage.rings.ideal.Katsura(P,6)

sage: g1 = buchberger(I)
sage: g2 = buchberger_improved(I)
sage: g3 = I.groebner_basis()
```

All algorithms actually compute a Groebner basis:

```
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:

```
sage: Ideal(g1) == Ideal(g2) == Ideal(g3)
True
```

If `get_verbose()` is >= 1 a protocol is provided:

```
sage: set_verbose(1)
sage: P.<a,b,c> = PolynomialRing(GF(127),3)
sage: I = sage.rings.ideal.Katsura(P)
// sage... ideal

sage: I
Ideal (a + 2*b + 2*c - 1, a^2 + 2*b^2 + 2*c^2 - a, 2*a*b + 2*b*c - b) of Multivariate Polynomial Ring
```

The original Buchberger algorithm performs 15 useless reductions to zero for this example:

```
sage: buchberger(I)
(a + 2*b + 2*c - 1, a^2 + 2*b^2 + 2*c^2 - a) => -2*b^2 - 6*b*c - 6*c^2 + b + 2*c
G: set([a + 2*b + 2*c - 1, 2*a*b + 2*b*c - b, a^2 + 2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2

(a^2 + 2*b^2 + 2*c^2 - a, a + 2*b + 2*c - 1) => 0
G: set([a + 2*b + 2*c - 1, 2*a*b + 2*b*c - b, a^2 + 2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2

(a + 2*b + 2*c - 1, 2*a*b + 2*b*c - b) => -5*b*c - 6*c^2 - 63*b + 2*c
G: set([a + 2*b + 2*c - 1, 2*a*b + 2*b*c - b, -5*b*c - 6*c^2 - 63*b + 2*c, a^2 + 2*b^2 + 2*c^2 - a, -

(2*a*b + 2*b*c - b, a + 2*b + 2*c - 1) => 0
G: set([a + 2*b + 2*c - 1, 2*a*b + 2*b*c - b, -5*b*c - 6*c^2 - 63*b + 2*c, a^2 + 2*b^2 + 2*c^2 - a, -

(2*a*b + 2*b*c - b, -5*b*c - 6*c^2 - 63*b + 2*c) => -22*c^3 + 24*c^2 - 60*b - 62*c
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 + 2*b^2 + 2*c^2 - a

(2*a*b + 2*b*c - b, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 + 2*b^2 + 2*c^2 - a

(2*a*b + 2*b*c - b, a^2 + 2*b^2 + 2*c^2 - a) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 + 2*b^2 + 2*c^2 - a

(a + 2*b + 2*c - 1, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 + 2*b^2 + 2*c^2 - a

(a^2 + 2*b^2 + 2*c^2 - a, 2*a*b + 2*b*c - b) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*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) => 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

(a + 2*b + 2*c - 1, -5*b*c - 6*c^2 - 63*b + 2*c) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 + 2*b^2 + 2*c^2 - a

(a^2 + 2*b^2 + 2*c^2 - a, -5*b*c - 6*c^2 - 63*b + 2*c) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 + 2*b^2 + 2*c^2 - a

(-5*b*c - 6*c^2 - 63*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

(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

(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, -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*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

(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

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
```

The 'improved' Buchberger algorithm in contrast only performs 3 reductions to zero:

```
sage: buchberger_improved(I)
(b^2 - 26*c^2 - 51*b + 51*c, b*c + 52*c^2 + 38*b + 25*c) => 11*c^3 - 12*c^2 + 30*b + 31*c
G: set([a + 2*b + 2*c - 1, b^2 - 26*c^2 - 51*b + 51*c, 11*c^3 - 12*c^2 + 30*b + 31*c, b*c + 52*c^2 +

(11*c^3 - 12*c^2 + 30*b + 31*c, b*c + 52*c^2 + 38*b + 25*c) => 0
G: set([a + 2*b + 2*c - 1, b^2 - 26*c^2 - 51*b + 51*c, 11*c^3 - 12*c^2 + 30*b + 31*c, b*c + 52*c^2 +

1 reductions to zero.
[a + 2*b + 2*c - 1, b^2 - 26*c^2 - 51*b + 51*c, c^3 + 22*c^2 - 55*b + 49*c, b*c + 52*c^2 + 38*b + 25
```

REFERENCES:

AUTHOR:

- Martin Albrecht (2007-05-24): initial version
- Marshall Hampton (2009-07-08): some doctest additions

`sage.rings.polynomial.toy_buchberger.`**LCM**(*f*, *g*)
x.__init__(...) initializes x; see help(type(x)) for signature

`sage.rings.polynomial.toy_buchberger.`**LM**(*f*)
x.__init__(...) initializes x; see help(type(x)) for signature

`sage.rings.polynomial.toy_buchberger.`**LT**(*f*)
x.__init__(...) initializes x; see help(type(x)) for signature

`sage.rings.polynomial.toy_buchberger.`**buchberger**(*F*)
The original version of Buchberger's algorithm as presented in [BW93], 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:

```
sage: from sage.rings.polynomial.toy_buchberger import buchberger
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: set_verbose(0)
sage: buchberger(R.ideal([x^2 - z - 1, z^2 - y - 1, x*y^2 - x - 1]))
[-y^3 + x*z - x + y, y^2*z + y^2 - x - z - 1, x*y^2 - x - 1, x^2 - z - 1, z^2 - y - 1]
```

`sage.rings.polynomial.toy_buchberger.`**`buchberger_improved`**`(F)`

An improved version of Buchberger's algorithm as presented in [BW93], 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:

```
sage: from sage.rings.polynomial.toy_buchberger import buchberger_improved
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: set_verbose(0)
sage: buchberger_improved(R.ideal([x^4-y-z,x*y*z-1]))
[x*y*z - 1, x^3 - y^2*z - y*z^2, y^3*z^2 + y^2*z^3 - x^2]
```

`sage.rings.polynomial.toy_buchberger.`**`inter_reduction`**`(Q)`

If `Q` is the set $(f_1, ..., f_n)$ this method returns $(g_1, ..., g_s)$ such that:

- $< f_1, ..., f_n >=< g_1, ..., g_s >$
- $LM(g_i)! = LM(g_j)$ for all $i! = j$
- $LM(g_i)$ 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.

INPUT:

- `Q` - a set of polynomials

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]))
sage: reduced == set([x*y^2, x^2-5*y^2])
True
sage: reduced == inter_reduction(set([2*(x^2-5*y^2),x^3]))
True
```

`sage.rings.polynomial.toy_buchberger.`**`select`**`(P)`

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,3, 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)`

Computes the S-polynomial of f and g.

INPUT:

•`f`, `g` - polynomials

OUTPUT:

•The S-polynomial of f and g.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
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 list of critical pairs `B` and the polynomial `h` as presented in [BW93], 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 list of critical pairs

•`h` - a polynomial

OUTPUT:

a tuple of an intermediate Groebner basis and a list of critical pairs

EXAMPLES:

```
sage: from sage.rings.polynomial.toy_buchberger import update
sage: R.<x,y,z> = PolynomialRing(QQ,3)
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 = update(G,B,x*y^2-1)
sage: G,B
({x*y*z - 1, x*y^2 - 1}, {(x*y^2 - 1, x*y*z - 1)})
```

7.2 Educational Versions of Groebner Basis Algorithms: Triangular Factorization.

In this file is the implementation of two algorithms in [Laz92].

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`.

REFERENCES:

`sage.rings.polynomial.toy_variety.`**`coefficient_matrix`**(*polys*)

Generates the matrix `M` whose entries are the coefficients of `polys`. The entries of row `i` of `M` consist of the coefficients of `polys[i]`.

INPUT:

•`polys` - a list/tuple of polynomials

OUTPUT:

A matrix `M` of the coefficients of `polys`.

EXAMPLE:

```
sage: from sage.rings.polynomial.toy_variety import coefficient_matrix
sage: R.<x,y> = PolynomialRing(QQ)
sage: coefficient_matrix([x^2 + 1, y^2 + 1, x*y + 1])
[1 0 0 1]
[0 0 1 1]
[0 1 0 1]
```

Note: This function may be merged with `sage.rings.polynomial.multi_polynomial_sequence.PolynomialS` in the future.

`sage.rings.polynomial.toy_variety.`**`elim_pol`**(*B*, *n=-1*)

Finds 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 [Laz92].

INPUT:

•`B` - a list/tuple of polynomials or a multivariate polynomial ideal

•`n` - the variable to check (see above) (default: `-1`)

EXAMPLE:

```
sage: set_verbose(0)
sage: from sage.rings.polynomial.toy_variety import elim_pol
sage: R.<x,y,z> = PolynomialRing(GF(32003))
sage: p1 = x^2*(x-1)^3*y^2*(z-3)^3
sage: p2 = z^2 - z
sage: p3 = (x-2)^2*(y-1)^3
sage: I = R.ideal(p1,p2,p3)
sage: elim_pol(I.groebner_basis())
z^2 - z
```

sage.rings.polynomial.toy_variety.**is_linearly_dependent**(*polys*)

Decides whether the polynomials of `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.

EXAMPLE:

```
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: p = 3*B[0] - 2*B[1] + B[2]
sage: is_linearly_dependent(B + [p])
True
sage: p = x*B[0]
sage: is_linearly_dependent(B + [p])
False
sage: is_linearly_dependent([])
False
```

sage.rings.polynomial.toy_variety.**is_triangular**(*B*)

Check whether the basis `B` of an ideal is triangular. That is: check whether the largest variable in `B[i]` with respect to the ordering of the base ring `R` is `R.gens()[i]`.

The algorithm is based on the definition of a triangular basis, given by Lazard in 1992 in [Laz92].

INPUT:

- `B` - a list/tuple of polynomials or a multivariate polynomial ideal

OUTPUT:

`True` if the basis is triangular; `False` otherwise.

EXAMPLE:

```
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 polys, determines coefficients that describe the linear combination. This probably doesn't work for any inputs except p, a polynomial, and polys, a sequence of polynomials. If p is not in fact a linear combination of polys, the function raises an exception.

The algorithm creates a matrix of coefficients of the monomials of 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
- polys - a list/tuple of polynomials

OUTPUT:

If n == len(polys), returns [a[0],a[1],...,a[n-1]] such that p == a[0]*poly[0] + ... + a[n-1]*poly[n-1].

EXAMPLE:

```
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: p = 3*B[0] - 2*B[1] + B[2]
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 [Laz92]. 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_0, T_1, etc.

EXAMPLE:

```
sage: set_verbose(0)
sage: from sage.rings.polynomial.toy_variety import triangular_factorization
sage: R.<x,y,z> = PolynomialRing(GF(32003))
sage: p1 = x^2*(x-1)^3*y^2*(z-3)^3
sage: p2 = z^2 - z
```

```
sage: p3 = (x-2)^2*(y-1)^3
sage: I = R.ideal(p1,p2,p3)
sage: triangular_factorization(I.groebner_basis())
[[x^2 - 4*x + 4, y, z],
 [x^5 - 3*x^4 + 3*x^3 - x^2, y - 1, z],
 [x^2 - 4*x + 4, y, z - 1],
 [x^5 - 3*x^4 + 3*x^3 - x^2, y - 1, z - 1]]
```

7.3 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 [BW93] 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 [BW93] 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 [BW93].

EXAMPLE:

```
sage: from sage.rings.polynomial.toy_d_basis import d_basis
```

First, consider an example from arithmetic geometry:

```
sage: A.<x,y> = PolynomialRing(ZZ, 2)
sage: B.<X,Y> = PolynomialRing(Rationals(),2)
sage: f = -y^2 - y + x^3 + 7*x + 1
sage: fx = f.derivative(x)
sage: fy = f.derivative(y)
sage: I = B.ideal([B(f),B(fx),B(fy)])
sage: I.groebner_basis()
[1]
```

Since the output is 1, we know that there are no generic singularities.

To look at the singularities of the arithmetic surface, we need to do the corresponding computation over **Z**:

```
sage: I = A.ideal([f,fx,fy])
sage: gb = d_basis(I); gb
[x - 2020, y - 11313, 22627]

sage: gb[-1].factor()
11^3 * 17
```

This Groebner Basis gives a lot of information. First, the only fibers (over **Z**) that are not smooth are at 11 = 0, and 17 = 0. Examining the Groebner Basis, we see that we have a simple node in both the fiber at 11 and at 17. From the factorization, we see that the node at 17 is regular on the surface (an I_1 node), but the node at 11 is not. After blowing up this non-regular point, we find that it is an I_3 node.

Another example. This one is from the Magma Handbook:

```
sage: P.<x, y, z> = PolynomialRing(IntegerRing(), 3, order='lex')
sage: I = ideal( x^2 - 1, y^2 - 1, 2*x*y - z)
sage: I = Ideal(d_basis(I))
sage: x.reduce(I)
```

```
x
sage: (2*x).reduce(I)
y*z
```

To compute modulo 4, we can add the generator 4 to our basis.:

```
sage: I = ideal( x^2 - 1, y^2 - 1, 2*x*y - z, 4)
sage: gb = d_basis(I)
sage: R = P.change_ring(IntegerModRing(4))
sage: gb = [R(f) for f in gb if R(f)]; gb
[x^2 - 1, x*z + 2*y, 2*x - y*z, y^2 - 1, z^2, 2*z]
```

A third example is also from the Magma Handbook.

This example shows how one can use Groebner bases over the integers to find the primes modulo which a system of equations has a solution, when the system has no solutions over the rationals.

We first form a certain ideal I in $\mathbf{Z}[x, y, z]$, and note that the Groebner basis of I over $\mathbf{Q}$ contains 1, so there are no solutions over $\mathbf{Q}$ or an algebraic closure of it (this is not surprising as there are 4 equations in 3 unknowns).:

```
sage: P.<x, y, z> = PolynomialRing(IntegerRing(), 3)
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 $\mathbf{Z}$), we note that there is a certain integer in the ideal which is not 1.:

```
sage: d_basis(I) # random -- waiting on upstream singular fixes at #6051
[x + 170269749119, y + 2149906854, z + ..., 282687803443]
```

Now for each prime p dividing this integer 282687803443, the Groebner basis of I modulo p will be non-trivial and will thus give a solution of the original system modulo p.:

```
sage: factor(282687803443)
101 * 103 * 27173681

sage: I.change_ring( P.change_ring( GF(101) ) ).groebner_basis()
[x + 19, y + 48, z - 33]

sage: I.change_ring( P.change_ring( GF(103) ) ).groebner_basis()
[x + 39, y + 8, z - 18]

sage: I.change_ring( P.change_ring( GF(27173681) ) ).groebner_basis()
[x - 536027, y + 3186055, z + 10380032]
```

Of course, modulo any other prime the Groebner basis is trivial so there are no other solutions. For example:

```
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*`)`

x.__init__(...) initializes x; see help(type(x)) for signature

`sage.rings.polynomial.toy_d_basis.`**`LM`**`(`*f*`)`

x.__init__(...) initializes x; see help(type(x)) for signature `sage.rings.polynomial.toy_d_basis.`**`d_basis`**`(`*F*, *strat=True*`)`

Return the d-basis for the Ideal `F` as defined in [BW93].

INPUT:

- `F` - an ideal
- `strat` - use update strategy (default: `True`)

EXAMPLE:

```
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 G-Polynomial of `g_1` and `g_2`.

Let $a_i t_i$ be $LT(g_i)$, $a = a_i * c_i + a_j * c_j$ 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: $c_1 s_1 g_1 - c_2 s_2 g_2$.

INPUT:

- `g1` - polynomial
- `g2` - polynomial

EXAMPLE:

```
sage: from sage.rings.polynomial.toy_d_basis import gpol
sage: P.<x, y, z> = PolynomialRing(IntegerRing(), 3, order='lex')
sage: f = x^2 - 1
sage: g = 2*x*y - z
sage: gpol(f,g)
x^2*y - y
```

`sage.rings.polynomial.toy_d_basis.`**`select`**`(`*P*`)`

The normal selection strategy.

INPUT:

- `P` - a list of critical pairs

OUTPUT: an element of P

EXAMPLE:

```
sage: from sage.rings.polynomial.toy_d_basis import select
sage: A.<x,y> = PolynomialRing(ZZ, 2)
sage: f = -y^2 - y + x^3 + 7*x + 1
sage: fx = f.derivative(x)
sage: fy = f.derivative(y)
sage: G = [f, fx, fy]
sage: B = set(filter(lambda (x,y): x!=y, [(f1,f2) for f1 in G for f2 in G]))
sage: select(B)
(-2*y - 1, 3*x^2 + 7)
```

`sage.rings.polynomial.toy_d_basis.`**`spol`**`(`*g1*, *g2*`)`

Return S-Polynomial of `g_1` and `g_2`.

Let $a_i t_i$ be $LT(g_i)$, $b_i = a/a_i$ 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:

- `g1` - polynomial
- `g2` - polynomial

EXAMPLE:

```
sage: from sage.rings.polynomial.toy_d_basis import spol
sage: P.<x, y, z> = PolynomialRing(IntegerRing(), 3, order='lex')
sage: f = x^2 - 1
sage: g = 2*x*y - z
sage: spol(f,g)
x*z - 2*y
```

`sage.rings.polynomial.toy_d_basis.`**`update`**`(`*G*, *B*, *h*`)`

Update `G` using the list of critical pairs `B` and the polynomial `h` as presented in [BW93], 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

EXAMPLE:

```
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)})
```

CHAPTER

EIGHT

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 $x = 2^k * y$ for some $k \geq 0$, we require that $R(x/2^k)$ returns y.)

The main function to be exported is convolution().

EXAMPLES:

```
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:

```
sage: v=convolution(range(1000), 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*`)`

Returns 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]

sage: R = Integers(47)
sage: L1 = [R.random_element() for _ in range(1000)]
sage: L2 = [R.random_element() for _ in range(3756)]
sage: L3 = convolution(L1, L2)
sage: L3[2000] == sum([L1[i] * L2[2000-i] for i in range(1000)])
True
sage: len(L3) == 1000 + 3756 - 1
True
```

CHAPTER

NINE

FAST CALCULATION OF CYCLOTOMIC POLYNOMIALS

This module provides a function `cyclotomic_coeffs()`, which calculates the coefficients of cyclotomic polynomials. This is not intended to be invoked directly by the user, but it is called by the method `cyclotomic_polynomial()` method of univariate polynomial ring objects and the top-level `cyclotomic_polynomial()` function.

`sage.rings.polynomial.cyclotomic.`**`bateman_bound`**`(`*nn*`)`

`sage.rings.polynomial.cyclotomic.`**`cyclotomic_coeffs`**`(`*nn*, *sparse=None*`)`

This calculates 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 Moebius 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:

```
sage: euler_phi(30)
8
sage: R(cyclotomic_coeffs(14)).factor()
x^6 - x^5 + x^4 - x^3 + x^2 - x + 1
```

The coefficients are not always +/-1:

```
sage: cyclotomic_coeffs(105)
[1, 1, 1, 0, 0, -1, -1, -2, -1, -1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, -1, 0, -1, 0, -1, 0, -1, 0, -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)
sage: max(v)
14102773
```

The polynomial is a palindrome for any n:

```
sage: n = ZZ.random_element(50000)
sage: factor(n)
3 * 10009
sage: v = cyclotomic_coeffs(n, sparse=False)
sage: v == list(reversed(v))
True
```

AUTHORS:

- Robert Bradshaw (2007-10-27): initial version (inspired by work of Andrew Arnold and Michael Monagan)

`sage.rings.polynomial.cyclotomic.`**`cyclotomic_value`**`(n, x)`

Returns the value of the n-th cyclotomic polynomial evaulated 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 Φ_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 μ is the Moebius function.

- Handles the case that x^d = 1 for some d, but not the case that x^d - 1 is non-invertible: in this case polynomial evaluation is used instead.

EXAMPLES:

```
sage: cyclotomic_value(51, 3)
1282860140677441
sage: cyclotomic_polynomial(51)(3)
1282860140677441
```

It works for non-integral values as well:

```
sage: cyclotomic_value(144, 4/3)
79148745433504023621920372161/79766443076872509863361
sage: cyclotomic_polynomial(144)(4/3)
79148745433504023621920372161/79766443076872509863361
```

TESTS:

```
sage: R.<x> = QQ[]
sage: K.<i> = NumberField(x^2 + 1)
sage: for y in [-1, 0, 1, 2, 1/2, Mod(3, 8), Mod(3,11), GF(9,'a').gen(), Zp(3)(54), i, x^2+2]:
....:     for n in [1..60]:
....:         val1 = cyclotomic_value(n, y)
....:         val2 = cyclotomic_polynomial(n)(y)
....:         if val1 != val2:
....:             print "Wrong value for cyclotomic_value(%s, %s) in %s"%(n,y,parent(y))
....:         if val1.parent() is not val2.parent():
....:             print "Wrong parent for cyclotomic_value(%s, %s) in %s"%(n,y,parent(y))

sage: cyclotomic_value(20, I)
5
sage: a = cyclotomic_value(10, mod(3, 11)); a
6
sage: a.parent()
Ring of integers modulo 11
sage: cyclotomic_value(30, -1.0)
1.00000000000000
sage: S.<t> = R.quotient(R.cyclotomic_polynomial(15))
sage: cyclotomic_value(15, t)
0
sage: cyclotomic_value(30, t)
2*t^7 - 2*t^5 - 2*t^3 + 2*t
sage: S.<t> = R.quotient(x^10)
sage: cyclotomic_value(2^128-1, t)
-t^7 - t^6 - t^5 + t^2 + t + 1
sage: cyclotomic_value(10,mod(3,4))
1
```

sage.rings.polynomial.cyclotomic.**my_cmp**(*a*, *b*)

CHAPTER

TEN

INDICES AND TABLES

[MF99] J.H. Mathews and K.D. Fink. *Numerical Methods Using MATLAB*. 3rd edition, Prentice-Hall, 1999.

[BF05] R.L. Burden and J.D. Faires. *Numerical Analysis*. Thomson Brooks/Cole, 8th edition, 2005.

[BD89] R. J. Bradford and J. H. Davenport, Effective tests for cyclotomic polynomials, Symbolic and Algebraic Computation (1989) pp. 244 – 251, doi:10.1007/3-540-51084-2_22

[CLO] D. Cox, J. Little, D. O'Shea. Using Algebraic Geometry. Springer, 2005.

[Can] J. Canny. Generalised characteristic polynomials. J. Symbolic Comput. Vol. 9, No. 3, 1990, 241–250.

[Mac] F.S. Macaulay. The algebraic theory of modular systems Cambridge university press, 1916.

[BFS04] Magali Bardet, Jean-Charles Faugère, and Bruno Salvy, On the complexity of Groebner basis computation of semi-regular overdetermined algebraic equations. Proc. International Conference on Polynomial System Solving (ICPSS), pp. 71-75, 2004.

[BPW06] J. Buchmann, A. Pychkine, R.-P. Weinmann *Block Ciphers Sensitive to Groebner Basis Attacks* in Topics in Cryptology – CT RSA'06; LNCS 3860; pp. 313–331; Springer Verlag 2006; pre-print available at http://eprint.iacr.org/2005/200

[CBJ07] Gregory V. Bard, and Nicolas T. Courtois, and Chris Jefferson. *Efficient Methods for Conversion and Solution of Sparse Systems of Low-Degree Multivariate Polynomials over GF(2) via SAT-Solvers*. Cryptology ePrint Archive: Report 2007/024. available at http://eprint.iacr.org/2007/024

[AB2007] M. Aschenbrenner, C. Hillar, Finite generation of symmetric ideals. Trans. Amer. Math. Soc. 359 (2007), no. 11, 5171–5192.

[AB2008] M. Aschenbrenner, C. Hillar, An Algorithm for Finding Symmetric Groebner Bases in Infinite Dimensional Rings.

[BD07] Michael Brickenstein, Alexander Dreyer; *PolyBoRi: A Groebner basis framework for Boolean polynomials*; pre-print available at http://www.itwm.fraunhofer.de/fileadmin/ITWM-Media/Zentral/Pdf/Berichte_ITWM/2007/bericht122.pdf

[BW93] Thomas Becker and Volker Weispfenning. *Groebner Bases - A Computational Approach To Commutative Algebra*. Springer, New York 1993.

[Laz92] Daniel Lazard, *Solving Zero-dimensional Algebraic Systems*, in Journal of Symbolic Computation (1992) vol. 13, pp. 117-131