<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Polynomial Rings</td>
<td>1.1 Constructors for polynomial rings</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Univariate Polynomials</td>
<td>2.1 Univariate Polynomials and Polynomial Rings</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.2 Generic Convolution</td>
<td>237</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.3 Fast calculation of cyclotomic polynomials</td>
<td>238</td>
</tr>
<tr>
<td>3</td>
<td>Multivariate Polynomials</td>
<td>3.1 Multivariate Polynomials and Polynomial Rings</td>
<td>241</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.2 Classical Invariant Theory</td>
<td>407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.3 Educational Versions of Groebner Basis Related Algorithms</td>
<td>453</td>
</tr>
<tr>
<td>4</td>
<td>Skew Polynomials</td>
<td>4.1 Univariate Skew Polynomials</td>
<td>467</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.2 Constructor for skew polynomial rings</td>
<td>495</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.3 Skew Univariate Polynomial Rings</td>
<td>497</td>
</tr>
<tr>
<td>5</td>
<td>Rational Functions</td>
<td>5.1 Fraction Field of Integral Domains</td>
<td>505</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.2 Fraction Field Elements</td>
<td>510</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.3 Univariate rational functions over prime fields</td>
<td>515</td>
</tr>
<tr>
<td>6</td>
<td>Laurent Polynomials</td>
<td>6.1 Ring of Laurent Polynomials</td>
<td>523</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.2 Elements of Laurent polynomial rings</td>
<td>530</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.3 MacMahon’s Partition Analysis Omega Operator</td>
<td>547</td>
</tr>
<tr>
<td>7</td>
<td>Infinite Polynomial Rings</td>
<td>7.1 Infinite Polynomial Rings</td>
<td>553</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.2 Elements of Infinite Polynomial Rings</td>
<td>562</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.3 Symmetric Ideals of Infinite Polynomial Rings</td>
<td>571</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.4 Symmetric Reduction of Infinite Polynomials</td>
<td>580</td>
</tr>
<tr>
<td>8</td>
<td>Boolean Polynomials</td>
<td>8.1 Boolean Polynomials</td>
<td>587</td>
</tr>
<tr>
<td>9</td>
<td>Noncommutative Polynomials</td>
<td>9.1 Noncommutative Polynomials via libSINGULAR/Plural</td>
<td>645</td>
</tr>
<tr>
<td>10</td>
<td>Indices and Tables</td>
<td></td>
<td>665</td>
</tr>
</tbody>
</table>
1.1 Constructors for polynomial rings

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

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

Construct a boolean polynomial ring with the following parameters:

**INPUT:**

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

**EXAMPLES:**

```python
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
```

```python
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)
```

```python
sage: R = BooleanPolynomialRing(3,'x',order='degneglex')
sage: R.term_order()
```

(continues on next page)
sage: BooleanPolynomialRing(names=('x','y'))
Boolean PolynomialRing in x, y

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

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

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

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

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

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

INPUT:

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

**Note:** If the given implementation does not exist for rings with the given number of generators and the given sparsity, then an error results.
PolynomialRing(base_ring, name, sparse=False) returns a univariate polynomial ring; also, PolynomialRing(base_ring, names, sparse=False) yields a univariate polynomial ring, if names is a list or tuple providing exactly one name. All other input formats return a multivariate polynomial ring.

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

**EXAMPLES:**

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

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

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

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

You must specify a name:

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

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

The square bracket notation:

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

However, rings with different variables are different:

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

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

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

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

The generic implementation uses neither NTL nor FLINT:

```sage
sage: Zx = PolynomialRing(ZZ, 'x', implementation='generic'); Zx
Univariate Polynomial Ring in x over Integer Ring

sage: Zx.element_class
<... 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
```

2. PolynomialRing(base_ring, names, ...)
There is a unique polynomial ring with each term order:

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

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

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

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

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

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

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

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

Note that a multivariate polynomial ring is returned when an explicit number is given.

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

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

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

1.1. Constructors for polynomial rings
By calling the \texttt{inject_variables()} method, all those variable names are available for interactive use:

\begin{verbatim}
\texttt{sage: R.inject_variables()}
Defining x2, x3, x5, x7, x11, x13, x17, x19, x23, x29, x31, x37, x41, x43, x47, ...
→ x53, x59, x61, x67, x71, x73, x79, x83, x89, x97
\texttt{sage: (x2 + x41 + x71)^2}
x2^2 + 2*x2*x41 + x41^2 + 2*x2*x71 + 2*x41*x71 + x71^2
\end{verbatim}

4. \texttt{PolynomialRing(base\_ring, n, \ldots, var\_array=var\_array, \ldots)}

This creates an array of variables where each variables begins with an entry in \texttt{var\_array} and is indexed from 0 to \( n - 1 \).

\begin{verbatim}
\texttt{sage: PolynomialRing(ZZ, 3, var\_array=['x','y'])}
Multivariate Polynomial Ring in x0, y0, x1, y1, x2, y2 over Integer Ring
\texttt{sage: PolynomialRing(ZZ, 3, var\_array='a,b')}
Multivariate Polynomial Ring in a0, b0, a1, b1, a2, b2 over Integer Ring
\end{verbatim}

It is possible to create higher-dimensional arrays:

\begin{verbatim}
\texttt{sage: PolynomialRing(ZZ, 2, 3, var\_array=('p', 'q'))}
Multivariate Polynomial Ring in p00, q00, p01, q01, p02, q02, p10, q10, p11, q11, ...
→ p12, q12 over Integer Ring
\texttt{sage: PolynomialRing(ZZ, 2, 3, 4, var\_array='m')}
Multivariate Polynomial Ring in m00, m01, m02, m03, m10, m11, m12, m13, ...
→ m20, m21, m22, m23, m00, m01, m02, m03, m10, m11, m12, m13, ...
→ m21, m22, m23 over Integer Ring
\end{verbatim}

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

\begin{verbatim}
\texttt{sage: PolynomialRing(ZZ, 2, var\_array='m')}
Multivariate Polynomial Ring in m00, m01, m10, m11 over Integer Ring
\end{verbatim}

\textbf{Square brackets notation}

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

\begin{verbatim}
\texttt{sage: RR["x"]}
Univariate Polynomial Ring in x over Real Field with 53 bits of precision
\texttt{sage: RR["x,y"]}
Multivariate Polynomial Ring in x, y over Real Field with 53 bits of precision
\texttt{sage: P.<x,y> = RR[]; P}
Multivariate Polynomial Ring in x, y over Real Field with 53 bits of precision
\end{verbatim}

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

\textbf{Changing variable names}

Consider

\begin{verbatim}
\texttt{sage: R.<x,y> = PolynomialRing(QQ,2); R}
Multivariate Polynomial Ring in x, y over Rational Field
\texttt{sage: f = x^2 - 2*y^2}
\end{verbatim}

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

```python
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:

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

sage.rings.polynomial.polynomial_ring_constructor.polynomial_default_category(base_ring_category, n_variables)

Choose an appropriate category for a polynomial ring.

It is assumed that the corresponding base ring is nonzero.

INPUT:

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

EXAMPLES:

```python
sage: from sage.rings.polynomial.polynomial_ring_constructor import polynomial_default_category
sage: polynomial_default_category(Rings(),1) is Algebras(Rings()).Infinite()
True
sage: polynomial_default_category(Rings().Commutative(),1) is Algebras(Rings()).Commutative().Infinite()
True
sage: polynomial_default_category(Fields(),1) is EuclideanDomains() & CommutativeAlgebras(Fields()).Infinite()
True
sage: polynomial_default_category(Fields(),2) is UniqueFactorizationDomains() & CommutativeAlgebras(Fields()).Infinite()
True
sage: QQ['t'].category() is EuclideanDomains() & CommutativeAlgebras(QQ.category()).Infinite()
True
sage: QQ['s','t'].category() is UniqueFactorizationDomains() & CommutativeAlgebras(QQ.category()).Infinite()
True
sage: QQ['s'][t].category() is UniqueFactorizationDomains() & CommutativeAlgebras(QQ['s'].category()).Infinite()
True
```

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

Custom unpickling function for polynomial rings.

1.1. Constructors for polynomial rings
This has the same positional arguments as the old `PolynomialRing` constructor before trac ticket #23338.
2.1 Univariate Polynomials and Polynomial Rings

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

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

2.1.1 Univariate Polynomial Rings

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

AUTHOR:

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

EXAMPLES:

```sage
z = QQ['z'].0
(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
k = PolynomialRing(QQ,'x','y'); loads(dumps(k)) == k
True
k = PolynomialRing(ZZ,'y'); loads(dumps(k)) == k
True
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):
We create a polynomial ring over a quaternion algebra:

```python
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:

```python
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:

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

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

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

A class for polynomial ring over complete discrete valuation fields

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

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

A class for polynomial ring over complete discrete valuation rings

```python
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`

Univariate polynomial ring over a commutative ring.

```python
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 \( \text{self} \), or a principal ideal of \( \text{self} \).

EXAMPLES:

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

The same example, using the polynomial instead of the ideal, and customizing the variable name:
**w**eyl_algebra()

Return the Weyl algebra generated from self.

EXAMPLES:

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

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

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

Univariate polynomial ring over a finite field.

EXAMPLES:

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

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

Construct a monic irreducible polynomial of degree *n*.

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

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

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

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_commutative

modulus()

EXAMPLES:

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

residue_field(ideal, names=None)

Return the residue finite field at the given ideal.

EXAMPLES:

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

Non-maximal ideals are not accepted:
sage: R.residue_field(t^2 + 1)
Traceback (most recent call last):
...  
ArithmeticError: ideal is not maximal
sage: R.residue_field(0)
Traceback (most recent call last):
...  
ArithmeticError: ideal is not maximal
sage: R.residue_field(1)
Traceback (most recent call last):
...  
ArithmeticError: ideal is not maximal

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


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.
A monic irreducible polynomial of degree $n$ in $\text{self}$.

EXAMPLES:

```python
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:

```python
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:

```python
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_generic

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_cdvf
A class for dense polynomial ring over padic fields

class sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_capped_absolute

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_generic

class sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_capped_relative

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_generic

class sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_fixed_mod

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_generic

class sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_generic

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_cdvr
A class for dense polynomial ring over padic rings
class sage.rings.polynomial.polynomial_ring.PolynomialRing_field(base_ring, name='x', sparse=False, element_class=None, category=None)

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_integral_domain, sage.rings.ring.PrincipalIdealDomain
divided_difference(points, full_table=False)

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

INPUT:

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

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

OUTPUT:

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

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

EXAMPLES:

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

```python
sage: points = [(1.0, 0.7651977), (1.3, 0.6200860), (1.6, 0.4554022), (1.9, 0.2818186), (2.2, 0.1103623)]
sage: R = PolynomialRing(RR, "x")
sage: R.divided_difference(points)
[0.765197700000000, -0.483705666666667, -0.108733888888889, 0.0658783950617283, 0.00182510288066044]
```

Now return the full divided-difference table:

```python
sage: points = [(1.0, 0.7651977), (1.3, 0.6200860), (1.6, 0.4554022), (1.9, 0.2818186), (2.2, 0.1103623)]
sage: R = PolynomialRing(RR, "x")
sage: R.divided_difference(points, full_table=True)
[[0.765197700000000, -0.483705666666667, -0.108733888888889, 0.0658783950617283, 0.00182510288066044],
 [0.620086000000000, -0.483705666666667, -0.548946000000000, -0.108733888888889, 0.0658783950617283],
 [0.455402200000000, -0.578612000000000, -0.0494433333333339, 0.00182510288066044],
 [0.281818600000000, -0.578612000000000, -0.0494433333333339, 0.00182510288066044],
 [0.110362300000000, -0.0494433333333339, 0.00182510288066044, 0.00182510288066044]]
```

(continues on next page)
The following example is taken from Example 4.12, page 225 of [MF1999]:

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

divided_difference(points, algorithm='divided_difference', previous_row=None)

Return the Lagrange interpolation polynomial through the given points.

INPUT:

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

- `algorithm` - (default: 'divided_difference'): one of the following:
  - 'divided_difference': use the method of divided differences.
  - `algorithm='neville'`: adapt Neville’s method as described on page 144 of [BF2005] to recursively generate the Lagrange interpolation polynomial. Neville’s method generates a table of approximating polynomials, where the last row of that table contains the \(n\)-th Lagrange interpolation polynomial. The adaptation implemented by this method is to only generate the last row of this table, instead of the full table itself. Generating the full table can be memory inefficient.

- `previous_row` - (default: None): This option is only relevant if used with `algorithm='neville'`. If provided, this should be the last row of the table resulting from a previous use of Neville’s method. If such a row is passed, then `points` should consist of both previous and new interpolating points. Neville’s method will then use that last row and the interpolating points to generate a new row containing an interpolation polynomial for the new points.

OUTPUT:

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

EXAMPLES:
By default, we use the method of divided differences:

```python
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')
```

```python
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:

```python
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')
```

```python
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:

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

```python
sage: a = R.base_ring().gen()
sage: p = R.lagrange_polynomial([(a^2+a,a), (a,1)], algorithm="neville")
sage: R.lagrange_polynomial([(a^2+a,a), (a,1), (a^2,a^2+a+1)], algorithm="neville", previous_row=p)[-1]
a^2*x^2 + a^2*x + a^2
```
class sage.rings.polynomial.polynomial_ring.PolynomialRing_general(
    base_ring,
    name=None,
    sparse=False,
    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
```

(continues on next page)

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
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^15 + x^16 + x^17 + x^18 + x^19 + O(x^20)
```

**construction** *

**cyclotomic_polynomial** *(n)*

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

**EXAMPLES:**

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

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

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

**EXAMPLES:**

```sage
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
```

2.1. Univariate Polynomials and Polynomial Rings
flattening_morphism()
Return the flattening morphism of this polynomial ring

EXAMPLES:

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

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

EXAMPLES:

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

is_exact()

EXAMPLES:

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

Return False, since polynomial rings are never fields.

EXAMPLES:

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

**is_integral_domain** (*proof=True*)

EXAMPLES:

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

**is_noetherian**

**is_sparse**

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

EXAMPLES:

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

EXAMPLES:

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

**krull_dimension**

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

EXAMPLES:
sage: R.<x> = QQ[]
sage: R.krull_dimension()
1
sage: 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:
monomial (exponent)
Return the monomial with the exponent.

INPUT:
• exponent – nonnegative integer

EXAMPLES:

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

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

EXAMPLES:

```
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
```

(continues on next page)
AUTHORS:

• Joel B. Mohler

random_element \( \text{degree=(-1, 2), *args, **kwds} \)

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:

\[
\begin{align*}
sage: & R.<x> = ZZ[] \\
sage: & R.random_element(10, 5,10) \\
& 5*x^10 + 5*x^9 + 9*x^8 + 8*x^7 + 6*x^6 + 8*x^5 + 8*x^4 + 9*x^3 + 8*x^2 + 8*x - 16
\\
sage: & R.random_element(6) \\
& x^6 - 2*x^5 - 2*x^3 + 2*x^2 - 3*x
\\
sage: & R.random_element(6) \\
& -x^6 + x^5 + x^2 - x
\\
sage: & R.random_element(6) \\
& -5*x^6 + x^5 + 14*x^4 - x^3 + x^2 - x + 4
\end{align*}
\]

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

\[
\begin{align*}
\text{sage: } & R.random_element(\text{degree=(0,8)}) \\
& 4*x^4 + 2*x^3 - x + 4 \\
\text{sage: } & R.random_element(\text{degree=(0,8)}) \\
& x + 1
\end{align*}
\]

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 range(100))
True

**set_karatsuba_threshold** (*Karatsuba_threshold*)
Changes the default threshold for this ring in the method _mul_karatsuba to fall back to the schoolbook algorithm.

**Warning:** This method may have a negative performance impact in polynomial arithmetic. So use it at your own risk.

**EXAMPLES:**

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

**variable_names_recursive**(depth=+Infinity)
Return the list of variable names of this ring and its base rings, as if it were a single multi-variate polynomial.

**INPUT:**
- depth – an integer or **Infinity**.

**OUTPUT:**
A tuple of strings.

**EXAMPLES:**

```sage
sage: R = QQ['x']['y']['z']
sage: R.variable_names_recursive()
('x', 'y', 'z')
sage: R.variable_names_recursive(2)
('y', 'z')
```
class sage.rings.polynomial.polynomial_ring.PolynomialRing_integral_domain(base_ring, name='x', sparse=False, implementation=None, element_class=None, category=None)

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

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

sage.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")

Chapter 2. Univariate Polynomials
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', *args)
```
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)
```
The number of generators can be passed as a third argument:
```
sage: polygens(QQ, 'x', 4)
(x0, x1, x2, x3)
```

### 2.1.2 Ring homomorphisms from a polynomial ring to another ring

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

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

**AUTHORS:**
- Peter Bruin (March 2014): initial version

```
class sage.rings.polynomial.polynomial_ring_homomorphism.PolynomialRingHomomorphism_from_base
```
Bases:
sage.rings.morphism.RingHomomorphism_from_base

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

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

**is_injective()**

Return whether this morphism is injective.

**EXAMPLES:**

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

**is_surjective()**

Return whether this morphism is surjective.

**EXAMPLES:**

```sage
sage: R.<x> = ZZ[
               ]
sage: S.<x> = Zmod(2)[
               ]
sage: R.hom(S).is_surjective()
True
```

### 2.1.3 Univariate Polynomial Base Class

**AUTHORS:**

- William Stein: first version.
- Martin Albrecht: Added singular coercion.
- Robert Bradshaw: Move Polynomial_generic_dense to Cython.
- Miguel Marco: Implemented resultant in the case where PARI fails.
- Simon King: Use a faster way of conversion from the base ring.
- Julian Rueth (2012-05-25,2014-05-09): Fixed is_squarefree() for imperfect fields, fixed division without remainder over QQbar; added _cache_key for polynomials with unhashable coefficients
- Edgar Costa (2017-07): Added rational reconstruction.
- Kiran Kedlaya (2017-09): Added reciprocal transform, trace polynomial.
- David Zureick-Brown (2017-09): Added is_weil_polynomial.

class sage.rings.polynomial.polynomial_element.ConstantPolynomialSection

Bases: sage.categories.map.Map

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

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

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

### class sage.rings.polynomial.polynomial_element.Polynomial

Bases: sage.structure.element.CommutativeAlgebraElement

A polynomial.

EXAMPLES:

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

**_add_(right)**

Add two polynomials.

EXAMPLES:

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

**_sub_(other)**

Default implementation of subtraction using addition and negation.

**_lmul_(left)**

Multiply self on the left by a scalar.

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

**_rmul_(right)**

Multiply self on the right by a scalar.

**EXAMPLES:**

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

**_mul_(right)**

**EXAMPLES:**

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

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

**_mul_trunc_(right, n)**

Return the truncated multiplication of two polynomials up to n.

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

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

**EXAMPLES:**

```python
sage: R = QQ['x']['y']
sage: y = R.gen()
sage: x = R.base_ring().gen()
sage: p1 = 1 - x*y + 2*y**3
```

(continues on next page)
Todo: implement a generic truncated Karatsuba and use it here.

adams_operator \((n, \text{monic}=False)\)

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

INPUT:

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

EXAMPLES:

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

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

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

\text{add\_bigoh} \((\text{prec})\)

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

EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: f = x^2 - 2*x + 2
sage: f.add_bigoh(10)
x^2 + 1024
```

(continues on next page)
all_roots_in_interval \((a=None, b=None)\)

Return True if the roots of this polynomial are all real and contained in the given interval.

EXAMPLES:

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

any_root \((ring=None, degree=None, assume_squarefree=False)\)

Return a root of this polynomial in the given ring.

INPUT:

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

EXAMPLES:

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

```python
sage: f = x^6 + 10*x^4 + 6*x^3 + 5*x^2 + 2*x + 2
sage: f.any_root(GF(11^6, 'a'))
a^5 + a^4 + 7*a^3 + 2*a^2 + 10*a
sage: sorted(f.roots(GF(11^6, 'a')))  # doctest: +NORMALIZE_WHITESPACE
[(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), (2*a^5 + 8*a^4 + 3*a^3 + 6*a + 2, 1),
 (a^5 + 3*a^4 + 8*a^3 + 3*a^2 + 4*a + 1, 1),
 (10*a^5 + 3*a^4 + 8*a^3 + a^2 + 10*a + 4, 1)]
```

```python
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)
```

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

**args()**

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

**EXAMPLES:**

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

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

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

**base_extend(R)**

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

**EXAMPLES:**

```python
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:**

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

**change_ring(R)**

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

**INPUT:**

- R - a ring or morphism.

**EXAMPLES:**

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

sage: f = x^2 + z
sage: f.change_ring(K.embeddings(CC)[0])
x^2 - 0.500000000000000 - 0.866025403784439*I

sage: R.<x> = QQ[

sage: f = x^2 + 1
sage: f.change_ring(QQ.embeddings(CC)[0])
x^2 + 1.00000000000000
```

**change_variable_name** *(var)*

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

**EXAMPLES:**

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

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

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

**EXAMPLES:**

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

**complex_roots**

Return the complex roots of this polynomial, without multiplicities.

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

**EXAMPLES:**

```python
sage: x = polygen(ZZ)
sage: (x^3 - 1).complex_roots()  # note: low order bits slightly different on ppc.
[1.00000000000000, -0.500000000000000 - 0.866025403784439*I, -0.500000000000000 + 0.866025403784439*I]
```

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

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

**INPUT:**

- *k* – a non-negative integer
- `algorithm=None` (default), "resultant" or "BFSS". See `composed_op()`
- `monic=False` (default) or `True`. See `composed_op()`
The polynomial of degree $d^k$ where $d$ is the degree, whose roots are all $k$-fold products of roots of this polynomial. That is, $f * f \cdots * f$ where this is $f$ and $f * f = f\text{.composed\_op}(f,\text{operator.mul})$.

EXAMPLES:

```python
sage: R.<a,b,c> = ZZ[]
sage: x = polygen(R)
sage: f = (x-a)*(x-b)*(x-c)
sage: f.compose_power(2).factor()
(x - c^2) * (x - b^2) * (x - a^2) * (x - b*c)^2 * (x - a*c)^2 * (x - a*b)^2
sage: x = polygen(QQ)
sage: f = x^2-2*x+2
sage: f2 = f.compose_power(2); f2
x^4 - 4*x^3 + 8*x^2 - 16*x + 16
sage: f2 == f.composed_op(f,operator.mul)
True
sage: f3 = f.compose_power(3); f3
x^8 - 8*x^7 + 32*x^6 - 64*x^5 + 128*x^4 - 512*x^3 + 2048*x^2 - 4096*x + 4096
sage: f3 == f2.composed_op(f,operator.mul)
True
```

**compose_trunc** (other, $n$)

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

This method currently works for some specific coefficient rings only.

EXAMPLES:

```python
sage: Pol.<x> = CBF[]
sage: (1 + x + x^2/2 + x^3/6 + x^4/24 + x^5/120).compose_trunc(1 + x, 2)

(2.708333333333333 +/- ...e-16)*x + (2.71666666666667 +/- ...e-15)
```

```
sage: Pol.<x> = QQ['y']['x']
```

```
Traceback (most recent call last):
...
NotImplementedError: truncated composition is not implemented for this __subclass of polynomials
```

**composed\_op** ($p1$, $p2$, op=None, monic=False)

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

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

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

where the roots $a$ and $b$ are to be considered in the algebraic closure of the fraction field of the coefficients and counted with multiplicities. If the polynomials are not monic this quantity is multiplied by

alpha^{deg(p_2)}_1

\quad alpha^{deg(p_1)}_2

where

alpha_1 and

alpha_2 are the leading coefficients of $p_1$ and $p_2$ respectively.
INPUT:

• \( p_2 \) – univariate polynomial belonging to the same polynomial ring as this polynomial
• \( \text{op} \) – \text{operator.OP} where \text{OP}=\text{add or sub or mul or truediv}.
• \( \text{algorithm} \) – can be “resultant” or “BFSS”; by default the former is used when the polynomials
  have few nonzero coefficients and small degrees or if the base ring is not \( \mathbb{Z} \) or \( \mathbb{Q} \). Otherwise the latter
  is used.
• \( \text{monic} \) – whether to return a monic polynomial. If \text{True} the coefficients of the result belong to the
  fraction field of the coefficients.

ALGORITHM:

The computation is straightforward using resultants. Indeed for the composed sum it would be
\[ R_y(p_1(x - y), p_2(y)) \]. However, the method from [BFSS2006] using series expansions is asymptotically
much faster.

Note that the algorithm BFSS with polynomials with coefficients in \( \mathbb{Z} \) needs to perform operations over
\( \mathbb{Q} \).

Todo:

• The [BFSS2006] algorithm has been implemented here only in the case of polynomials over rationals. For other rings of zero characteristic (or if the characteristic is larger than the product of the
degrees), one needs to implement a generic method \_exp_series. In the general case of non-zero
characteristic there is an alternative algorithm in the same paper.
• The Newton series computation can be done much more efficiently! See [BFSS2006].

EXAMPLES:

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

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

```
sage: x = polygen(QQbar)
sage: p1 = x**2 - AA(2).sqrt()
sage: p2 = x**3 - AA(3).sqrt()
sage: r1 = p1.roots(multiplicities=False)
sage: r2 = p2.roots(multiplicities=False)
sage: p = p1.composed_op(p2, operator.add)
sage: p
x^6 - 4.242640687119285?*x^4 - 3.464101615137755?*x^3 + 6*x^2 - 14.
→6963845669907?*x + 0.1715728752538099?
sage: all(p(x+y).is_zero() for x in r1 for y in r2)
True
```
sage: p1 = x**2 + x - 1
sage: p2 = x**3 + x - 1
sage: p_add = p1.composed_op(p2, operator.add)
sage: p_add
x^6 + x^5 + x^3 + x^2 + 1
sage: p_mul = p1.composed_op(p2, operator.mul)
sage: p_mul
x^6 + x^4 + x^2 + x + 1
sage: p_div = p1.composed_op(p2, operator.truediv)
sage: p_div
x^6 + x^5 + x^4 + x^2 + 1

sage: K = GF(2**6, 'a')
sage: r1 = p1.roots(K, multiplicities=False)
sage: r2 = p2.roots(K, multiplicities=False)
sage: all(p_add(x1+x2).is_zero() for x1 in r1 for x2 in r2)
True
sage: all(p_mul(x1*x2).is_zero() for x1 in r1 for x2 in r2)
True
sage: all(p_div(x1/x2).is_zero() for x1 in r1 for x2 in r2)
True

constant_coefficient()

Return the constant coefficient of this polynomial.

OUTPUT: element of base ring

EXAMPLES:

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

content_ideal()

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

EXAMPLES:

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

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

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

cyclotomic_part()

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

The algorithm assumes that the polynomial has rational coefficients.

See also:

- `is_cyclotomic()`
- `is_cyclotomic_product()`
- `has_cyclotomic_factor()`
EXAMPLES:

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

degree (gen=None)

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

EXAMPLES:

```python
code
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 trac ticket #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
differentiate(*args)

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

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

See also:

_derivative()

EXAMPLES:

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

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

discriminant()

Return the discriminant of self.

The discriminant is

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

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.

ALGORITHM:

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

EXAMPLES:

In the case of elliptic curves in special form, the discriminant is easy to calculate:
```python
sage: R.<x> = QQ[]
sage: f = x^3 + x + 1
sage: d = f.discriminant(); d
-31
sage: d.parent() is QQ
True
sage: EllipticCurve([1, 1]).discriminant()/16
-31
```

We can compute discriminants over univariate and multivariate polynomial rings:

```python
sage: R.<a> = QQ[]
sage: S.<x> = R[]
sage: f = a*x + x + a + 1
sage: d = f.discriminant(); d
1
sage: d.parent() is R
True
sage: R.<a, b> = QQ[]
sage: S.<x> = R[]
sage: f = x^2 + a + b
sage: d = f.discriminant(); d
-4*a - 4*b
sage: d.parent() is R
True
```

**dispersion** *(other=None)*

Compute the dispersion of a pair of polynomials.

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

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

See also:

*dispersion_set()*

**EXAMPLES:**

```python
sage: Pol.<x> = QQ[]
sage: x.dispersion(x + 1)
1
sage: (x + 1).dispersion(x)
-Infinity
sage: Pol.<x> = QQbar[]
sage: pol = Pol([sqrt(5), 1, 3/2])
sage: pol.dispersion()
0
sage: (pol*pol(x+3)).dispersion()
3
```
**dispersion_set** *(other=None)*

Compute the dispersion set of two polynomials.

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

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

**Algorithm:**

See Section 4 of Man & Wright [MW1994].

**See also:**

*dispersion()*

**Examples:**

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

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

**divides** *(p)*

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

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

**Examples:**

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

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

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

**EXAMPLES:**

```python
degree = R.<x> = QQ[]
degree = x.euclidean_degree()
degree
1
```

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

**exponents()**

Return the exponents of the monomials appearing in `self`.

**EXAMPLES:**

```python
exponents = _.<x> = PolynomialRing(ZZ)
exponents = f.exponents()
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`:

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

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

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

Over `Z` the irreducible factors need not be monic:
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 + (3*a + 3)*x^3 + 2*a*x^2 + (3*a + 1)*x + 3*a + 1)
```

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

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

A new ring. In the example below, we set the special method

```python
_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()  # 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 - I) * (x + I)
```

(continues on next page)
(1.00000000000000000000000000000000*I) * (x - I) * (x + I)
\[\text{sage: expand(F)}\]
I*x^2 + I

Over a number field:

\[\text{sage: K.<z> = CyclotomicField(15)}\]
\[\text{sage: x = polygen(K)}\]
\[\text{sage: ((x^3 + z*x + 1)^3*(x - z)).factor()}\]
\[\text{(x - z) * (x^3 + z*x + 1)^3}\]
\[\text{sage: cyclotomic_polynomial(12).change_ring(K).factor()}\]
\[\text{(x^2 - z^5 - 1) * (x^2 + z^5)}\]
\[\text{sage: ((x^3 + z*x + 1)^3*(x/(z+2) - 1/3)).factor()}\]
\[\text{(-1/331*z^7 + 3/331*z^6 - 6/331*z^5 + 11/331*z^4 - 21/331*z^3 + 41/331*z^2 -} \]
\[\rightarrow -82/331*z + 165/331) * (x - 1/3*z - 2/3) * (x^3 + z*x + 1)^3\]

Over a relative number field:

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

Over the real double field:

\[\text{sage: R.<x> = RDF[]}\]
\[\text{sage: (-2*x^2 - 1).factor()}\]
\[(-2.0) * (x^2 + 0.5000000000000001)\]
\[\text{sage: (-2*x^2 - 1).factor().expand()}\]
\[-2.0*x^2 - 1.0000000000000002\]
\[\text{sage: f = (x - 1)^3}\]
\[\text{sage: f.factor()} # abs tol 2e-5\]
\[(x - 1.0000065719436413) * (x^2 - 1.999934280563585*x + 0.9999934280995487)\]

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

\[\text{sage: f.roots()} # abs tol 2e-5\]
\[
[(1.0000065719436413, 1)]
\]

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

\[\text{sage: f = (x - 1)^3}\]
\[\text{sage: f.factor()} # abs tol 2e-5\]
\[(x - 1.0000138879287663 + 1.0000013435286879*I) * (x - 0.9999942196864997 + 0.9999930308395994*I) * (x - 0.9999918923847313 + 1.00001135354909125*I) * (x - 0.9999908759550227 - 1.0000069659624138*I) * (x + 0.9999985293187753 - 0.99999886153831807*I) * (x + 1.0000105947233 - 1.0000044186544053*I)\]
\[\text{sage: [f(t[0][0]).abs() for t in F]} # abs tol 1e-13\]
\[[1.979365054e-14, 1.97936298566e-14, 1.97936990747e-14, 3.6812407475e-14, 3.65211563729e-14, 3.65220890052e-14]\]
Factoring polynomials over $\mathbb{Z}/n\mathbb{Z}$ for composite $n$ is not implemented:

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

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

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

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

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

**gcd**

Return a greatest common divisor of this polynomial and other.

**INPUT:**

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

**OUTPUT:**

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

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

**EXAMPLES:**

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

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

Use multivariate implementation for polynomials over polynomials rings:

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

gradient()

Return a list of the partial derivative of self with respect to the variable of this univariate polynomial. There is only one partial derivative.

EXAMPLES:

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

hamming_weight()

Return the number of non-zero coefficients of self. Also called weight, Hamming weight or sparsity.

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

The method `hamming_weight()` is an alias:

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

**has_cyclotomic_factor()**

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

The algorithm assumes that the polynomial has rational coefficients.

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

See also:

`is_cyclotomic() is_cyclotomic_product() cyclotomic_part()`

**EXAMPLES:**

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

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

Return the homogenization of this polynomial.

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

**INPUT:**

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

**OUTPUT:**

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

**EXAMPLES:**
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 characteristic, the degree can drop in this case:

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

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

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

```
integral (var=None)

Return the integral of this polynomial.

By default, the integration variable is the variable of the polynomial.

Otherwise, the integration variable is the optional parameter `var`.

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

**EXAMPLES:**
```
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
```

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

```
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.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 Integer Ring
```

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.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 Integer Ring
```

```
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).

**See also:**

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

**EXAMPLES:**

```
sage: S.<t> = QQ[]
sage: f = inverse_mod(t^2 + 1, t^3 + 1); f
-1/2*t^2 - 1/2*t + 1/2
sage: f * (t^2 + 1) % (t^3 + 1)
1
sage: f = t.inverse_mod((t+1)^7); f
-t^6 - 7*t^5 - 21*t^4 - 35*t^3 - 35*t^2 - 21*t - 7
sage: (f * t) + (t+1)^7
1
sage: t.inverse_mod(S.ideal((t + 1)^7)) == f
True
```

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

```
sage: 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)
# 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
```

**ALGORITHM:** Solve the system as \(as + mt = 1\), 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).
inverse_of_unit() Returns the multiplicative inverse of this polynomial.

\[ f^{-1} \]

If the polynomial is a unit, \( f^{-1} \) is returned, otherwise a `ValueError` is raised.

**EXAMPLES:**

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

inverse_series_trunc(prec) Returns a polynomial approximation of precision prec of the inverse series of this polynomial.

**EXAMPLES:**

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

Note that the constant coefficient needs to be a unit:

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

The method works over any polynomial ring:

```python
sage: R = Zmod(4)
sage: Rx.<x> = R[]
sage: Rxy.<y> = Rx[]
sage: p = 1 + (1+2*x)*y + x**2*y**4
sage: q = p.inverse_series_trunc(10)
sage: (p*q).truncate(11)
(2*x^4 + 3*x^2 + 3)*y^10 + 1
```

Even noncommutative ones:
AUTHORS:

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

**is_constant()**

Return True if this is a constant polynomial.

**OUTPUT:**

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

**EXAMPLES:**

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

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

Test if this polynomial is a cyclotomic polynomial.

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

By default the answer is a boolean. But if certificate is True, the result is a non-negative integer: it is 0 if self is not cyclotomic, and a positive integer n if self is the n-th cyclotomic polynomial.

See also:

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

**INPUT:**

- certificate - boolean, default to False. Only works with algorithm set to “pari”.
- algorithm - either “pari” or “sage” (default is “pari”)

**ALGORITHM:**

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

**EXAMPLES:**

Quick tests:
The `is_cyclotomic()` method checks if a polynomial is cyclotomic. For example:

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

When the `certificate=True` option is used, it returns the cyclotomic number associated with the polynomial. For instance:

```python
sage: (x^2 + 1).is_cyclotomic(certificate=True)
3
sage: (x^2 + 2*x + 1).is_cyclotomic(certificate=True)
0
```

To test the first 100 cyclotomic polynomials, you can:

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

Some more tests include:

```python
sage: (x^16 + x^14 - x^10 + x^8 - x^6 + x^2 + 1).is_cyclotomic(algorithm="pari")
False
sage: (x^16 + x^14 - x^10 + x^8 - x^6 + x^2 + 1).is_cyclotomic(algorithm="sage")
False
sage: (x^16 + x^14 - x^10 - x^8 - x^6 + x^2 + 1).is_cyclotomic(algorithm="pari")
True
sage: (x^16 + x^14 - x^10 - x^8 - x^6 + x^2 + 1).is_cyclotomic(algorithm="sage")
True
```

When invalid arguments are provided:

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

Test using other rings:

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

The `is_cyclotomic_product()` method tests whether a polynomial is a product of cyclotomic polynomials.

```python
sage: is_cyclotomic_product()
```

This method simply calls the function `polishcycloprod` from the Pari library.

2.1. Univariate Polynomials and Polynomial Rings
See also:

is_cyclotomic() cyclotomic_part() has_cyclotomic_factor()

EXAMPLES:

```python
sage: x = polygen(ZZ)
sage: (x^5 - 1).is_cyclotomic_product()
True
sage: (x^5 + x^4 - x^2 + 1).is_cyclotomic_product()
False
sage: p = prod(cyclotomic_polynomial(i) for i in [2,5,7,12])
sage: p.is_cyclotomic_product()
True
sage: (x^5 - 1/3).is_cyclotomic_product()
False
sage: x = polygen(Zmod(5))
sage: (x-1).is_cyclotomic_product()
Traceback (most recent call last):
... NotImplementedError: not implemented in non-zero characteristic
```

**is_gen()**

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

EXAMPLES:

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

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

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

**is_homogeneous()**

Return True if this polynomial is homogeneous.

EXAMPLES:

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

Return whether this polynomial is irreducible.

EXAMPLES:

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

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

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

is_monic()

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

EXAMPLES:

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

AUTHORS:

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

is_monomial()

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

EXAMPLES:

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

(continues on next page)
The coefficient must be 1:

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

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

```python
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:**

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

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

### `is_one`()

Test whether this polynomial is 1.

**EXAMPLES:**

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

### `is_primitive`(n=None, n_prime_divs=None)

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

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

- (ring theory) A polynomial over a ring is primitive if its coefficients generate the unit ideal.

Calling \( \text{is}_\text{primitive} \) on a polynomial over an infinite field will raise an error.

The additional inputs to this function are to speed up computation for field semantics (see note).

INPUT:

- \( n \) (default: None) - if provided, should equal \( q - 1 \) where \( \text{self.parent()} \) is the field with \( q \) elements; otherwise it will be computed.

- \( \text{n}_\text{prime}\_\text{divs} \) (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. \( \text{pdivs}=\text{n}_\text{prime}\_\text{divisors()} \)) is a good idea for repeated calls to \( \text{is}_\text{primitive} \) 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
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
    #f.is_primitive() #BUG:: elsewhere in Sage, should return True
    sage: f=4*x^2+2
    #f.is_primitive() #BUG:: elsewhere in Sage, should return False

**is_real_rooted()**

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

EXAMPLES:

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

**is_square** *(root=False)*

Return whether or not polynomial is square.

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

INPUT:

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

OUTPUT:

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

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

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

```sage
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()
True
```

(continues on next page)
In 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
```

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

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

`is_term()`
Return True if this polynomial is a nonzero element of the base ring times a power of the variable.

EXAMPLES:

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

### `is_weil_polynomial(return_q=False)`

Return True if this is a Weil polynomial.

This polynomial must have rational or integer coefficients.

**INPUT:**

* self – polynomial with rational or integer coefficients

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

**EXAMPLES:**

```
sage: polRing.<x> = PolynomialRing(Rationals())
sage: P0 = x^4 + 5*x^3 + 15*x^2 + 25*x + 25
sage: P1 = x^4 + 25*x^3 + 15*x^2 + 5*x + 25
sage: P2 = x^4 + 5*x^3 + 25*x^2 + 25*x + 25
sage: P0.is_weil_polynomial(return_q=True)
(True, 5)
sage: P0.is_weil_polynomial(return_q=False)
True
sage: P1.is_weil_polynomial(return_q=True)
(False, 0)
sage: P1.is_weil_polynomial(return_q=False)
```

(continues on next page)
AUTHORS:
David Zureick-Brown (2017-10-01)

is_zero()
Test whether this polynomial is zero.

EXAMPLES:

```python
sage: R = GF(2)['x']['y']
sage: R([0,1]).is_zero()
False
sage: R([0]).is_zero()
True
sage: R([-1]).is_zero()
False
```

lc()
Return the leading coefficient of this polynomial.

OUTPUT: element of the base ring This method is same as leading_coefficient().

EXAMPLES:

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

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:

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

list (copy=True)
Return a new copy of the list of the underlying elements of self.

EXAMPLES:

```python
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:
Here is an example with a generic polynomial ring:

```sage
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]
```

**lm()**

Return the leading monomial of this polynomial.

**EXAMPLES:**

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

**lt()**

Return the leading term of this polynomial.

**EXAMPLES:**

```sage
sage: R.<x> = QQ[]
sage: f = (-2/5)*x^3 + 2*x - 1/3
sage: f.lt()
-2/5*x^3
sage: R(5).lt()
5
sage: R(0).lt()
0
sage: R(0).lt().parent() is R
True
```

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

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

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

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

EXAMPLES:

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

**mod** *(other)*

Remainder of division of self by other.

EXAMPLES:

```
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:
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

**monomial_coefficient** ($m$)

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

**INPUT:**

• $m$ - a monomial

**OUTPUT:**

Coefficient in base ring.

**EXAMPLES:**

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

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

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

Return the list of the monomials in \texttt{self} in a decreasing order of their degrees.

**EXAMPLES:**

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

multiplication_trunc\((\texttt{other}, \texttt{n})\)

Truncated multiplication

**EXAMPLES:**

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

Check that coercion is working:

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

newton_raphson\((\texttt{n}, \texttt{x0})\)

Return a list of \texttt{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),
• $x_0$ - an initial guess $x_0$.

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

If one of the iterates is a critical point of $f$ 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, lengths=False)
```

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

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

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

ALGORITHM: Uses PARI if $lengths$ is False.

```
norm (p)
```

Return the $p$-norm of this polynomial.

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

INPUT:
• $p$ - (positive integer or +infinity) the degree of the norm

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

(continues on next page)
AUTHORS:

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

\texttt{nth\_root (n)}

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

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

EXAMPLES:

\begin{verbatim}
sage: R.<x> = ZZ[]
sage: a = 27 * (x+3)**6 * (x+5)**3
sage: a.nth_root(3)
3*x^3 + 33*x^2 + 117*x + 135
sage: b = 25 * (x^2 + x + 1)
sage: b.nth_root(2)
Traceback (most recent call last):
  ... ValueError: not a 2nd power
sage: R(0).nth_root(3)
0
sage: R.<x> = QQ[]
sage: a = 1/4 * (x/7 + 3/2)^2 * (x/2 + 5/3)^4
sage: a.nth_root(2)
1/56*x^3 + 103/336*x^2 + 365/252*x + 25/12
sage: K.<sqrt2> = QuadraticField(2)
sage: R.<x> = K[]
sage: a = (x + sqrt2)^3 * (1+sqrt2)*x - 1/sqrt2
sage: b = a.nth_root(3); b
(2*sqrt2 + 3)*x^3 + (2*sqrt2 + 2)*x^2 + (-2*sqrt2 - 3/2)*x + 1/2*sqrt2
sage: b^3 == a
True
sage: R.<x> = QQbar[]
sage: p = x**3 + QQbar(2).sqrt() * x - QQbar(3).sqrt()
sage: r = (p**5).nth_root(5)
sage: r * p[0] == p * r[0]
\end{verbatim}

(True

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

sage: z = GF(4).gen()
sage: R.<x> = GF(4)[]
sage: p = z*x**4 + 2*x - 1
sage: r = (p**15).nth_root(15)
sage: r * p[0] == p * r[0]
True
sage: ((x+1)**2).nth_root(2)
x + 1
sage: ((x+1)**4).nth_root(4)
x + 1
sage: ((x+1)**12).nth_root(12)
x + 1
sage: (x^4 + x^3 + 1).nth_root(2)
Traceback (most recent call last):
...
ValueError: not a 2nd power
sage: p = (x+1)^17 + x^17
sage: r = p.nth_root(17)
Traceback (most recent call last):
...
ValueError: not a 17th power

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

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

\texttt{number\_of\_real\_roots}()

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

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

number_of_roots_in_interval \( (a=None, b=None) \)
Return the number of roots of this polynomial in the interval \([a,b]\), counted without multiplicity. The endpoints \(a, b\) default to -Infinity, Infinity (which are also valid input values).

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

EXAMPLES:

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

number_of_terms()
Return the number of non-zero coefficients of \(self\).
Also called weight, Hamming weight or sparsity.

EXAMPLES:

sage: R.<x> = ZZ[]
sage: f = x^3 - x
sage: f.number_of_terms()
2
sage: R(0).number_of_terms()
0

(continues on next page)
sage: f = (x+1)^100
sage: f.number_of_terms()
101
sage: S = GF(5)['y']
sage: S(f).number_of_terms()
5
sage: cyclotomic_polynomial(105).number_of_terms()
33

The method `hamming_weight()` is an alias:

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

**numerator()**

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

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

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

**EXAMPLES:**

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

```
sage: R.<x> = ZZ[
  sage: f = x^3 + 17*x + 1
  sage: f.numerator()
  x^3 + 17*x + 1
  sage: f == f.numerator()  # This is not the numerator of the rational function
  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()  # This is not the numerator of the rational function
  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: R.<x> = ZZ[]
sage: (x^2+x).ord(x+1)
   1

padded_list \(n=None\)
Return list of coefficients of self up to (but not including) \(q^n\).
Includes 0’s in the list on the right so that the list has length \(n\).

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

EXAMPLES:

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:
**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
```

**power_trunc (n, prec)**

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

**INPUT:**

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

**EXAMPLES:**

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

(continues on next page)
....: q2 = q2.truncate(100)
....: assert q1 == q2, "i = {}".format(i)

prec()

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

EXAMPLES:

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

pseudo_quo_rem(other)

Compute the pseudo-division of two polynomials.

INPUT:
• other – a nonzero polynomial

OUTPUT:

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

ALGORITHM:

Algorithm 3.1.2 in [Coh1993].

EXAMPLES:

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

radical()

Return the radical of self.

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

EXAMPLES:
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
```

```
rational_reconstruct (m, n_deg=None, d_deg=None)

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

INPUT:

• m – a univariate polynomial
• n_deg – (optional) an integer; the default is \(\lfloor \frac{\deg(m) - 1}{2} \rfloor\)
• d_deg – (optional) an integer; the default is \(\lfloor \frac{\deg(m) - 1}{2} \rfloor\)

ALGORITHM:

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

EXAMPLES:

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

```
sage: z = PolynomialRing(QQ, 'z').gen()
sage: p = -z**16 - z**15 - z**14 + z**13 + z**12 + z**11 - z**5 - z**4 - z**3 + z**2 + z + 1
```

```
sage: m = z**21
sage: n, d = p.rational_reconstruct(m)
sage: print((n, d))
(z^4 + 2*z^3 + 3*z^2 + 2*z + 1, z^10 + z^9 + z^8 + z^7 + z^6 + z^5 + z^4 + z^3 + z^2 + z + 1)
sage: print(((p*d - n) % m).is_zero())
True
```

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

```
sage: z = PolynomialRing(ZZ, 'z').gen()
sage: p = -z**16 - z**15 - z**14 + z**13 + z**12 + z**11 - z**5 - z**4 - z**3 + z**2 + z + 1
```

```
sage: m = z**21
sage: n, d = p.rational_reconstruct(m)
sage: print((n, d))
(z^4 + 2*z^3 + 3*z^2 + 2*z + 1, z^10 + z^9 + z^8 + z^7 + z^6 + z^5 + z^4 + z^3 + z^2 + z + 1)
sage: print(((p*d - n) % m).is_zero())
True
```

Over an integral domain d might not be monic:
```
sage: P = PolynomialRing(ZZ,'x')
sage: x = P.gen()
sage: p = 7*x^5 - 10*x^4 + 16*x^3 - 32*x^2 + 128*x + 256
sage: m = x^5
sage: n, d = p.rational_reconstruct(m, 3, 2)
sage: print((n ,d))
(-32*x^3 + 384*x^2 + 2304*x + 2048, 5*x + 8)
sage: print(((p*d - n) % m ).is_zero())
True
sage: n, d = p.rational_reconstruct(m, 4, 0)
sage: print((n ,d))
(-10*x^4 + 16*x^3 - 32*x^2 + 128*x + 256, 1)
sage: print(((p*d - n) % m ).is_zero())
True
```

Over \(\mathbb{Q}(t)[z]\):
```
sage: P = PolynomialRing(QQ, 't')
sage: t = P.gen()
sage: Pz = PolynomialRing(P.fraction_field(), 'z')
sage: z = Pz.gen()
sage: # p = (1 + t^2*z + z^4) / (1 - t*z)
sage: p = (1 + t^2*z + z^4)*(1 - t*z).inverse_mod(z^9)
sage: m = z^9
sage: n, d = p.rational_reconstruct(m)
sage: print((n ,d))
(-1/t*z^4 - t*z - 1/t, z - 1/t)
sage: print(((p*d - n) % m ).is_zero())
True
sage: w = PowerSeriesRing(P.fraction_field(), 'w').gen()
sage: n = -10*t^2*z^4 + (-t^2 + t - 1)*z^3 + (-t - 8)*z^2 + z + 2*t^2 - t
sage: d = z^4 + (2*t + 4)*z^3 + (-t + 5)*z^2 + (t^2 + 2)*z + t^2 + 2*t + 1
sage: prec = 9
sage: nc, dc = Pz((n.subs(z = w)/d.subs(z = w) + O(w^prec)).list()).rational_reconstruct(z^prec)
sage: print((nc, dc) == (n, d))
True
```

Over \(\mathbb{Q}[t][z]\):
```
sage: P = PolynomialRing(QQ, 't')
sage: t = P.gen()
sage: z = PolynomialRing(P, 'z').gen()
sage: # p = (1 + t^2*z + z^4) mod z^9
sage: p = (1 + t^2*z + z^4) * sum((t*z)**i for i in range(9))
sage: m = z^9
sage: n, d = p.rational_reconstruct(m)
sage: print((n ,d))
(-z^4 - t^2*z - 1, t*z - 1)
sage: print(((p*d - n) % m ).is_zero())
True
```

Over \(\mathbb{Q}_5\):
```
sage: x = PolynomialRing(Qp(5),'x').gen()
sage: p = 4*x^5 + 3*x^4 + 2*x^3 + 2*x^2 + 4*x + 2
sage: m = x^6
```
(continues on next page)
Can also be used to obtain known Padé approximations:

```python
sage: z = PowerSeriesRing(QQ, 'z').gen()
sage: P = PolynomialRing(QQ,'x')
sage: x = P.gen()
sage: p = P(exp(z).list())
sage: m = x^5
sage: n, d = p.rational_reconstruct(m, 4, 0)
sage: print((n, d))
(1/24*x^4 + 1/6*x^3 + 1/2*x^2 + x + 1, 1)
sage: print(((p*d - n) % m).is_zero())
True
sage: m = x^3
sage: n, d = p.rational_reconstruct(m, 1, 1)
sage: print((n, d))
(-x - 2, x - 2)
```

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

```python
sage: p = P(log(1-z).list())
sage: m = x^9
sage: n, d = p.rational_reconstruct(m, 4, 4)
sage: print((n, d))
(25/6*x^4 - 130/3*x^3 + 105*x^2 - 70*x, x^4 - 20*x^3 + 90*x^2 - 140*x + 70)
sage: print(((p*d - n) % m).is_zero())
True
```

```python
sage: p = P(sqrt(1+z).list())
sage: m = x^6
sage: n, d = p.rational_reconstruct(m, 3, 2)
sage: print((n, d))
(1/6*x^3 + 3*x^2 + 8*x + 16/3, x^2 + 16/3*x + 16/3)
```

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

```python
sage: p = P(exp(2*z).list())
sage: m = x^7
sage: n, d = p.rational_reconstruct(m, 3, 3)
```

```python
sage: print((n, d))
# absolute tolerance 1e-10
(-x^3 - 6*x^2 - 15*x - 15, x^3 - 6*x^2 + 15*x - 15)
```

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

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

```python
sage: z = PowerSeriesRing(RR, 'z').gen()
sage: P = PolynomialRing(RR,'x')
sage: x = P.gen()
sage: p = P(exp(2*z).list())
sage: m = x^7
sage: n, d = p.rational_reconstruct(m, 3, 3)
```

```python
sage: print((n, d))
(-x^3 - 6*x^2 - 15*x - 15, x^3 - 6*x^2 + 15*x - 15)
```

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

See also:
real_roots()

Return the real roots of this polynomial, without multiplicities.

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

EXAMPLES:

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

reciprocal_transform(R=1, q=1)

Transform a general polynomial into a self-reciprocal polynomial.

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

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

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

See also:
The inverse operation is trace_polynomial().

INPUT:

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

EXAMPLES:

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

resultant(other)

Return the resultant of self and other.

INPUT:

- other – a polynomial

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

ALGORITHM:

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

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

We can compute resultants over univariate and multivariate polynomial rings:

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

\texttt{reverse}(\texttt{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 before reversing it. Assuming that the constant coefficient of \texttt{self} is nonzero, 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

\texttt{revert_series}(n)

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

Currently, this is only implemented over some coefficient rings.

EXAMPLES:

sage: Pol.<x> = QQ[]
sage: (x + x^3/6 + x^5/120).revert_series(6)
3/40*x^5 - 1/6*x^3 + x
sage: Pol.<x> = CBF[]
sage: (x + x^3/6 + x^5/120).revert_series(6)
(([-0.0750000000000000 +/- ...e-17])*x^5 + ([-0.1666666666666667 +/- ...e-16])*x^3 + x)
sage: Pol.<x> = SR[]
sage: x.revert_series(6)
Traceback (most recent call last):
...
NotImplementedError: only implemented for certain base rings

\textbf{root\_field}(\textit{names, check\_irreducible=True})

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

\textbf{EXAMPLES:}

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 \(x^3 + x^2 + 2\)
sage: alpha^2 + 1
\alpha^2 + 1
sage: alpha^3 + alpha^2
1

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

\textbf{roots}(\textit{ring=None, multiplicities=True, algorithm=None, **kwds})

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

\textbf{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:

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

```python
sage: f = -19*x + 884736
sage: f.roots()
[(884736/19, 1)]
sage: f.roots()[(884736/19, 1)]
sage: f.roots()[(884736/19, 1)]
```

```python
sage: K.<z> = CyclotomicField(3)
sage: f = K.defining_polynomial()
sage: f.roots(ring=GF(7))
[(4, 1), (2, 1)]
sage: g = f.change_ring(GF(7))
sage: g.roots()
[(4, 1), (2, 1)]
sage: g.roots(multiplicities=False)
[4, 2]
```
A new ring. In the example below, we add the special method _roots_univariate_polynomial to the base ring, and observe that this method is called instead to find roots of polynomials over this ring. This facility can be used to easily extend root finding to work over new rings you introduce:

```python
sage: R.<x> = QQ[]
sage: (x^2 + 1).roots()
[]
sage: g = lambda f, *args, **kwds: f.change_ring(CDF).roots()
sage: QQ._roots_univariate_polynomial = g
sage: (x^2 + 1).roots()  # abs tol 1e-14
[(2.7755575615628914e-17 - 1.0*I, 1), (0.999999999999997*I, 1)]
sage: del QQ._roots_univariate_polynomial
```

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

```python
sage: x = RR['x'].0
sage: f = x^3 -2
sage: f.roots()
[(1.25992104989487, 1)]
sage: f.factor()
(x - 1.25992104989487) * (x^2 + 1.25992104989487*x + 1.58740105196820)
sage: x = RealField(100)['x'].0
sage: f = x^3 -2
sage: f.roots()
[(1.259921049894873164767210673, 1)]
sage: x = CC['x'].0
sage: f = x^3 -2
sage: f.roots()    # abs tol 1e-14
[(1.25992104989487, 1), (-0.629960524947437 - 1.09112363597172*I, 1), (-0.629960524947437 + 1.09112363597172*I, 1)]
sage: f.roots(algorithm='pari')
[(1.25992104989487, 1), (-0.629960524947437 - 1.09112363597172*I, 1), (-0.629960524947437 + 1.09112363597172*I, 1)]
```

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

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

An example involving large numbers:

```python
sage: x = RR['x'].0
sage: f = x^2 - 1e100
sage: f.roots()    # abs tol 1e-14
[(-1.00000000000000e50, 1), (1.00000000000000e50, 1)]
sage: f = x^10 - 2*(5*x-1)^2
sage: f.roots(multiplicities=False)
[-1.6772670339941..., 0.1999547962..., 0.2000453061..., 1.5763035161844...]
sage: x = CC['x'].0
sage: i = CC.0
sage: f = (x - 1)*(x - i)
```

(continues on next page)
Describing roots using radical expressions:

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

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

```python
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 defects. Until that gets implemented, one such example is the following:

```python
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:

```python
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 +...-
\rightarrow (I + 3)*X^3 - (3*I - 3)*sqrt(2)*X^2 - I*X^2 - (I + 3)*sqrt(2)*X + I*sqrt(2)
sage: f.roots()
[(I, 3), (-2^(1/4), 1), (2^(1/4), 1), (1, 1)]
```

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

```python
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 +...-
\rightarrow 3)*X^3 + (-3*I - 3)*sqrt(2)*X^2 - I*X^2 - (I + 3)*sqrt(2)*X + I*sqrt(2)
sage: f.roots()
[(I, 3), (-2^(1/4), 1), (2^(1/4), 1), (1, 1)]
```

A couple of examples where the base ring does not have a factorization algorithm (yet). Note that this is currently done via a rather naive enumeration, so could be very slow:

```python
sage: R = Integers(6)
sage: S.<x> = R['x']
sage: p = x^2-1
sage: p.roots()
Traceback (most recent call last):
...
```
NotImplementedError: root finding with multiplicities for this polynomial not implemented (try the multiplicities=False option)

```python
sage: p.roots(multiplicities=False)
[5, 1]
sage: R = Integers(9)
sage: A = PolynomialRing(R, 'y')
sage: y = A.gen()
sage: f = 10*y^2 - y^3 - 9
sage: f.roots(multiplicities=False)
[1, 0, 3, 6]
```

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

```python
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, 0.30901699437494773 - 0.951056516295154*I, 0.30901699437494756 + 0.9510565162951525*I]
sage: [z^5 for z in f.roots(multiplicities=False)] # abs tol 2e-14
[0.9999999999999957 - 1.2864981197413038e-15*I, 0.9999999999999976 + 3.06285495141552e-15*I, 1.0000000000000024 + 1.133107779295987e-15*I, 1.0000000000000025 - 2.0212861992297117e-15*I]
sage: f = CDF['x'](1,2,3,4); f
4.0*x^3 + 3.0*x^2 + 2.0*x + 1.0
sage: r = f.roots(multiplicities=False)
sage: [f(a).abs() for a in r] # abs tol 1e-14
```

Another example over RDF:

```python
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:

```python
sage: x = CDF['x'].0
def: i = CDF.0
def: f = x^3 + 2*i; f
x^3 + 2.0*i
sage: f.roots()
[(-1.09112363597172... - 0.62996052494743...*I, 1), (...1.25992104989487...*I, 1), (-1.09112363597172... - 0.62996052494743...*I, 1)]
sage: f.roots(multiplicities=False)
[-1.09112363597172... - 0.62996052494743...*I, ...1.25992104989487...*I, 1]  
[-1.09112363597172... - 0.62996052494743...*I]
sage: [abs(f(z)) for z in f.roots(multiplicities=False)] # abs tol 1e-14
[8.95090418262362e-16, 8.728374398092689e-16, 1.023575053041806e-15]
sage: f = i*x^3 + 2; f
I*x^3 + 2.0
sage: f.roots()
[(-1.09112363597172... + 0.62996052494743...*I, 1), (...1.25992104989487...*I, 1), (-1.09112363597172... + 0.62996052494743...*I, 1)]
```

(continues on next page)
Examples using real root isolation:

```python
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))
[(-0.61803398874989484820458683436563811772030917980576268621354486227?, 1),
˓→(1.61803398874989484820458683436563811772030917980576268621354486223?, 1)]
sage: f.roots(ring=AA)
[(-0.618033988749895?, 1), (1.618033988749895?, 1)]
sage: f = f^2 * (x - 1)
sage: f.roots(ring=RIF)
[(-0.6180339887498948482045868343657?, 2), (1.0000000000000000000000000000000?
˓→, 1), (1.6180339887498948482045868343657?, 2)]
sage: f.roots(ring=RIF, multiplicities=False)
[-0.6180339887498948482045868343657?, 1.0000000000000000000000000000000?, 1.
˓→6180339887498948482045868343657?]
```

Examples using complex root isolation:

```python
sage: x = polygen(ZZ)
sage: p = x^5 - x - 1
sage: p.roots()
[]
sage: p.roots(ring=CIF)
[(1.167303978261419?, 1), (-0.7648844336005847? - 0.3524715460317262?*I, 1),
˓→(-0.7648844336005847? + 0.3524715460317262?*I, 1), (0.1812324446498786? - 1.
˓→0839541013177117?*I, 1), (0.1812324446498786? + 1.0839541013177117?*I, 1)]
sage: p.roots(ring=ComplexIntervalField(200))
[(1.1673039782614184186842560458998484218072056371525489039140082?, 1), (-0."
˓→76488443360058472602982318770854173032899665194736756700778? - 0."
˓→352471546031726249317947091402581054394206480824247332837702?*I, 1),
˓→(-0.76488443360058472602982318770854173032899665194736756700778? + 0."
˓→352471546031726249317947091402581054394206480824247332837702?*I, 1), (-0."
˓→1812324444698758390180023778112063996871646618462304743774? - 1.
˓→0839541013177106684303444492980766574273640243155115654301147?*I, 1),
˓→(0.1812324444698758390180023778112063996871646618462304743774? + 1.
˓→0839541013177106684303444492980766574273640243155115654301147?*I, 1)]
sage: rts = p.roots(ring=QQbar); rts
[(1.167303978261419?, 1), (-0.7648844336005847? - 0.3524715460317262?*I, 1),
˓→(-0.7648844336005847? + 0.3524715460317262?*I, 1), (0.1812324444698786? - 1.
˓→0839541013177117?*I, 1), (0.1812324444698786? + 1.0839541013177117?*I, 1)]
sage: p.roots(ring=AA)
[(1.167303978261419?, 1)]
sage: p = (x - rts[4][0])^2 * (3*x^2 + x + 1)
sage: p.roots(ring=QQbar)
[(-0.1666666666666667? - 0.552770798392567?*I, 1), (-0.1666666666666667? + 0.
˓→552770798392567?*I, 1), (0.1812324444698785? + 1.0839541013177117?*I, 1)]
```

2.1. Univariate Polynomials and Polynomial Rings
In some cases, it is possible to isolate the roots of polynomials over complex ball fields:

```python
sage: Pol.<x> = CBF[]
sage: (x^2 + 2).roots(multiplicities=False)
[[+/- ...e-19] + [+/- ...e-17]*I,
  [+/- ...e-19] + [-1.414213562373095 +/- ...e-17]*I]
sage: (x^3 - 1/2).roots(RBF, multiplicities=False)
[[0.7937005259840997 +/- ...e-17]]
sage: ((x - 1)^2).roots(multiplicities=False, proof=False)
doctest:...
UserWarning: roots may have been lost...
[[1.00000000000 +/- ...e-12] + [+/- ...e-11]*I,
  [1.00000000000 +/- ...e-12] + [+/- ...e-12]*I]
```

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

```python
sage: K.<im> = QuadraticField(-1)
sage: y = polygen(K)
sage: p = y^4 - 2 - im
sage: p.roots(ring=CC)
[(-1.2146389322441... - 0.14142505258239...*I, 1), (-0.14142505... + 1.2146389322441...*I, 1), (0.14142505... - 1.2146389322441...*I, 1), (1.2146389322441... + 0.14142505258239...*I, 1)]
```

We can also find roots over number fields:

```python
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("{} {}", format(fld_in, rts))
....:         for rt in rts:
....:             assert(abs(f2(rt)) <= 1e-10)
....:             assert(rt.parent() == fld_out)

Real Field with 53 bits of precision [1.2599210498948731647672106073]
Real Double Field [1.25999210498948731647672106073]
Complex Field with 53 bits of precision [1.25999210498948731647672106073, -0.62999605249474344170984945576395 + 1.0911236359717214035600726142*I]

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), (1.732050807568879?, 1)]
sage: rts[0][0] == rt2
True

We can handle polynomials with huge coefficients.

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

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

Polynomials using such large coefficients can’t be handled by numpy, but pari can deal with them:
sage: p.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 $\text{cube}_\text{root}(j)$.

### shift $(n)$

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

**EXAMPLES:**

```python
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:

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

**AUTHORS:**

- David Harvey (2006-08-06)

### specialization ($D=\text{None}, \phi=\text{None}$)

Specialization of this polynomial.

Given a family of polynomials defined over a polynomial ring. A specialization is a particular member of that family. The specialization can be specified either by a dictionary or a `SpecializationMorphism`.

**INPUT:**
• D – dictionary (optional)
• phi – SpecializationMorphism (optional)

OUTPUT: a new polynomial

EXAMPLES:

```python
sage: R.<c> = PolynomialRing(ZZ)
sage: S.<z> = PolynomialRing(R)
sage: F = c*z^2 + c^2
sage: F.specialization({c:2})
2*z^2 + 4
```

```python
sage: A.<c> = QQ[]
sage: R.<x> = Frac(A)[]
sage: X = (1 + x/c).specialization({c:20})
sage: X
1/20*x + 1
sage: X.parent()
Univariate Polynomial Ring in x over Rational Field
```

splitting_field(names=None, map=False, **kwds)
Compute the absolute splitting field of a given polynomial.

INPUT:

• names – (default: None) a variable name for the splitting field.
• map – (default: False) also return an embedding of self into the resulting field.
• kwds – additional keywords depending on the type. Currently, only number fields are imple-
mented. 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:

```python
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:

```python
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:
An example over a finite field:

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

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

```python
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)
```

See also:

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

**square()**
Return the square of this polynomial.

**Todo:**
- This is just a placeholder; for now it just uses ordinary multiplication. But generally speaking, squaring is faster than ordinary multiplication, and it’s frequently used, so subclasses may choose to provide a specialised squaring routine.
- Perhaps this even belongs at a lower level? RingElement or something?

**AUTHORS:**
- David Harvey (2006-09-09)
EXAMPLES:

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

**squarefree_decomposition()**

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

EXAMPLES:

```
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)*

Return the Sylvester matrix of self and right.

Note that the Sylvester matrix is not defined if one of the polynomials is zero.

**INPUT:**

- right: a polynomial in the same ring as self.
- variable: optional, included for compatibility with the multivariate case only. The variable of the polynomials.

**EXAMPLES:**

```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = (6*x + 47)*(7*x^2 - 2*x + 38)
sage: g = (6*x + 47)*(3*x^3 + 2*x + 1)
sage: M = f.sylvester_matrix(g)
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_nonzero_element()
sage: h2 = R._random_nonzero_element()
sage: M1 = h1.sylvester_matrix(h2)
sage: M1.determinant() == h1.resultant(h2)
True
```

The rank of the Sylvester matrix is related to the degree of the gcd of self and right:
sage: f.gcd(g).degree() == f.degree() + g.degree() - M.rank()
True
sage: h1.gcd(h2).degree() == h1.degree() + h2.degree() - M1.rank()
True

**symmetric_power** \((k,\;\text{monic}=False)\)

Return the polynomial whose roots are products of \(k\)-th distinct roots of this.

**EXAMPLES:**

```python
sage: x = polygen(QQ)
sage: f = x^4-x+2
sage: [f.symmetric_power(k) for k in range(5)]
[x - 1, x^4 - x + 2, x^6 - 2*x^4 - x^3 - 4*x^2 + 8, x^4 - x^3 + 8, x - 2]
sage: f = x^5-2*x+2
sage: [f.symmetric_power(k) for k in range(6)]
[x - 1, x^5 - 2*x + 2, x^10 + 2*x^8 - 4*x^6 - 8*x^5 - 8*x^4 + 32*x^2 + 64, x^5 + 2*x^4 - 16, x + 2]
sage: R.<a,b,c,d> = ZZ[]
sage: x = polygen(R)
sage: f = (x-a)*(x-b)*(x-c)*(x-d)
sage: [f.symmetric_power(k).factor() for k in range(5)]
[x - 1, (-x + d) * (-x + c) * (-x + b) * (-x + a), (x - c*d) * (x - b*d) * (x - a*d) * (x - b*c) * (x - a*c) * (x - a*b), (x - b*c*d) * (x - a*c*d) * (x - a*b*d) * (x - a*b*c), x - a*b*c*d]
```

**trace_polynomial**

Compute the trace polynomial and cofactor.

The input \(P\) and output \(Q\) satisfy the relation

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

In this relation, \(Q\) has all roots in the real interval \([-2\sqrt{q}, 2\sqrt{q}]\) if and only if \(P\) has all roots on the circle \(|x| = \sqrt{q}\) and \(R\) divides \(x^2 - q\). We thus require that the base ring of this polynomial have a coercion to the real numbers.

**See also:**

The inverse operation is `reciprocal_transform()`.

**OUTPUT:**

- \(Q\) – trace polynomial
- \(R\) – cofactor
- \(q\) – scaling factor

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

```
sage: K.<a> = NumberField(x^2-2, embedding=1.4)
sage: u = x^4 + a*x^3 + 3*x^2 + 2*a*x + 4
sage: u.trace_polynomial()
(x^2 + a*x - 1, 1, 2)
sage: (u*(x^2-2)).trace_polynomial()
(x^2 + a*x - 1, x^2 - 2, 2)
sage: (u*(x^2-2)^2).trace_polynomial()
(x^4 + a*x^3 - 9*x^2 - 8*a*x + 8, 1, 2)
sage: (u*(x^2-2)^3).trace_polynomial()
(x^4 + a*x^3 - 9*x^2 - 8*a*x + 8, x^2 - 2, 2)
sage: u = x^4 + a*x^3 + 3*x^2 + 4*a*x + 16
sage: u.trace_polynomial()
(x^2 + a*x - 5, 1, 4)
sage: (u*(x-2)).trace_polynomial()
(x^2 + a*x - 5, x - 2, 4)
sage: (u*(x+2)).trace_polynomial()
(x^2 + a*x - 5, x + 2, 4)
```

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

Return 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 \(\infty\).

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

The valuation at \(\infty\) is -self.degree().

**EXAMPLES:**

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

(continues on next page)
variable_name()

Return name of variable used in this polynomial as a string.

OUTPUT: string

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: f = t^3 + 3/2*t + 5
sage: f.variable_name()
't'
```

variables()

Return the tuple of variables occurring in this polynomial.

EXAMPLES:

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

A constant polynomial has no variables.

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

xgcd(other)

Return an extended gcd of this polynomial and other.

INPUT:

• other — a polynomial in the same ring as this polynomial

OUTPUT:

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

**Note:** The actual algorithm for computing the extended gcd depends on the base ring underlying the polynomial ring. If the base ring defines a method \(\text{__xgcd_univariate_polynomial}\), then this method will be called (see examples below).

EXAMPLES:

```
sage: R.<x> = QQbar[]
sage: (2+x^2).gcd(2*x)
x
sage: R.zero().gcd(0)
0
sage: (2+x).gcd(0)
x
```
One can easily add \texttt{xgcd} functionality to new rings by providing a method
\_\texttt{xgcd}\_\texttt{univariate}\_\texttt{polynomial}:

\begin{Verbatim}
sage: R.<x> = QQ[]
sage: S.<y> = R[]
sage: h1 = y\ast x
sage: h2 = y\wedge 2\ast x\wedge 2
sage: h1.xgcd(h2)
Traceback (most recent call last):
  ...
NotImplementedError: Univariate Polynomial Ring in x over Rational Field does
\rightarrow\text{not provide an xgcd implementation for univariate polynomials}
sage: T.<x,y> = QQ[]
sage: def poor\_xgcd(f,g):
....:   ret = S(T(f).gcd(g))
....:   if ret \ast= f: return ret,S.one(),S.zero()
....:   if ret \ast= g: return ret,S.zero(),S.one()
....:   raise NotImplementedException
sage: R._xgcd\_univariate\_polynomial = poor\_xgcd
sage: h1.xgcd(h2)
(x\ast y, 1, 0)
sage: del R._xgcd\_univariate\_polynomial
\end{Verbatim}

\texttt{class sage.rings.polynomial.polynomial\_element.PolynomialBaseringInjection}

\texttt{Bases: sage.categories.morphism.Morphism}

This class is used for conversion from a ring to a polynomial over that ring.

It calls the \_\texttt{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.

\textbf{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 \texttt{trac ticket #9944}.

\begin{Verbatim}
sage: R.<x> = Qp(3)[]
sage: R.coerce\_map\_from(R.base\_ring())
Polynomial base injection morphism:
  \text{From: 3-adic Field with capped relative precision 20}
  \text{To:  Univariate Polynomial Ring in x over 3-adic Field with capped relative_}
  \rightarrow\text{precision 20}
sage: R.<x,y> = Qp(3)[]
sage: R.coerce\_map\_from(R.base\_ring())
Polynomial base injection morphism:
  \text{From: 3-adic Field with capped relative precision 20}
  \text{To: Multivariate Polynomial Ring in x, y over 3-adic Field with capped_}
  \rightarrow\text{relative precision 20}
sage: R.<x,y> = QQ[]
sage: R.coerce\_map\_from(R.base\_ring())
Polynomial base injection morphism:
  \text{From: Rational Field}
  \text{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:
\end{Verbatim}

(continues on next page)
By trac ticket \#9944, there are now only very few exceptions:

```python
sage: PolynomialRing(QQ, names=[]).coerce_map_from(QQ)
Call morphism:
  From: Rational Field
  To:   Univariate Polynomial Ring in x over Rational Field
```

**is_injective()**

Return whether this morphism is injective.

**EXAMPLES:**

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

Check that trac ticket \#23203 has been resolved:

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

**is_surjective()**

Return whether this morphism is surjective.

**EXAMPLES:**

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

**section()**

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

**constant_coefficient()**

Return the constant coefficient of this polynomial.

**OUTPUT:** element of base ring

**EXAMPLES:**

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

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

is_term()
Return True if this polynomial is a nonzero element of the base ring times a power of the variable.

EXAMPLES:

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

list (copy=True)
Return a new copy of the list of the underlying elements of self.

EXAMPLES:

```
sage: R.<x> = GF(17)[]
sage: f = (1+2*x)^3 + 3*x; f
8*x^3 + 12*x^2 + 9*x + 1
sage: f.list()
[1, 9, 12, 8]
```

quo_rem (other)
Return the quotient and remainder of the Euclidean division of self and other.

Raises a ZerodivisionError if other is zero. Raises an ArithmeticError if the division is not exact.

AUTHORS:
- Kwankyu Lee (2013-06-02)
- Bruno Grenet (2014-07-13)

EXAMPLES:

```
sage: P.<x> = QQ[]
sage: R.<y> = P[]
sage: f = R.random_element(10)
sage: g = y^5+R.random_element(4)
sage: q, r = f.quo_rem(g)
sage: f == q*g + r
True
sage: g = x*y^5
sage: f.quo_rem(g)
Traceback (most recent call last):
  ...  # ArithmeticError: division non exact (consider coercing to polynomials over the fraction field)
```
\begin{verbatim}
sage: g = 0
sage: f.quo_rem(g)
Traceback (most recent call last):
...
ZeroDivisionError: division by zero polynomial
\end{verbatim}

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

Return this polynomial multiplied by the power \(x^n\).

If \(n\) is negative, terms below \(x^n\) will be discarded. Does not change this polynomial.

**EXAMPLES:**

\begin{verbatim}
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
\end{verbatim}

**AUTHORS:**

- David Harvey (2006-08-06)

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

Return the polynomial of degree ‘< n’ which is equivalent to self modulo \(x^n\).

**EXAMPLES:**

\begin{verbatim}
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
\end{verbatim}

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

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

A dense polynomial over an inexact ring.

**AUTHOR:**

- Xavier Caruso (2013-03)

**degree** \((secure=False)\)

**INPUT:**

- secure – a boolean (default: False)

**OUTPUT:**

The degree of self.
If `secure` is True and the degree of this polynomial is not determined (because the leading coefficient is indistinguishable from 0), an error is raised.

If `secure` is False, the returned value is the largest $n$ so that the coefficient of $x^n$ does not compare equal to 0.

**EXAMPLES:**

```
sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.degree()
1
sage: (f-T).degree()
0
sage: (f-T).degree(secure=True)
Traceback (most recent call last):
  ...PrecisionError: the leading coefficient is indistinguishable from 0
sage: x = O(3^5)
sage: li = [3^i * x for i in range(0,5)]; li
[O(3^5), O(3^6), O(3^7), O(3^8), O(3^9)]
sage: f = R(li); f
O(3^9)*T^4 + O(3^8)*T^3 + O(3^7)*T^2 + O(3^6)*T + O(3^5)
sage: f.degree()
-1
sage: f.degree(secure=True)
Traceback (most recent call last):
  ...PrecisionError: the leading coefficient is indistinguishable from 0
```

**AUTHOR:**

- Xavier Caruso (2013-03)

**prec_degree()**

Return the largest $n$ so that precision information is stored about the coefficient of $x^n$.

Always greater than or equal to degree.

**EXAMPLES:**

```
sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.prec_degree()
1
sage: f.prec_degree()
1
sage: g = f - T; g
O(3^10)*T + 2 + O(3^10)
sage: g.prec_degree()
0
sage: g.prec_degree()
1
```

**AUTHOR:**

2.1. Univariate Polynomials and Polynomial Rings 105
Generic truncated power algorithm

**INPUT:**
- p - a polynomial
- n - an integer (of type `sage.rings.integer.Integer`)
- prec - a precision (should fit into a C long)

**EXAMPLES:**

```python
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:

```python
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
```

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

**INPUT:**
- n - degree of the polynomial

**OUTPUT:**
The discriminant as a polynomial in \( n + 1 \) variables over \( \mathbb{Z} \). The result will be cached, so subsequent computations of discriminants of the same degree will be faster.

**EXAMPLES:**
sage: from sage.rings.polynomial.polynomial_element import universal_discriminant
sage: universal_discriminant(1)
1
sage: universal_discriminant(2)
a1^2 - 4*a0*a2
sage: universal_discriminant(3)
a1^2*a2^2 - 4*a0*a2^3 - 4*a1^3*a3 + 18*a0*a1*a2*a3 - 27*a0^2*a3^2
sage: universal_discriminant(4).degrees()
(3, 4, 4, 4, 3)

See also:

Polynomial.discriminant()

2.1.4 Univariate Polynomials over domains and fields

AUTHORS:

- William Stein: first version
- Martin Albrecht: Added singular coercion.
- David Harvey: split off polynomial_integer_dense_ntl.pyx (2007-09)
- Robert Bradshaw: split off polynomial_modn_dense_ntl.pyx (2007-09)

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

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_domain

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

AUTHOR:

- Xavier Caruso (2013-03)

factor_of_slope (slope=None)

INPUT:

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

OUTPUT:

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

EXAMPLES:

sage: K = Qp(5)
sage: R.<x> = K[]
sage: K = Qp(5)
sage: R.<t> = K[]
sage: f = 5 + 3*t + t^4 + 25*t^10
sage: f.newton_slopes()
[1, 0, 0, -1/3, -1/3, -1/3, -1/3, -1/3, -1/3, -1/3]
sage: g = f.factor_of_slope(0)
sage: g.newton_slopes()
If `slope` is not a slope of `self`, the corresponding factor is 1:

```
sage: f.factor_of_slope(-1)
1 + O(5^20)
```

AUTHOR:
• Xavier Caruso (2013-03-23)

`hensel_lift(a)`

Lift `a` to a root of this polynomial (using Newton iteration).

If `a` is not close enough to a root (so that Newton iteration does not converge), an error is raised.

EXAMPLES:
```
sage: K = Qp(5, 10)
sage: P.<x> = PolynomialRing(K)
sage: f = x^2 + 1
sage: root = f.hensel_lift(2); root
2 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9 + O(5^10)
sage: f(root)
O(5^10)
sage: g = (x^2 + 1)*(x - 7)
sage: g.hensel_lift(2)  # here, 2 is a multiple root modulo p
Traceback (most recent call last):
...  
ValueError: a is not close enough to a root of this polynomial
```

AUTHOR:
• Xavier Caruso (2013-03-23)

`newton_polygon()`

Returns a list of vertices of the Newton polygon of this polynomial.

**Note:** If some coefficients have not enough precision an error is raised.

**EXAMPLES:**
```
sage: K = Qp(5)
sage: R.<t> = K[]
sage: f = 5 + 3*t + t^4 + 25*t^10
sage: f.newton_polygon()
Finite Newton polygon with 4 vertices: (0, 1), (1, 0), (4, 0), (10, 2)
```

(continues on next page)
AUTHOR:
- Xavier Caruso (2013-03-20)

newton_slopes (repetition=True)

Returns a list of the Newton slopes of this polynomial.

These are the valuations of the roots of this polynomial.

If repetition is True, each slope is repeated a number of times equal to its multiplicity. Otherwise it appears only one time.

EXAMPLES:

```sage
sage: K = Qp(5)
sage: R.<t> = K[]
sage: f = 5 + 3*t + t^4 + 25*t^10
sage: f.newton_polygon()
Finite Newton polygon with 4 vertices: (0, 1), (1, 0), (4, 0), (10, 2)
sage: f.newton_slopes()
[1, 0, 0, 0, -1/3, -1/3, -1/3, -1/3, -1/3, -1/3]
sage: f.newton_slopes(repetition=False)
[1, 0, -1/3]
```

AUTHOR:
- Xavier Caruso (2013-03-20)

slope_factorization()

Return a factorization of self into a product of factors corresponding to each slope in the Newton polygon.

EXAMPLES:

```sage
sage: K = Qp(5)
sage: R.<x> = K[]
sage: K = Qp(5)
sage: R.<t> = K[]
sage: f = 5 + 3*t + t^4 + 25*t^10
sage: f.newton_polygon()
Finite Newton polygon with 4 vertices: (0, 1), (1, 0), (4, 0), (10, 2)
sage: f.newton_slopes()
[1, 0, 0, 0, -1/3, -1/3, -1/3, -1/3, -1/3, -1/3]
sage: F = f.slope_factorization()
sage: F.prod() == f
True
sage: for (f,_) in F:
.....:     print(f.newton_slopes())
[-1/3, -1/3, -1/3, -1/3, -1/3, -1/3]
[0, 0, 0]
[1]
```

AUTHOR:
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: 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

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

quo_rem(other)

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

EXAMPLES:

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

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse

Bases: sage.rings.polynomial.polynomial_element.Polynomial

A generic sparse polynomial.

The Polynomial_generic_sparse class defines functionality for sparse polynomials over any base ring. A sparse polynomial is represented using a dictionary which maps each exponent to the corresponding coefficient. The coefficients must never be zero.

EXAMPLES:

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

A more extensive example:
 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]

gcd (other, algorithm=None)
  Return the gcd of this polynomial and other
INPUT:
- **other** – a polynomial defined over the same ring as this polynomial.

ALGORITHM:

Two algorithms are provided:

- **generic**: Uses the generic implementation, which depends on the base ring being a UFD or a field.
- **dense**: The polynomials are converted to the dense representation, their gcd is computed and is converted back to the sparse representation.

Default is **dense** for polynomials over **ZZ** and **generic** in the other cases.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: p = x^6 + 7*x^5 + 8*x^4 + 6*x^3 + 2*x^2 + x + 2
sage: q = 2*x^4 - x^3 - 2*x^2 - 4*x - 1
sage: gcd(p, q)
x^2 + x + 1
sage: gcd(p, q, algorithm = "dense")
x^2 + x + 1
sage: gcd(p, q, algorithm = "generic")
x^2 + x + 1
sage: gcd(p, q, algorithm = "foobar")
Traceback (most recent call last):
... ValueError: Unknown algorithm 'foobar'
```

**integral**(var=None)

Return the integral of this polynomial.

By default, the integration variable is the variable of the polynomial.

Otherwise, the integration variable is the optional parameter var

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

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: (1 + 3*x^10 - 2*x^100).integral()
-2/101*x^101 + 3/11*x^11 + x
```

**list**(copy=True)

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

EXAMPLES:

```
sage: R.<z> = PolynomialRing(Integers(100), sparse=True)
sage: f = 13*z^5 + 15*z^2 + 17*z
sage: f.list()
[0, 17, 15, 0, 0, 13]
```

**number_of_terms**()

Return the number of nonzero terms.

EXAMPLES:
 sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: p = x^100 - 3*x^10 + 12
sage: p.number_of_terms()
3

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

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

 Raises \texttt{ZerodivisionError} if \texttt{other} is zero. Raises \texttt{ArithmeticError} if \texttt{other} has a nonunit leading coefficient.

\textbf{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: Division non exact (consider coercing to polynomials over the fraction field)
 sage: g = 0
 sage: f.quo_rem(g)
 Traceback (most recent call last):
   ...  
 ZeroDivisionError: Division by zero polynomial

\textbf{AUTHORS:}

• Bruno Grenet (2014-07-09)

\textbf{reverse}(\textit{degree}=\texttt{None})

Return this polynomial but with the coefficients reversed.

If an optional degree argument is given the coefficient list will be truncated or zero padded as necessary and the reverse polynomial will have the specified degree.

\textbf{EXAMPLES:}

 sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: p = x^100 + 2*x + 4
 sage: p.reverse()
x^1267650600228229401496703205372 + 2
 sage: p.reverse(10)
x^6

\textbf{shift}(\textit{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.

\textbf{EXAMPLES:}

 sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: p = x^100000 + 2*x^2^100
 sage: p.reverse()
x^1267650600228229401496703205372 + 2
 sage: p.reverse(10)
x^6

(continues on next page)
sage: type(p)
<class 'sage.rings.polynomial.polynomial_ring.PolynomialRing_integral_domain_
˓
_neurons
˓

sage: p.shift(0)
x^100000 + 2*x + 4
sage: p.shift(-1)
x^99999 + 2
sage: p.shift(-100002)
0
sage: p.shift(2)
x^100002 + 2*x^3 + 4*x^2

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

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

EXAMPLES:

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

valuation()
Return the valuation of self.

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_cdvf

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse

class sage.rings.polynomial.polynomial_element Generic.Polynomial_generic_sparse_cdvf

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_cdv
class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_cdvr

parent, x=None, check=True, is_gen=False, construct=False

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_cdvr
sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdvr

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_field

parent, x=None, check=True, is_gen=False, construct=False

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

EXAMPLES:

sage: R.<x> = PolynomialRing(Frac(RR['t']), sparse=True)
sage: f = x^3 - x + 17
sage: type(f)
<class 'sage.rings.polynomial.polynomial_ring.PolynomialRing_field_with_category.element_class'>

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

AUTHOR: - Martin Albrecht (2008-10) initial implementation

sage.rings.polynomial.polynomial_gf2x.GF2X_BuildIrred_list(n)

Return the list of coefficients of the lexicographically smallest irreducible polynomial of degree n over the field of 2 elements.

EXAMPLES:

sage: from sage.rings.polynomial.polynomial_gf2x import GF2X_BuildIrred_list
sage: GF2X_BuildIrred_list(2)
[1, 1, 1]
sage: GF2X_BuildIrred_list(3)
[1, 1, 0, 1]
sage: GF2X_BuildIrred_list(4)
[1, 1, 0, 0, 1]
sage: GF(2))['x'](GF2X_BuildIrred_list(33))
x^33 + x^6 + x^3 + x + 1

sage.rings.polynomial.polynomial_gf2x.GF2X_BuildRandomIrred_list(n)

Return the list of coefficients of an irreducible polynomial of degree n of minimal weight over the field of 2 elements.

EXAMPLES:
sage: from sage.rings.polynomial.polynomial_gf2x import GF2X_BuildRandomIrred_list
sage: GF2X_BuildRandomIrred_list(2)
[1, 1, 1]
sage: GF2X_BuildRandomIrred_list(3) in [[1, 1, 0, 1], [1, 0, 1, 1]]
True

sage.rings.polynomial.polynomial_gf2x.GF2X_BuildSparseIrred_list(n)
Return the list of coefficients of an irreducible polynomial of degree \( n \) of minimal weight over the field of 2 elements.

EXAMPLES:

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

class sage.rings.polynomial.polynomial_gf2x.Polynomial_GF2X
Bases: sage.rings.polynomial.polynomial_gf2x.Polynomial_template

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

EXAMPLES:

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

\texttt{is_irreducible}()
Return whether this polynomial is irreducible over \( F_2 \).

EXAMPLES:

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

Test that caching works:

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

\texttt{modular_composition} (g, h, algorithm=None)
Compute \( f(g) \) (mod \( h \)).


INPUT:
- \( g \) – a polynomial
• \( h \) – a polynomial

• \texttt{algorithm} – either ‘native’ or ‘ntl’ (default: ‘native’)

EXAMPLES:

```python
sage: P.<x> = GF(2)[]
sage: r = 279
sage: f = x^r + x +1
sage: g = x^r
sage: g.modular_composition(g, f) == g(g) % f
True

sage: P.<x> = GF(2)[]
sage: f = x^29 + x^24 + x^22 + x^21 + x^16 + x^15 + x^14 + x^10 + x^9 -> x^8 + x^7 + x^6 + x^5 + x^2
sage: g = x^31 + x^30 + x^28 + x^26 + x^24 + x^21 + x^19 + x^18 + x^11 + x^10 -> x^9 + x^8 + x^5 + x^2 + 1
sage: h = x^30 + x^28 + x^26 + x^25 + x^24 + x^22 + x^21 + x^18 + x^17 + x^15 -> x^13 + x^12 + x^11 + x^10 + x^9 + x^4
sage: f.modular_composition(g,h) == f(g) % h
True
```

AUTHORS:

• Paul Zimmermann (2008-10) initial implementation

• Martin Albrecht (2008-10) performance improvements

```python
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 \texttt{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 \texttt{sage.rings.polynomial.polynomial_gf2x} for an example.

We illustrate the generic glueing using univariate polynomials over GF(2).

\textbf{Note:} Implementations using this template MUST implement coercion from base ring elements and \texttt{get_unsafe()}. See \texttt{Polynomial_GF2X} for an example.

```python
def degree()

EXAMPLES:

```
\texttt{gcd}(\textit{other})

Return the greatest common divisor of self and other.

\textbf{EXAMPLES:}

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

\texttt{get_cparent()} 

\texttt{is_gen()} 

\textbf{EXAMPLES:}

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

\texttt{is_one()} 

\textbf{EXAMPLES:}

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

\texttt{is_zero()} 

\textbf{EXAMPLES:}

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

\texttt{list}(\textit{copy=True}) 

\textbf{EXAMPLES:}

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

\texttt{quo_rem}(\textit{right}) 

\textbf{EXAMPLES:}

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

\texttt{shift}(\textit{n}) 

\textbf{EXAMPLES:}
\begin{verbatim}
sage: P.<x> = GF(2)[]
sage: f = x^3 + x^2 + 1
sage: f.shift(1)
x^3 + x^2 + x
sage: f.shift(-1)
x^2 + x

\verb|truncate| \((n)\)
Returns this polynomial mod \(x^n\).

EXAMPLES:

\begin{verbatim}
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
\end{verbatim}
If the precision is higher than the degree of the polynomial then the polynomial itself is returned:

\begin{verbatim}
sage: f.truncate(10) is f
True
\end{verbatim}

\verb|xgcd| \((other)\)
Computes extended gcd of self and other.

EXAMPLES:

\begin{verbatim}
sage: P.<x> = GF(7)[]
sage: f = x*(x+1)
sage: f.xgcd(x+1)
(x + 1, 0, 1)
sage: f.xgcd(x^2)
(x, 1, 6)
\end{verbatim}
\end{verbatim}

\section*{2.1.6 Univariate polynomials over number fields.}

AUTHOR:
\begin{itemize}
  \item Luis Felipe Tabera Alonso (2014-02): initial version.
\end{itemize}

EXAMPLES:

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

\begin{verbatim}
sage: N.<a> = NumberField(x^2-2)
sage: K.<x> = N[
\end{verbatim}
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
```
The monic gcd of self and other.

EXAMPLES:

```python
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
```

class sage.rings.polynomial.polynomial_number_field.Polynomial_relative_number_field_dense

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_field

Class of dense univariate polynomials over a relative number field.

gcd(\text{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:

• \text{other} – a polynomial with the same parent as self.

OUTPUT:

• The monic gcd of self and other.

See \texttt{Polynomial_absolute_number_field_dense.gcd()} for more details.

EXAMPLES:

```python
sage: N = QQ[sqrt(2), sqrt(3)]
sage: s2, s3 = N.gens()
sage: x = polygen(N)
sage: f = x^4 - 5*x^2 + 6
sage: g = x^3 + (-2*s2 + s3)*x^2 + (-2*s3*s2 + 2)*x + 2*s3
sage: gcd(f, g)  
x^2 + (-sqrt2 + sqrt3)*x - sqrt3*sqrt2
sage: f.gcd(g)  
x^2 + (-sqrt2 + sqrt3)*x - sqrt3*sqrt2
```
2.1.7 Dense univariate polynomials over $\mathbb{Z}$, implemented using FLINT.

AUTHORS:
- David Harvey: rewrote to talk to NTL directly, instead of via ntl.pyx (2007-09); a lot of this was based on Joel Mohler’s recent rewrite of the NTL wrapper
- David Harvey: split off from polynomial_element_generic.py (2007-09)
- Burcin Erocal: rewrote to use FLINT (2008-06-16)

```python
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.
```

- **add** (right)
  Returns self plus right.
  
  EXAMPLES:
  ```python
  sage: R.<x> = PolynomialRing(ZZ)
  sage: f = 2*x + 1
  sage: g = -3*x^2 + 6
  sage: f + g
  -3*x^2 + 2*x + 7
  ```

- **sub** (right)
  Return self minus right.
  
  EXAMPLES:
  ```python
  sage: R.<x> = PolynomialRing(ZZ)
  sage: f = 2*x + 1
  sage: g = -3*x^2 + 6
  sage: f - g
  3*x^2 + 2*x - 5
  ```

- **lmul** (right)
  Returns self multiplied by right, where right is a scalar (integer).
  
  EXAMPLES:
  ```python
  sage: R.<x> = PolynomialRing(ZZ)
  sage: x*3
  3*x
  sage: (2*x^2 + 4)*3
  6*x^2 + 12
  ```

- **rmul** (right)
  Returns self multiplied by right, where right is a scalar (integer).
  
  EXAMPLES:
  ```python
  sage: R.<x> = PolynomialRing(ZZ)
  sage: 3*x
  3*x
  sage: 3*(2*x^2 + 4)
  6*x^2 + 12
  ```

2.1. Univariate Polynomials and Polynomial Rings
Returns self multiplied by right.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(ZZ)
sage: (x - 2)*(x^2 - 8*x + 16)
x^3 - 10*x^2 + 32*x - 32
```

Truncated multiplication

See also:

- `_mul_()` for standard multiplication

EXAMPLES:

```python
sage: x = polygen(ZZ)
sage: p1 = 1 + x + x^2 + x^4
sage: p2 = -2 + 3*x^2 + 5*x^4
sage: p1._mul_trunc_(p2, 4)
3*x^3 + x^2 - 2*x - 2
sage: (p1*p2).truncate(4)
3*x^3 + x^2 - 2*x - 2
sage: p1._mul_trunc_(p2, 6)
5*x^5 + 6*x^4 + 3*x^3 + x^2 - 2*x - 2
```

Return the greatest common divisor of the coefficients of this polynomial. The sign is the sign of the leading coefficient. The content of the zero polynomial is zero.

EXAMPLES:

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

Return the degree of this polynomial.

The zero polynomial has degree -1.

EXAMPLES:

```python
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 \texttt{(proof=True)}

Return the discriminant of self, which is by definition

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

where \( m = \text{deg}(a) \), and \( \text{lc}(a) \) is the leading coefficient of \( a \). If \texttt{proof} is False (the default is True), then this function may use a randomized strategy that errors with probability no more than \( 2^{-80} \).

EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: f = 3*x^3 + 2*x + 1
sage: f.discriminant()
-339
sage: f.discriminant(proof=False)
-339
```

discriminant \texttt{(proof=True)}

Return the discriminant of self, which is by definition

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

where \( m = \text{deg}(a) \), and \( \text{lc}(a) \) is the leading coefficient of \( a \). If \texttt{proof} is False (the default is True), then this function may use a randomized strategy that errors with probability no more than \( 2^{-80} \).

EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: f = 3*x^3 + 2*x + 1
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:

```python
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 \texttt{(p)}

Return the factorization of self modulo the prime \( p \).

INPUT:
• $p$ – prime

OUTPUT:

factorization of self reduced modulo $p$.

EXAMPLES:

```python
sage: R.<x> = ZZ['x']
sage: f = -3*x*(x-2)*(x-9) + x
sage: f.factor_mod(3)
x
sage: f = -3*x*(x-2)*(x-9)
sage: f.factor_mod(3)
Traceback (most recent call last):
  ... ArithmeticError: factorization of 0 is not defined
sage: f = 2*x*(x-2)*(x-9)
sage: f.factor_mod(7)
(2) * x * (x + 5)^2
```

`factor_padic(p, prec=10)`

Return $p$-adic factorization of self to given precision.

INPUT:

• $p$ – prime
• $prec$ – integer; the precision

OUTPUT:

• factorization of self over the completion at $p$.

EXAMPLES:

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

A more difficult example:

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

gcd(right)

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

EXAMPLES:

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

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

**EXAMPLES:**

```python
sage: x = polygen(ZZ)
sage: p = 1+x+2*x^2
sage: q5 = p.inverse_series_trunc(5)
sage: q5
-x^4 + 3*x^3 - x^2 - x + 1
sage: p*q5
-2*x^6 + 5*x^5 + 1
sage: (x-1).inverse_series_trunc(5)
-x^4 - x^3 - x^2 - x - 1
sage: q100 = p.inverse_series_trunc(100)
sage: (q100 * p).truncate(100)
1
```

**is_one** ()

Returns True if self is equal to one.

**EXAMPLES:**

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

**is_zero** ()

Returns True if self is equal to zero.

**EXAMPLES:**

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

**lcm** *(right)*

Return the LCM of self and right.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(ZZ)
sage: f = (6*x + 47)*(7*x^2 - 2*x + 38)
sage: g = (6*x + 47)*(3*x^3 + 2*x + 1)
sage: h = f.lcm(g); h
126*x^6 + 951*x^5 + 486*x^4 + 6034*x^3 + 585*x^2 + 3706*x + 1786
sage: h == (6*x + 47)*(7*x^2 - 2*x + 38)*(3*x^3 + 2*x + 1)
True
```
list (copy=True)
Return a new copy of the list of the underlying elements of self.

EXAMPLES:

```
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

\[ \text{lead}(B)^d A = BQ + R \]

where R has degree less than that of B.

INPUT:
- B – a polynomial over \( \mathbb{Z} \)

OUTPUT:
- Q, R – polynomials
- d – nonnegative integer

EXAMPLES:

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

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

EXAMPLES:

```
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

(continues on next page)

(continued from previous page)

```sage
sage: f = (x^2 + 3) * (2*x - 1)
sage: f.quo_rem(2*x - 1)
(x^2 + 3, 0)
sage: f = x^2
sage: f.quo_rem(2*x - 1)
(0, x^2)
```

### real_root_intervals()
Returns isolating intervals for the real roots of this polynomial.

**EXAMPLES:** We compute the roots of the characteristic polynomial of some Salem numbers:

```sage
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
sage: x = PolynomialRing(ZZ,'x').0
sage: f = x^3 + x + 1; g = x^3 - x - 1
sage: r = f.resultant(g); r
-8
sage: r.parent() is ZZ
True
```

### reverse (degree=None)
Return a polynomial with the coefficients of this polynomial reversed.

If an optional degree argument is given the coefficient list will be truncated or zero padded as necessary before computing the reverse.

**EXAMPLES:**

```sage
sage: R.<x> = ZZ[]
sage: p = R([1,2,3,4]); p
4*x^3 + 3*x^2 + 2*x + 1
sage: p.reverse()
x^3 + 2*x^2 + 3*x + 4
sage: p.reverse(degree=6)
x^6 + 2*x^5 + 3*x^4 + 4*x^3
sage: p.reverse(degree=2)
x^2 + 2*x + 3
```
revert_series \((n)\)

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

**EXAMPLES:**

```python
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:**

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

xgcd \((right)\)

Return a triple \((g, s, t)\) such that \(g = s \ast self + t \ast right\) 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:**

```python
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

2.1.8 Dense univariate polynomials over \( \mathbb{Z} \), implemented using NTL.

AUTHORS:

- David Harvey: split off from polynomial_element_generic.py (2007-09)
- David Harvey: rewrote to talk to NTL directly, instead of via ntl.pyx (2007-09); a lot of this was based on Joel Mohler’s recent rewrite of the NTL wrapper

Sage includes two implementations of dense univariate polynomials over \( \mathbb{Z} \); this file contains the implementation based on NTL, but there is also an implementation based on FLINT in sage.rings.polynomial.polynomial_integer_dense_flint.

The FLINT implementation is preferred (FLINT’s arithmetic operations are generally faster), so it is the default; to use the NTL implementation, you can do:

```
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
```

(continues on next page)
degree (gen=None)

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

EXAMPLES:

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

discriminant (proof=True)

Return the discriminant of self, which is by definition

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

where \( m = \deg(a) \), and \( \text{lc}(a) \) is the leading coefficient of \( a \). If \( \text{proof} \) is False (the default is True), then this function may use a randomized strategy that errors with probability no more than \( 2^{-80} \).

EXAMPLES:

```python
sage: 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:

```python
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
```
factor_mod \( (p) \)
Return the factorization of self modulo the prime \( p \).

**INPUT:**

- \( p \) – prime

**OUTPUT:** factorization of self reduced modulo \( p \).

**EXAMPLES:**

```python
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)
Traceback (most recent call last):
  ... ArithmeticError: factorization of 0 is not defined
```

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:**

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

A more difficult example:

```python
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 *
```

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

**EXAMPLES:**

2.1. Univariate Polynomials and Polynomial Rings 133
.. code-block:: sage

    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

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

Return the LCM of self and right.

\textbf{EXAMPLES:}

\begin{verbatim}
    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
\end{verbatim}

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

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

\textbf{EXAMPLES:}

\begin{verbatim}
    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()
    []
\end{verbatim}

\textbf{quo_rem}(\textit{right})

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

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

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

\textbf{EXAMPLES:}

\begin{verbatim}
    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
\end{verbatim}
real_root_intervals()

Returns isolating intervals for the real roots of this polynomial.

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

```python
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:

```python
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:

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

xgcd (right)

This function can’t in general return $(g,s,t)$ as above, since they need not exist. Instead, over the integers, we first multiply $g$ by a divisor of the resultant of $a/g$ and $b/g$, up to sign, and return $g$, $u$, $v$ such that $g = s*\text{self} + s*\text{right}$. But note that this $g$ may be a multiple of the gcd.
If `self` and `right` are coprime as polynomials over the rationals, then `g` is guaranteed to be the resultant of `self` and `right`, as a constant polynomial.

**EXAMPLES:**

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

## 2.1.9 Univariate polynomials over \( \mathbb{Q} \) implemented via FLINT

**AUTHOR:**

- Sebastian Pancratz

```python
class sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint
    Bases: sage.rings.polynomial.polynomial_element.Polynomial

Univariate polynomials over the rationals, implemented via FLINT.

Internally, we represent rational polynomial as the quotient of an integer polynomial and a positive denominator which is coprime to the content of the numerator.

**__add__(right)**

Returns the sum of two rational polynomials.

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: f = 2/3 + t + 2*t^3
sage: g = -1 + t/3 - 10/11*t^4
sage: f + g
-10/11*t^4 + 2*t^3 + 4/3*t - 1/3
```

**__sub__(right)**

Returns the difference of two rational polynomials.

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: f = -10/11*t^4 + 2*t^3 + 4/3*t - 1/3
sage: g = 2*t^3
sage: f - g
# indirect doctest
-10/11*t^4 + 4/3*t - 1/3
```
_lmul_(right)
Returns self * right, where right is a rational number.

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: f = 3/2*t^3 - t + 1/3
sage: f * 6 # indirect doctest
9*t^3 - 6*t + 2
```

_rmul_(left)
Returns left * self, where left is a rational number.

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: f = 3/2*t^3 - t + 1/3
sage: 6 * f # indirect doctest
9*t^3 - 6*t + 2
```

_mul_(right)
Returns the product of self and right.

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: f = -1 + 3*t/2 - t^3
sage: g = 2/3 + 7/3*t + 3*t^2
sage: f * g # indirect doctest
-3*t^5 - 7/3*t^4 + 23/6*t^3 + 1/2*t^2 - 4/3*t - 2/3
```

_mul_trunc_(right, n)
Truncated multiplication.

EXAMPLES:

```
sage: x = polygen(QQ)
sage: p1 = 1/2 - 3*x + 2/7*x^3
sage: p2 = x + 2/5*x^5 + x^7
sage: p1._mul_trunc_(p2, 5)
2/7*x^4 - 3*x^2 + 1/2*x
sage: (p1*p2).truncate(5)
2/7*x^4 - 3*x^2 + 1/2*x
sage: p1._mul_trunc_(p2, 1)
0
sage: p1._mul_trunc_(p2, 0)
Traceback (most recent call last):
... ValueError: n must be > 0
```

ALGORITHM:
Call the FLINT method fmpq_poly_mullow.

degree()
Return the degree of self.
By convention, the degree of the zero polynomial is -1.

EXAMPLES:
\begin{verbatim}
sage: R.<t> = QQ[]
sage: f = 1 + t + t^2/2 + t^3/3 + t^4/4
sage: f.degree()
4
sage: g = R(0)

sage: g.degree()
-1
\end{verbatim}

**denominator()**

Returns the denominator of self.

**EXAMPLES:**

\begin{verbatim}
sage: R.<t> = QQ[

sage: f = (3 * t^3 + 1) / -3
sage: f.denominator()
3
\end{verbatim}

**disc()**

Returns the discriminant of this polynomial.

The discriminant $R_n$ is defined as

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

where $n$ is the degree of this polynomial, $a_n$ is the leading coefficient and the roots over $\overline{\mathbb{Q}}$ are $r_1, \ldots, r_n$.

The discriminant of constant polynomials is defined to be 0.

**OUTPUT:**

- Discriminant, an element of the base ring of the polynomial ring

**Note:** Note the identity $R_n(f) := (-1)^{n(n-1)/2}R(f, f')a_n^kn - k - 2)$, where $n$ is the degree of this polynomial, $a_n$ is the leading coefficient, $f'$ is the derivative of $f$, and $k$ is the degree of $f'$. Calls \texttt{resultant()}.

**ALGORITHM:**

Use PARI.

**EXAMPLES:**

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

\begin{verbatim}
sage: R.<t> = QQ[

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

sage: R.<t> = QQ[

sage: f = 2*t^3 + t + 1
sage: d = f.discriminant(); d
-116
\end{verbatim}

138 Chapter 2. Univariate Polynomials
```python
sage: R.<t> = QQ[]
sage: f = t^3 + 3*t - 17
sage: f.discriminant()
-7911
```

**discriminant()**

Returns the discriminant of this polynomial.

The discriminant $R_n$ is defined as

$$R_n = a_{n}^{2n-2} \prod_{1 \leq i < j \leq n} (r_i - r_j)^2,$$

where $n$ is the degree of this polynomial, $a_n$ is the leading coefficient and the roots over $\overline{\mathbb{Q}}$ are $r_1, \ldots, r_n$.

The discriminant of constant polynomials is defined to be 0.

**OUTPUT:**

- Discriminant, an element of the base ring of the polynomial ring

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

**ALGORITHM:**

Use PARI.

**EXAMPLES:**

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

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

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

```python
sage: R.<t> = QQ[]
sage: f = t^3 + 3*t - 17
sage: f.discriminant()
-7911
```

**factor_mod**

Returns the factorization of self modulo the prime $p$.

Assumes that the degree of this polynomial is at least one, and raises a `ValueError` otherwise.

**INPUT:**

2.1. Univariate Polynomials and Polynomial Rings
• **p** - Prime number

**OUTPUT:**

• Factorization of this polynomial modulo **p**

**EXAMPLES:**

```python
sage: R.<x> = QQ[]
sage: (x^5 + 17*x^3 + x+ 3).factor_mod(3)
x * (x^2 + 1)^2
sage: (x^5 + 2).factor_mod(5)
(x + 2)^5
```

Variable names that are reserved in PARI, such as `zeta`, are supported (see trac ticket #20631):

```python
tsage: R.<zeta> = QQ[]
tsage: (zeta^2 + zeta + 1).factor_mod(7)
(zeta + 3) * (zeta + 5)
```

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

Return the **p**-adic factorization of this polynomial to the given precision.

**INPUT:**

• **p** - Prime number

• **prec** - Integer; the precision

**OUTPUT:**

• factorization of **self** viewed as a **p**-adic polynomial

**EXAMPLES:**

```python
tsage: R.<x> = QQ[]
tsage: f = x^3 - 2
tsage: f.factor_padic(2)
(1 + O(2^10))*x^3 + O(2^10)*x^2 + O(2^10)*x + 2 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6,…
˓→+ 2^7 + 2^8 + 2^9 + O(2^10)
tsage: f.factor_padic(3)
(1 + O(3^10))*x^3 + O(3^10)*x^2 + O(3^10)*x + 1 + 2*3 + 2*3^2 + 2*3^3 + 2*3^4,…
˓→+ 2*3^5 + 2*3^6 + 2*3^7 + 2*3^8 + 2*3^9 + O(3^10)
tsage: f.factor_padic(5)
((1 + O(5^10))*x + 2 + 4*5 + 2*5^2 + 2*5^3 + 5^4 + 3*5^5 + 4*5^7 + 2*5^8 + 5^…
˓→+ 9 + O(5^10)) * ((1 + O(5^10))*x^2 + (3 + 2*5^2 + 2*5^3 + 3*5^4 + 5^5 + 4*5^…
˓→+ 6 + 2*5^8 + 3*5^9 + O(5^10))*x + 4 + 5 + 2*5^2 + 4*5^3 + 4*5^4 + 3*5^5 +,…
˓→+ 3*5^6 + 4*5^7 + 4*5^9 + O(5^10))
```

The input polynomial is considered to have “infinite” precision, therefore the **p**-adic factorization of the polynomial is not the same as first coercing to **Q_p** and then factoring (see also trac ticket #15422):

```python
tsage: f = x^2 - 3^6
tsage: f.factor_padic(3,5)
((1 + O(3^5))*x + 3^3 + O(3^5)) * ((1 + O(3^5))*x + 2*3^3 + 2*3^4 + O(3^5))
tsage: f.change_ring(Qp(3,5)).factor()
Traceback (most recent call last):
...
PrecisionError: p-adic factorization not well-defined since the discriminant,…
˓→is zero up to the requestion p-adic precision
```

A more difficult example:
$$\textit{sage}: \ f = 100 \cdot (5x + 1)^2 \cdot (x + 5)^2$$
$$\textit{sage}: \ f.\text{factor_padic}(5, 10)$$
$$(4 \cdot 5^4 + O(5^{14})) \cdot ((1 + O(5^9)) \cdot x + 5^{-1} + O(5^9))^2 \cdot ((1 + O(5^{10})) \cdot x + 5^{-1} + O(5^{10}))^2$$

Try some bogus inputs:

$$\textit{sage}: \ f.\text{factor_padic}(3,-1)$$
Traceback (most recent call last):
...\nValueError: prec_cap must be non-negative.

$$\textit{sage}: \ f.\text{factor_padic}(6,10)$$
Traceback (most recent call last):
...\nValueError: p must be prime

$$\textit{sage}: \ f.\text{factor_padic}('hello', 'world')$$
Traceback (most recent call last):
...\nTypeError: unable to convert 'hello' to an integer

galois\_group(pari\_group=False, algorithm='pari')
Returns the Galois group of self as a permutation group.

INPUT:
- **self** - Irreducible polynomial
- **pari\_group** - bool (default: False); if True instead return the Galois group as a PARI group. This has a useful label in it, and may be slightly faster since it doesn’t require looking up a group in Gap. To get a permutation group from a PARI group $P$, type $\text{PermutationGroup}(P)$.
- **algorithm** - 'pari', 'kash', 'magma' (default: 'pari', except when the degree is at least 12 in which case 'kash' is tried).

OUTPUT:
- Galois group

ALGORITHM:
The Galois group is computed using PARI in C library mode, or possibly KASH or MAGMA.

Note: The PARI documentation contains the following warning: The method used is that of resolvent polynomials and is sensitive to the current precision. The precision is updated internally but, in very rare cases, a wrong result may be returned if the initial precision was not sufficient.

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

EXAMPLES:

```
sage: R.<x> = \mathbb{Q}[x]
sage: f = x^4 - 17x^3 - 2x + 1
sage: G = f.galois_group(); G
Transitive group number 5 of degree 4
sage: G.gens()
[(1,2), (1,2,3,4)]
sage: G.order()
24
```
It is potentially useful to instead obtain the corresponding PARI group, which is little more than a 4-tuple. See the PARI manual for the exact details. (Note that the third entry in the tuple is in the new standard ordering.)

```python
sage: f = x^4 - 17*x^3 - 2*x + 1
sage: G = f.galois_group(pari_group=True); G
PARI group [24, -1, 5, "S4"] of degree 4
sage: PermutationGroup(G)
Transitive group number 5 of degree 4
```

You can use KASH to compute Galois groups as well. The advantage is that KASH can compute Galois groups of fields up to degree 21, whereas PARI only goes to degree 11. (In my not-so-thorough experiments PARI is faster than KASH.)

```python
sage: f = x^4 - 17*x^3 - 2*x + 1
sage: f.galois_group(algorithm='kash')  # optional - kash
Transitive group number 5 of degree 4
sage: f = x^4 - 17*x^3 - 2*x + 1
sage: f.galois_group(algorithm='magma')  # optional - magma
Transitive group number 5 of degree 4
```

gcd(right)

Returns the (monic) greatest common divisor of self and right.

Corner cases: if self and right are both zero, returns zero. If only one of them is zero, returns the other polynomial, up to normalisation.

EXAMPLES:

```python
sage: R.<t> = QQ[]
sage: f = -2 + 3*t/2 + 4*t^2/7 - t^3
sage: g = 1/2 + 4*t + 2*t^4/3
sage: f.gcd(g)
1
sage: f = (-3*t + 1/2) * f
sage: g = (-3*t + 1/2) * (4*t^2/3 - 1) * g
sage: f.gcd(g)
t - 1/6
```

hensel_lift(p, e)

Assuming that this polynomial factors modulo \( p \) into distinct monic factors, computes the Hensel lifts of these factors modulo \( p^e \). We assume that self has integer coefficients.

Returns an empty list if this polynomial has degree less than one.

INPUT:

- \( p \) - Prime number; coercible to Integer
- \( e \) - Exponent; coercible to Integer

OUTPUT:

- Hensel lifts; list of polynomials over \( \mathbb{Z}/p^e \mathbb{Z} \)

EXAMPLES:

```python
sage: R.<x> = QQ[]
sage: R((x-1)*(x+1)).hensel_lift(7, 2)
[x + 1, x + 48]
```
If the input polynomial \( f \) is not monic, we get a factorization of \( f/lc(f) \):

```sage
R(2*x^2 - 2).hensel_lift(7, 2)
[x + 1, x + 48]
```

**inverse_series_trunc** \((\text{prec})\)

Return a polynomial approximation of precision \( \text{prec} \) of the inverse series of this polynomial.

**EXAMPLES:**

```sage
x = polygen(QQ)
p = 2 + x - 3/5*x^2
q5 = p.inverse_series_trunc(5)
q5
151/800*x^4 - 17/80*x^3 + 11/40*x^2 - 1/4*x + 1/2
q5 * p
-453/4000*x^6 + 253/800*x^5 + 1
q100 = p.inverse_series_trunc(100)
(q100 * p).truncate(100)
1
```

**is_irreducible**

Return whether this polynomial is irreducible.

This method computes the primitive part as an element of \( \mathbb{Z}[t] \) and calls the method **is_irreducible** for elements of that polynomial ring.

By definition, over any integral domain, an element \( r \) is irreducible if and only if it is non-zero, not a unit and whenever \( r = ab \) then \( a \) or \( b \) is a unit.

**EXAMPLES:**

```sage
R.<t> = QQ[]
(t^2 + 2).is_irreducible()
True
(t^2 - 1).is_irreducible()
False
```

**is_one**

Returns whether or not this polynomial is one.

**EXAMPLES:**

```sage
R.<x> = QQ[]
R([0,1]).is_one()
False
R([1]).is_one()
True
R([0]).is_one()
False
R([-1]).is_one()
False
R([1,1]).is_one()
False
```

**is_zero**

Returns whether or not self is the zero polynomial.

**EXAMPLES:**
```python
sage: R.<t> = QQ[]
sage: f = 1 - t + 1/2*t^2 - 1/3*t^3
sage: f.is_zero()
False
sage: R(0).is_zero()
True
```

**lcm(right)**

Returns the monic (or zero) least common multiple of self and right.

Corner cases: if either of self and right are zero, returns zero. This behaviour is ensures that the relation \(\text{lcm}(a,b) \text{gcd}(a,b) = a b\) holds up to multiplication by rationals.

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: f = -2 + 3*t/2 + 4*t^2/7 - t^3
sage: g = 1/2 + 4*t + 2*t^4/3
sage: f.lcm(g)
t^7 - 4/7*t^6 - 3/2*t^5 + 8*t^4 - 75/28*t^3 - 66/7*t^2 + 87/8*t + 3/2
sage: f.lcm(g) * f.gcd(g) // (f * g)
-3/2
```

**list(copy=True)**

Return a list with the coefficients of self.

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: f = 1 + t + t^2/2 + t^3/3 + t^4/4
sage: f.list()
[1, 1, 1/2, 1/3, 1/4]
sage: g = R(0)
sage: g.list()
[]
```

**numerator()**

Returns the numerator of self.

Representing self as the quotient of an integer polynomial and a positive integer denominator (coprime to the content of the polynomial), returns the integer polynomial.

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: f = (3 * t^3 + 1) / -3
sage: f.numerator()
-3*t^3 - 1
```

**quo_rem(right)**

Returns the quotient and remainder of the Euclidean division of self and right.

Raises a \(\text{ZerodivisionError}\) if right is zero.

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: g = R.random_element(1000)
```

(continues on next page)
sage: q, r = f.quo_rem(g)
sage: f == q*g + r
True

**real_root_intervals()**

Returns isolating intervals for the real roots of self.

**EXAMPLES:**

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

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

**resultant(right)**

Returns the resultant of self and right.

Enumerating the roots over \( \mathbb{Q} \) as \( r_1, \ldots, r_m \) and \( s_1, \ldots, s_n \) and letting \( x \) and \( y \) denote the leading coefficients of \( f \) and \( g \), the resultant of the two polynomials is defined by

\[
x^{\deg g} y^{\deg f} \prod_{i,j} (r_i - s_j).
\]

Corner cases: if one of the polynomials is zero, the resultant is zero. Note that otherwise if one of the polynomials is constant, the last term in the above is the empty product.

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: f = (t - 2/3) * (t + 4/5) * (t - 1)
sage: g = (t - 1/3) * (t + 1/2) * (t + 1)
sage: f.resultant(g)
119/1350
sage: h = (t - 1/3) * (t + 1/2) * (t - 1)
sage: f.resultant(h)
0
```

**reverse(degree=None)**

Reverse the coefficients of this polynomial (thought of as a polynomial of degree `degree`).

**INPUT:**

- `degree` (None or integral value that fits in an unsigned long, default: degree of self) - if specified, truncate or zero pad the list of coefficients to this degree before reversing it.

**EXAMPLES:**

We first consider the simplest case, where we reverse all coefficients of a polynomial and obtain a polynomial of the same degree:

```python
sage: R.<t> = QQ[]
sage: f = 1 + t + t^2 / 2 + t^3 / 3 + t^4 / 4
sage: f.reverse()
t^4 + t^3 + 1/2*t^2 + 1/3*t + 1/4
```

Next, an example where the returned polynomial has lower degree because the original polynomial has low coefficients equal to zero:
The next example illustrates the passing of a value for degree less than the length of self, notationally resulting in truncation prior to reversing:

\[
\begin{align*}
\text{sage: } & R.<t> = \mathbb{Q}[t] \\
\text{sage: } & f = 1 + t + t^2 / 2 + t^3 / 3 + t^4 / 4 \\
\text{sage: } & f.reverse(2) \\
& t^2 + t + 1/2
\end{align*}
\]

Now we illustrate the passing of a value for degree greater than the length of self, notationally resulting in zero padding at the top end prior to reversing:

\[
\begin{align*}
\text{sage: } & R.<t> = \mathbb{Q}[t] \\
\text{sage: } & f = 1 + t + t^2 / 2 + 1/2*t^2 + 1/3*t \\
\text{sage: } & f.reverse(4) \\
& t^4 + t^3 + 1/2*t^2 + 1/3*t
\end{align*}
\]

**revert_series** (*n*)

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

**EXAMPLES:**

\[
\begin{align*}
\text{sage: } & R.<t> = \mathbb{Q}[t] \\
\text{sage: } & f = t - t^3/6 + t^5/120 \\
\text{sage: } & f.reverse_series(6) \\
& 3/40*t^5 + 1/6*t^3 + t \\
\text{sage: } & f.reverse_series(-1) \\
& \text{ValueError: argument n must be a non-negative integer, got -1} \\
\text{sage: } & g = - t^3/3 + t^5/5 \\
\text{sage: } & g.reverse_series(6) \\
& \text{Traceback (most recent call last):} \\
& \text{ValueError: self must have constant coefficient 0 and a unit for coefficient } \_\text{-}t^1
\end{align*}
\]

**truncate** (*n*)

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

**INPUT:**

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

**EXAMPLES:**

\[
\begin{align*}
\text{sage: } & R.<t> = \mathbb{Q}[t] \\
\text{sage: } & f = 1 - t + 1/2*t^2 - 1/3*t^3 \\
\text{sage: } & f.truncate(0) \\
& 0 \\
\text{sage: } & f.truncate(2) \\
& -t + 1
\end{align*}
\]
**xgcd***(right)***

Returns polynomials d, s, and t such that d == s * self + t * right, where d is the (monic) greatest common divisor of self and right. The choice of s and t is not specified any further.

Corner cases: if self and right are zero, returns zero polynomials. Otherwise, if only self is zero, returns (d, s, t) = (right, 0, 1) up to normalisation, and similarly if only right is zero.

**EXAMPLES:**

```
sage: R.<t> = QQ[]
sage: f = 2/3 + 3/4 * t - t^2
sage: g = -3 + 1/7 * t
sage: f.xgcd(g)
(1, -12/5095, -84/5095*t - 1701/5095)
```

### 2.1.10 Dense univariate polynomials over \(\mathbb{Z}/n\mathbb{Z}\), implemented using FLINT.

This module gives a fast implementation of \((\mathbb{Z}/n\mathbb{Z})[x]\) whenever \(n\) is at most \(\text{sys.maxsize}\). We use it by default in preference to NTL when the modulus is small, falling back to NTL if the modulus is too large, as in the example below.

**EXAMPLES:**

```
sage: R.<a> = PolynomialRing(Integers(100))
sage: type(a)
<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 GF(2).
Note: Implementations using this template MUST implement coercion from base ring elements and get_unsafe(). See Polynomial_GF2X for an example.

degree()
EXAMPLES:

```python
sage: P.<x> = GF(2)[]
sage: x.degree()
1
sage: P(1).degree()
0
sage: P(0).degree()
-1
```

gcd(other)
Return the greatest common divisor of self and other.

EXAMPLES:

```python
sage: P.<x> = GF(2)[]
sage: f = x*(x+1)
sage: f.gcd(x+1)
x + 1
sage: f.gcd(x^2)
x
```

get_cparent()

is_gen()

EXAMPLES:

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

is_one()

EXAMPLES:

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

is_zero()

EXAMPLES:

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

list(copy=True)

EXAMPLES:

```python
sage: P.<x> = GF(2)[]
sage: x.list()
(continues on next page)```
EXAMPLES:

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

### shift(n)

EXAMPLES:

```python
sage: P.<x> = GF(2)[]
sage: f = x^3 + x^2 + 1
sage: f.shift(1)
x^4 + x^3 + x
sage: f.shift(-1)
x^2 + x
```

### truncate(n)

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

EXAMPLES:

```python
sage: R.<x> =GF(2)[]
sage: f = sum(x^n for n in range(10)); f
x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
sage: f.truncate(6)
x^5 + x^4 + x^3 + x^2 + x + 1
```

If the precision is higher than the degree of the polynomial then the polynomial itself is returned:

```python
sage: f.truncate(10) is f
True
```

### xgcd(other)

Computes extended gcd of self and other.

EXAMPLES:

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

class sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint

Bases: sage.rings.polynomial.polynomial_zmod_flint.Polynomial_template

Polynomial on \(\mathbb{Z}/n\mathbb{Z}\) implemented via FLINT.

```python
_add_(right)

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

_sub__ (right)
EXAMPLES:

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

_lmul__ (left)
EXAMPLES:

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

sage: R.<y> = GF(5)[]
sage: u = y^2 + y + 1
sage: 3*u
3*y^2 + 3*y + 3
sage: 5*u
0
sage: (2^81)*u
2*y^2 + 2*y + 2
sage: (-2^81)*u
3*y^2 + 3*y + 3

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

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

```
```
_rmul__ (right)
Multiply self on the right by a scalar.

EXAMPLES:

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

EXAMPLES:

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

_mul_trunc_ (right, n)

Return the product of this polynomial and other truncated to the given length n.

This function is usually more efficient than simply doing the multiplication and then truncating. The function is tuned for length n about half the length of a full product.

EXAMPLES:

```
sage: P.<a>=GF(7)[]
sage: a = P(range(10)); b = P(range(5, 15))
sage: a._mul_trunc_(b, 5)
4*a^4 + 6*a^3 + 2*a^2 + 5*a
```

factor ()

Returns the factorization of the polynomial.

EXAMPLES:

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

is_irreducible ()

Return whether this polynomial is irreducible.

EXAMPLES:

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

Not implemented when the base ring is not a field:

```
sage: S.<s> = Zmod(10)[]
sage: (s^2).is_irreducible() 
Traceback (most recent call last):
... 
NotImplementedError: checking irreducibility of polynomials over rings with composite characteristic is not implemented
```

monic ()

Return this polynomial divided by its leading coefficient.

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

EXAMPLES:

```
sage: R.<x> = GF(5)[]
sage: (2*x^2+1).monic() 
x^2 + 3
```
rational_reconstruct \((m, n\_deg=0, d\_deg=0)\)

Construct a rational function \(n/d\) such that \(p \cdot d\) is equivalent to \(n\) modulo \(m\) where \(p\) is this polynomial.

EXAMPLES:

```python
sage: P.<x> = GF(5) []
sage: p = 4*x^5 + 3*x^4 + 2*x^3 + 2*x^2 + 4*x + 2
sage: n, d = p.rational_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:

```python
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 trac ticket #11782 has been fixed:

```python
sage: R.<x> = ZZ.quo(9)['x']
sage: f = 2*x^3 + x^2 + x; g = 6*x^2 + 2*x + 1
sage: f.resultant(g)
5
```

reverse \((degree=None)\)

Return a polynomial with the coefficients of this polynomial reversed.

If an optional degree argument is given the coefficient list will be truncated or zero padded as necessary before computing the reverse.

EXAMPLES:

```python
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
```

(continues on next page)
Note that if \( f \) has zero constant coefficient, its reverse will have lower degree.

In this case, reverse is not an involution unless we explicitly specify a degree.

\[
\text{sage: } f = x^3 + 2 \times x \\
\text{sage: } f.\text{reverse()} \\
2 \times x^2 + 1
\]

**revert_series** \((n)\)

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

**EXAMPLES:**

\[
\text{sage: } R, t = \text{GF}(5)[] \\
\text{sage: } f = t + 2 \times t^2 - t^3 - 3 \times t^4 \\
\text{sage: } f.\text{revert_series}(5) \\
3 \times t^4 + 4 \times t^3 + 3 \times t^2 + t
\]

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

\[
\text{sage: } g = - t^3 + t^5 \\
\text{sage: } g.\text{revert_series}(6) \\
\text{Traceback (most recent call last):} \\
\text{...} \\
\text{ValueError: self must have constant coefficient 0 and a unit for coefficient } t \\
\text{...}
\]

\[
\text{sage: } g = t + 2 \times t^2 - t^3 - 3 \times t^4 + t^5 \\
\text{sage: } g.\text{revert_series}(6) \\
\text{Traceback (most recent call last):} \\
\text{...} \\
\text{ValueError: the integers 1 up to n=5 are required to be invertible over the } \\
\text{base field}
\]

**small_roots** \((*\text{args}, **\text{kwds})\)

See `sage.rings.polynomial.polynomial_modn_dense_ntl.small_roots()` for the documentation of this function.

**EXAMPLES:**

\[
\text{sage: } N = 10001 \\
\text{sage: } K = \text{Zmod}(10001) \\
\text{sage: } P, x = \text{PolynomialRing}(K) \\
\text{sage: } f = x^3 + 10 \times x^2 + 5000 \times x - 222
\]
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

sage.rings.polynomial.polynomial_zmod_flint.make_element(parent, args)

2.1.11 Dense univariate polynomials over $\mathbb{Z}/n\mathbb{Z}$, implemented using NTL.

This implementation is generally slower than the FLINT implementation in polynomial_zmod_flint, so we use FLINT by default when the modulus is small enough; but NTL does not require that $n$ be int-sized, so we use it as default when $n$ is too large for FLINT.

Note that the classes Polynomial_dense_modn_ntl_zz and Polynomial_dense_modn_ntl_ZZ are different; the former is limited to moduli less than a certain bound, while the latter supports arbitrarily large moduli.

AUTHORS:

• Robert Bradshaw: Split off from polynomial_element_generic.py (2007-09)
• Robert Bradshaw: Major rewrite to use NTL directly (2007-09)

class sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n

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({10:-1})
7*x^10
**degree** *(gen=None)*

Return the degree of this polynomial.

The zero polynomial has degree -1.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(Integers(100), implementation='NTL')
sage: (x^3 + 3*x - 17).degree() 3
sage: R.zero().degree() -1
```

**int_list()**

**list** *(copy=True)*

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

**EXAMPLES:**

```python
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:**

```python
sage: R.<x> = PolynomialRing(Integers(100), implementation='NTL')
sage: from sage.rings.polynomial.polynomial_modn_dense_ntl import Polynomial_˓→dense_mod_n as poly_modn_dense
sage: poly_modn_dense(R, ([1,-2,3]))
3*x^2 + 98*x + 1
sage: f = poly_modn_dense(R, 0)
sage: fntl_set_directly([1,-2,3])
sage: f
3*x^2 + 98*x + 1
sage: fntl_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:**

```python
sage: R.<x> = PolynomialRing(Integers(12345678901234567890), implementation='NTL')
sage: p = x^2 + 2*x + 4
sage: p.shift(0)
x^2 + 2*x + 4
sage: p.shift(-1)
x + 2
sage: p.shift(-5)
0
sage: p.shift(2)
x^4 + 2*x^3 + 4*x^2
```

**AUTHOR:**

- David Harvey (2006-08-06)

**small_roots** (*args, **kwds*)

See `sage.rings.polynomial.polynomial_modn_dense_ntl.small_roots()` for the documentation of this function.

**EXAMPLES:**

```python
sage: N = 10001
sage: K = Zmod(10001)
sage: P.<x> = PolynomialRing(K, implementation='NTL')
sage: f = x^3 + 10*x^2 + 5000*x - 222
sage: f.small_roots()
[4]
```

**class** `sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_p`

**Bases:**

`sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n`

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

**discriminant** ()

**EXAMPLES:**

```python
sage: _.<x> = PolynomialRing(GF(19), implementation='NTL')
sage: f = x^3 + 3*x - 17
sage: f.discriminant()
12
```

**gcd** (*right*)

Return the greatest common divisor of this polynomial and other, as a monic polynomial.

**INPUT:**

- other – a polynomial defined over the same ring as self

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(GF(3), implementation="NTL")
sage: f, g = x + 2, x^2 - 1
```
resultant (other)
Returns the resultant of self and other, which must lie in the same polynomial ring.

INPUT:
• other – a polynomial

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

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(GF(19), implementation='NTL')
sage: f = x^3 + x + 1; g = x^3 - x - 1
sage: r = f.resultant(g); r
11
sage: r.parent() is GF(19)
True
```

xgcd (other)
Compute the extended gcd of this element and other.

INPUT:
• other – an element in the same polynomial ring

OUTPUT:
A tuple (r,s,t) of elements in the polynomial ring such that r = s*self + t*other.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(GF(3), implementation='NTL')
sage: x.xgcd(x)
(x, 0, 1)
```

```python
sage: (x^2 - 1).xgcd(x - 1)
(x + 2, 0, 1)
```

```python
sage: R.zero().xgcd(R.one())
(1, 0, 1)
```

```python
sage: (x^3 - 1).xgcd((x - 1)^2)
(x^2 + x + 1, 0, 1)
```

```python
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_modn

degree()

EXAMPLES:

```python
def degree():
```

```python
sage: R.<x> = PolynomialRing(Integers(14^34), implementation='NTL')
sage: f = x^4 - x - 1
sage: f.degree()
4
```

```python
sage: f = 14^43*x + 1
sage: f.degree()
0
```
is_gen()
list(copy=True)
quo_rem(right)
Returns \(q\) and \(r\), with the degree of \(r\) less than the degree of \(right\), such that \(q \ast right + r = self\).

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(Integers(10^30), implementation='NTL')
sage: f = x^5+1; g = (x+1)^2
sage: q, r = f.quo_rem(g)
sage: q
x^3 + 999999999999999999999999999996
˓→ 999999999999999999999999999996
sage: r
5*x + 5
sage: q\ast g + r
x^5 + 1
```

reverse(degree=None)
Return the reverse of the input polynomial thought as a polynomial of degree `degree`.

If \(f\) is a degree-\(d\) polynomial, its reverse is \(x^d f(1/x)\).

INPUT:

- `degree` (None or an integer) - if specified, truncate or zero pad the list of coefficients to this degree before reversing it.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(Integers(12^29), implementation='NTL')
sage: f = x^4 + 2*x + 5
sage: f.reverse()
5*x^4 + 2*x^3 + 1
sage: f.reverse(1)
1
sage: f.reverse(5)
x^4 + x^2
```

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

EXAMPLES:

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

truncate(n)
Returns this polynomial mod \(x^n\).

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

### valuation()

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

#### EXAMPLES:

```python
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
```

```python
class sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_zz
Bases: sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n
Polynomial on \( \mathbb{Z}/n\mathbb{Z} \) implemented via NTL.

_add_(\_right)

_sub_(\_right)

_lmul_(\_c)

_rmul_(\_c)

_mul_(\_right)

_mul_trunc_(\_right, \_n)

Return the product of self and right truncated to the given length \( n \)

#### EXAMPLES:

```python
sage: R.<x> = PolynomialRing(Integers(100), implementation="NTL")
sage: f = x - 2
sage: g = x^2 - 8*x + 16
sage: f*g
x^3 + 90*x^2 + 32*x + 68
sage: f._mul_trunc_(g, 42)
x^3 + 90*x^2 + 32*x + 68
sage: f._mul_trunc_(g, 3)
90*x^2 + 32*x + 68
sage: f._mul_trunc_(g, 2)
32*x + 68
sage: f._mul_trunc_(g, 1)
68
sage: f._mul_trunc_(g, 0)
0
sage: f = x^2 - 8*x + 16
sage: f._mul_trunc_(f, 2)
44*x + 56
```
```

2.1. Univariate Polynomials and Polynomial Rings  159
**degree()**

EXAMPLES:

```python
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:

```python
sage: R.<x> = PolynomialRing(Integers(100), implementation='NTL')
sage: from sage.rings.polynomial.polynomial_modn_dense_ntl import Polynomial_modn_dense
sage: f = poly_modn_dense(R,[5,0,0,1])
sage: f.int_list()
[5, 0, 0, 1]
```

**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 \( q \ast right + r = self \).

EXAMPLES:

```python
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
```

**reverse(degree=None)**

Return the reverse of the input polynomial thought as a polynomial of degree \( degree \).

If \( f \) is a degree-\( d \) polynomial, its reverse is \( x^d f(1/x) \).

INPUT:

- `degree` (None or an integer) - if specified, truncate or zero pad the list of coefficients to this degree before reversing it.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(Integers(77), implementation='NTL')
sage: f = x^4 - x - 1
sage: f.reverse()
76*x^4 + 76*x^3 + 1
```
\texttt{sage: f.reverse(2)}
\begin{verbatim}
76*x^2 + 76*x
\end{verbatim}
\texttt{sage: f.reverse(5)}
\begin{verbatim}
76*x^5 + 76*x^4 + x
\end{verbatim}
\texttt{sage: g = x^3 - x}
\texttt{sage: g.reverse()}
\begin{verbatim}
76*x^2 + 1
\end{verbatim}
\textbf{shift} \((n)\)
Shift self to left by \(n\), which is multiplication by \(x^n\), truncating if \(n\) is negative.

\textbf{EXAMPLES:}
\begin{verbatim}
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
\end{verbatim}
\textbf{truncate} \((n)\)
Returns this polynomial mod \(x^n\).

\textbf{EXAMPLES:}
\begin{verbatim}
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
\end{verbatim}
\textbf{valuation}()
Returns the valuation of self, that is, the power of the lowest non-zero monomial of self.

\textbf{EXAMPLES:}
\begin{verbatim}
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
\end{verbatim}
\texttt{sage.rings.polynomial.polynomial_modn_dense_ntl.make_element}(parent, args)
\texttt{sage.rings.polynomial.polynomial_modn_dense_ntl.small_roots}(self, \(X=\text{None}, \) \(\beta=1.0, \) \(\epsilon=\text{None}, \)**kwds)

Let \(N\) be the characteristic of the base ring this polynomial is defined over: \(N = \text{self.base_ring().characteristic()}\). This method returns small roots of this polynomial modulo some factor \(b\) of \(N\) with the constraint that \(b >= N^\beta\). Small in this context means that if \(x\) is a root of \(f\) modulo \(b\) then \(|x| < X\). This \(X\) is either provided by the user or the maximum \(X\) is chosen such that this algorithm terminates in polynomial
time. If $X$ is chosen automatically it is $X = \text{ceil}(1/2N^{\beta^2/\delta-t})$. 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 $\epsilon$ described above. (Default: $\beta/8$)
- **kwds – passed through to method Matrix_integer_dense.LLL().**

**EXAMPLES:**

First consider a small example:

```python
sage: N = 10001
sage: K = Zmod(10001)
sage: P.<x> = PolynomialRing(K, implementation='NTL')
sage: f = x^3 + 10*x^2 + 5000*x - 222

This polynomial has no roots without modular reduction (i.e. over $\mathbb{Z}$):

```python
sage: f.change_ring(ZZ).roots()
[]
```

To compute its roots we need to factor the modulus $N$ and use the Chinese remainder theorem:

```python
sage: p,q = N.prime_divisors()
sage: f.change_ring(GF(p)).roots()
[(4, 1)]
sage: f.change_ring(GF(q)).roots()
[(4, 1)]
sage: crt(4, 4, p, q)
4
```

This root is quite small compared to $N$, so we can attempt to recover it without factoring $N$ using Coppersmith’s small root method:

```python
sage: f.small_roots()
[4]
```

An application of this method is to consider RSA. We are using 512-bit RSA with public exponent $e = 3$ to encrypt a 56-bit DES key. Because it would be easy to attack this setting if no padding was used we pad the key $K$ with 1s to get a large number:

```python
sage: Nbits, Kbits = 512, 56
sage: e = 3
```

We choose two primes of size 256-bit each:

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

We choose a random key:
```
sage: K = ZZ.random_element(0, 2^Kbits)

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

```sage:

```
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:

```

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

```sage:

```

and recover its small roots:

```sage:

```

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:

```

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

```sage:

```

And try to recover $q$ from it:

```sage:

```

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


2.1.12 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

change_ring \((R)\)

EXAMPLES:

```python
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RR['x'], [-2, 0, 1.5])
sage: f.change_ring(QQ)
3/2*x^2 - 2
sage: f.change_ring(RealField(10))
1.5*x^2 - 2.0
sage: f.change_ring(RealField(100))
1.5000000000000000000000000000*x^2 - 2.0000000000000000000000000000
```

degree()

Return the degree of the polynomial.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RR['x'], [1, 2, 3]); f
-2.00000000000000*x^2 + 1.00000000000000
sage: f.list()
[1.00000000000000, 0.000000000000000, -2.00000000000000]
```

```python
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RR['x'], [3, pi, 1])
sage: f.integral()
0.333333333333333*x^3 + 1.57079632679490*x^2 + 3.00000000000000*x
```

list(copy=True)

EXAMPLES:

```python
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RR['x'], [1, 0, -2]); f
-2.00000000000000*x^2 + 1.000000000000000
sage: f.list()
[1.00000000000000, 0.000000000000000, -2.00000000000000]
```
\texttt{quo\_rem}(\texttt{other})

Return the quotient with remainder of \texttt{self} by \texttt{other}.

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

\texttt{reverse}(\texttt{degree=\texttt{None}})

Return reverse of the input polynomial thought as a polynomial of degree \texttt{degree}.

If \texttt{f} is a degree-\texttt{d} polynomial, its reverse is \(x^d f(1/x)\).

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{degree (\texttt{None} or an integer) - if specified, truncate or zero pad the list of coefficients to this degree before reversing it.}
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: f = RR['x']([-3, pi, 0, 1])
sage: f.reverse()
-3.00000000000000*x^3 + 3.14159265358979*x^2 + 1.00000000000000
sage: f.reverse(2)
-3.00000000000000*x^2 + 3.14159265358979*x
sage: f.reverse(5)
-3.00000000000000*x^5 + 3.14159265358979*x^4 + x^2
\end{verbatim}

\texttt{shift}(\texttt{n})

Returns this polynomial multiplied by the power \(x^n\). If \texttt{n} is negative, terms below \(x^n\) will be discarded. Does not change this polynomial.

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

(continues on next page)
**truncation**

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

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RealField(10)['x'], [1, 2, 4, 8])
sage: f
1.0e-9*x^9 + 1.0e-8*x^8 + 1.0e-7*x^7 + 1.0e-6*x^6 + 0.000010*x^5 + 0.000010*x^4 + 0.000010*x^3 + 0.010*x^2 + 0.10*x + 1.0
sage: f.truncate_abs(0.5e-6)
1.0e-6*x^6 + 0.000010*x^5 + 0.000010*x^4 + 0.000010*x^3 + 0.010*x^2 + 0.10*x + 1.0
```

**truncation_abs**

Truncate all high order coefficients below bound.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RealField(10)['x'], [1, 2, 3])
sage: f
3.00000000000000*x + 2.00000000000000
sage: f.truncate(-10)
0
```

`sage.rings.polynomial.polynomial_real_mpfr_dense.make_PolynomialRealDense(parent, data)`

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import make_PolynomialRealDense
sage: make_PolynomialRealDense(RR['x'], [1, 2, 3])
3.00000000000000*x^2 + 2.00000000000000*x + 1.00000000000000
```

### 2.1.13 Polynomial Interfaces to Singular

**AUTHORS:**

- Martin Albrecht <malb@informatik.uni-bremen.de> (2006-04-21)
- Robert Bradshaw: Re-factor to avoid multiple inheritance vs. Cython (2007-09)
• Syed Ahmad Lavasani: Added function field to _singular_init_ (2011-12-16)  Added non-prime finite fields to _singular_init_ (2012-1-22)

```python
class sage.rings.polynomial.polynomial_singular_interface.PolynomialRing_singular_repr
Bases: object

Implements methods to convert polynomial rings to Singular.
This class is a base class for all univariate and multivariate polynomial rings which support conversion from and to Singular rings.

class sage.rings.polynomial.polynomial_singular_interface.Polynomial_singular_repr
Bases: object

Implements coercion of polynomials to Singular polynomials.
This class is a base class for all (univariate and multivariate) polynomial classes which support conversion from and to Singular polynomials.
Due to the incompatibility of Python extension classes and multiple inheritance, this just defers to module-level functions.
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:

```python
sage: from sage.rings.polynomial.polynomial_singular_interface import can_convert_to_singular
sage: can_convert_to_singular(PolynomialRing(QQ, names=['x']))
True
sage: can_convert_to_singular(PolynomialRing(ZZ, names=['x']))
True
sage: can_convert_to_singular(PolynomialRing(QQ, names=[]))
False
```

### 2.1.14 Base class for generic $p$-adic polynomials

This provides common functionality for all $p$-adic polynomials, such as printing and factoring.

AUTHORS:

• Jeroen Demeyer (2013-11-22): initial version, split off from other files, made Polynomial_padic the common base class for all p-adic polynomials.

```python
class sage.rings.polynomial.padics.polynomial_padic.Polynomial_padic(parent, x=None, check=True, is_gen=False, construct=False)
Bases: sage.rings.polynomial.polynomial_element.Polynomial

c.content()
    Compute the content of this polynomial.

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

EXAMPLES:

```
sage: K = Zp(13,7)
sage: R.<t> = K[]
sage: f = 13^7*t^3 + K(169,4)*t - 13^4
sage: f.content()
13^2 + O(13^9)
sage: R(0).content()
0
sage: f = R(K(0,3)); f
0(13^3)
sage: f.content()
0(13^3)
sage: P.<x> = ZZ[]
sage: f = x + 2
1
sage: fp = f.change_ring(pAdicRing(2, 10))
sage: fp
(1 + O(2^10))*x + 2 + O(2^11)
sage: fp.content()
1 + O(2^10)
sage: (2*fp).content()
2 + O(2^11)
```

Over a field it would be sufficient to return only zero or one, as the content is only defined up to multiplication with a unit. However, we return $\pi^k$ where $k$ is the minimal valuation of any coefficient:

```
sage: K = Qp(13,7)
sage: R.<t> = K[]
sage: f = 13^7*t^3 + K(169,4)*t - 13^-4
sage: f.content()
13^-4 + O(13^3)
sage: f = R.zero()
sage: f.content()
0(13^3)
sage: f = 13*t^3 + K(0,1)*t
sage: f.content()
13 + O(13^8)
```

**factor()**

Return the factorization of this polynomial.

EXAMPLES:

```
sage: R.<t> = PolynomialRing(Qp(3,3,print_mode='terse',print_pos=False))
sage: pol = t^8 - 1
sage: for p,e in pol.factor():
....:     print("{} {}".format(e, p))
1 (1 + O(3^3))*t + 1 + O(3^3)
```

(continues on next page)
The same factorization over \( \mathbb{Z}_p \). In this case, the “unit” part is a \( p \)-adic unit and the power of \( p \) is considered to be a factor:

```python
sage: R.<t> = PolynomialRing(Zp(5,6,print_mode='terse',print_pos=False))
sage: pol = 100 * (5*t - 1) * (t - 5)
sage: pol.factor().value()
(500 + O(5^8))*t^2 + (-2600 + O(5^8))*t + 500 + O(5^8)
```

In the following example, the discriminant is zero, so the \( p \)-adic factorization is not well defined:

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

An example of factoring a constant polynomial (see trac ticket #26669):

```python
sage: R.<x> = Qp(5)[]
sage: R(2).factor()
2 + O(5^20)
```

More examples over \( \mathbb{Z}_p \):

```python
sage: R.<w> = PolynomialRing(Zp(5, prec=6, type = 'capped-abs', print_mode = 'val-unit'))
sage: f = w^5-1
sage: f.factor()
((1 + O(5^6))*w + 3124 + O(5^6)) * ((1 + O(5^6))*w^4 + (12501 + O(5^6))*w^3 + (9376 + O(5^6))*w^2 + (6251 + O(5^6))*w + 3126 + O(5^6))
```

See trac ticket #4038:

```python
sage: E = EllipticCurve('37a1')
sage: K = Qp(7,10)
sage: EK = E.base_extend(K)
sage: E = EllipticCurve('37a1')
```

(continues on next page)
```python
sage: K = Qp(7,10)
sage: EK = E.base_extend(K)
sage: g = EK.division_polynomial_0(3)
sage: g.factor()
(3 + O(7^10)) * ((1 + O(7^10))*x + 1 + 2*7 + 4*7^2 + 2*7^3 + 5*7^4 + 7^5 +
˓→ 5*7^6 + 3*7^7 + 5*7^8 + 3*7^9 + O(7^10)) * ((1 + O(7^10))*x^3 + (6 + 4*7 +
˓→ 2*7^2 + 4*7^3 + 7^4 + 5*7^5 + 7^6 + 3*7^7 + 7^8 + 3*7^9 + O(7^10))*x^2 + (6 +
˓→ 3*7 + 5*7^2 + 2*7^3 + 7^4 + 7^6 + 2*7^8 + 3*7^9 + O(7^10))*x + 2 + 5*7 +
˓→ 4*7^2 + 2*7^3 + 6*7^4 + 3*7^5 + 7^6 + 4*7^7 + O(7^10))
```

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

Return the p-adic extension field generated by the roots of the irreducible polynomial self.

INPUT:

- `names` – name of the generator of the extension
- `check_irreducible` – check whether the polynomial is irreducible
- `kwds` – see `sage.ring.padics.padic_generic.pAdicGeneric.extension()`

EXAMPLES:

```python
sage: R.<x> = Qp(3,5,print_mode='digits')
sage: f = x^2 - 3
sage: f.root_field('x')
3-adic Eisenstein Extension Field in x defined by x^2 - 3
```

```python
sage: R.<x> = Qp(5,5,print_mode='digits')
sage: f = x^2 - 3
sage: f.root_field('x', print_mode='bars')
5-adic Unramified Extension Field in x defined by x^2 - 3
```

```python
sage: R.<x> = Qp(11,5,print_mode='digits')
sage: f = x^2 - 3
sage: f.root_field('x', print_mode='bars')
Traceback (most recent call last):
...
ValueError: polynomial must be irreducible
```

### 2.1.15 p-adic Capped Relative Dense Polynomials

```python
sage: sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense
```

Bases:

- `sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdv`
- `sage.rings.polynomial.padics.polynomial_padic.Polynomial_padic`
- `sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense`
**degree** *(secure=False)*

Return the degree of self.

**INPUT:**

- secure – a boolean (default: False)

If secure is True and the degree of this polynomial is not determined (because the leading coefficient is indistinguishable from 0), an error is raised.

If secure is False, the returned value is the largest $n$ so that the coefficient of $x^n$ does not compare equal to 0.

**EXAMPLES:**

```python
sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.degree()
1
sage: (f-T).degree()
0
sage: (f-T).degree(secure=True)
Traceback (most recent call last):
  ...  
PrecisionError: the leading coefficient is indistinguishable from 0

sage: x = O(3^5)
sage: li = [3^i * x for i in range(0,5)]; li
[O(3^5), O(3^6), O(3^7), O(3^8), O(3^9)]
sage: f = R(li); f
0(3^9)*T^4 + O(3^8)*T^3 + O(3^7)*T^2 + O(3^6)*T + O(3^5)
sage: f.degree()
-1
sage: f.degree(secure=True)
Traceback (most recent call last):
  ...  
PrecisionError: the leading coefficient is indistinguishable from 0
```

**disc()**

**factor_mod()**

Return the factorization of self modulo $p$.

**is_eisenstein** *(secure=False)*

Return True if this polynomial is an Eisenstein polynomial.

**AUTHOR:**

- Xavier Caruso (2013-03)

**lift()**

Return an integer polynomial congruent to this one modulo the precision of each coefficient.
Note: The lift that is returned will not necessarily be the same for polynomials with the same coefficients (i.e. same values and precisions): it will depend on how the polynomials are created.

EXAMPLES:

```sage
sage: K = Qp(13,7)
sage: R.<t> = K[]
sage: a = 13^7*t^3 + K(169,4)*t - 13^4
sage: a.lift()
62748517*t^3 + 169*t - 28561
```

`list (copy=True)`

Return a list of coefficients of `self`.

Note: The length of the list returned may be greater than expected since it includes any leading zeros that have finite absolute precision.

EXAMPLES:

```sage
sage: K = Qp(13,7)
sage: R.<t> = K[]
sage: a = 2*t^3 + 169*t - 1
sage: a
(2 + O(13^7))*t^3 + (13^2 + O(13^9))*t + 12 + 12*13 + 12*13^2 + 12*13^3 + ...
\rightarrow 12*13^4 + 12*13^5 + 12*13^6 + O(13^7)
sage: a.list()
[12 + 12*13 + 12*13^2 + 12*13^3 + 12*13^4 + 12*13^5 + 12*13^6 + O(13^7),
13^2 + O(13^9),
0,
2 + O(13^7)]
```

`lshift_coeffs (shift, no_list=False)`

Return a new polynomials whose coefficients are multiplied by $p^\text{shift}$.

EXAMPLES:

```sage
sage: K = Qp(13, 4)
sage: R.<t> = K[]
sage: a = t + 52
sage: a.lshift_coeffs(3)
(13^3 + O(13^7))*t + 4*13^4 + O(13^8)
```

`newton_polygon()`

Return the Newton polygon of this polynomial.

Note: If some coefficients have not enough precision an error is raised.

OUTPUT:

- a Newton polygon

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

```
sage: f = x^4 + 2^3*x^3 + 2^13*x^2 + 2^21*x + 2^37
sage: f.newton_polygon()
Finite Newton polygon with 4 vertices: (0, 37), (1, 21), (3, 3), (4, 0)
```

```
sage: g = f + K(0,0)*t^4; g
(5^2 + O(5^22))*t^10 + O(5^0)*t^4 + (3 + O(5^20))*t + 5 + O(5^21)
```

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

**newton_slopes** *(repetition=True)*

Return a list of the Newton slopes of this polynomial.

These are the valuations of the roots of this polynomial.

If repetition is True, each slope is repeated a number of times equal to its multiplicity. Otherwise it appears only one time.

**INPUT:**

- repetition – boolean (default True)

**OUTPUT:**

- a list of rationals

**EXAMPLES:**

```
sage: K = Qp(5)
sage: R.<t> = K[]
sage: f = 5 + 3*t + t^4 + 25*t^10
sage: f.newton_polygon()
Finite Newton polygon with 4 vertices: (0, 1), (1, 0), (4, 0), (10, 2)
```

```
sage: f.newton_slopes()
[1, 0, -1/3]
sage: f.newton_slopes(repetition=False)
[1, 0, -1/3]
```

**AUTHOR:**

- Xavier Caruso (2013-03-20)

**prec_degree**

Return the largest \( n \) so that precision information is stored about the coefficient of \( x^n \).
Always greater than or equal to degree.

EXAMPLES:

```python
sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.prec_degree()
1
```

`precision_absolute(n=\text{None})`

Return absolute precision information about `self`.

INPUT:

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

OUTPUT:

If `n == \text{None}`, returns a list of absolute precisions of coefficients. Otherwise, returns the absolute precision of the coefficient of `x^n`.

EXAMPLES:

```python
sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.precision_absolute()
[10, 10]
```

`precision_relative(n=\text{None})`

Return relative precision information about `self`.

INPUT:

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

OUTPUT:

If `n == \text{None}`, returns a list of relative precisions of coefficients. Otherwise, returns the relative precision of the coefficient of `x^n`.

EXAMPLES:

```python
sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.precision_relative()
[10, 10]
```

`quo_rem(right, secure=False)`

Return the quotient and remainder in division of `self` by `right`.

EXAMPLES:
Sage: K = Qp(3,10)
Sage: R.<T> = K[
Sage: f = T + 2
Sage: g = T^4 + 3*T+22
Sage: g.quo_rem(f)
((1 + O(3^10))*T^3 + (1 + 2*3 + 2*3^2 + 2*3^3 + 2*3^4 + 2*3^5 + 2*3^6 + 2*3^7 → + 2*3^8 + 2*3^9 + O(3^10))*T^2 + (1 + 3 + O(3^10))*T + 1 + 3 + 2*3^2 + 2*3^3 → + 2*3^4 + 2*3^5 + 2*3^6 + 2*3^7 + 2*3^8 + 2*3^9 + O(3^10),
2 + 3 + 3^3 + O(3^10))

rescale(a)

Return f(a*X)

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

EXAMPLES:

sage: K = Zp(13, 5)
Sage: R.<t> = K[
Sage: f = t^3 + K(13, 3) * t
Sage: f.rescale(2) # not implemented

reverse(degree=None)

Return the reverse of the input polynomial, thought as a polynomial of degree degree.

If f is a degree-d polynomial, its reverse is x^df(1/x).

INPUT:

• degree (None or an integer) - if specified, truncate or zero pad the list of coefficients to this degree before reversing it.

EXAMPLES:

sage: K = Qp(13,7)
Sage: R.<t> = K[
Sage: f = t^3 + 4*t; f
(1 + O(13^7))*t^3 + (4 + O(13^7))*t
Sage: f.reverse()
0*t^3 + (4 + O(13^7))*t^2 + 1 + O(13^7)
Sage: f.reverse(3)
0*t^3 + (4 + O(13^7))*t^2 + 1 + O(13^7)
Sage: f.reverse(2)
0*t^2 + (4 + O(13^7))*t
Sage: f.reverse(4)
0*t^4 + (4 + O(13^7))*t^3 + (1 + O(13^7))*t
Sage: f.reverse(6)
0*t^6 + (4 + O(13^7))*t^5 + (1 + O(13^7))*t^3

rshift_coeffs(shift, no_list=False)

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

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

EXAMPLES:
sage: K = Zp(13, 4)
sage: R.<t> = K[]
sage: a = t^2 + K(13,3)*t + 169;
a
(1 + O(13^4))*t^2 + (13 + O(13^3))*t + 13 + O(13^6)
sage: b = a.rshift_coeffs(1); b
0(13^3)*t^2 + (1 + O(13^2))*t + 13 + O(13^5)
sage: b.list()
[13 + O(13^5), 1 + O(13^2), O(13^3)]
sage: b = a.rshift_coeffs(2); b
O(13^2)*t^2 + O(13)*t + 1 + O(13^4)
sage: b.list()
[1 + O(13^4), O(13), O(13^2)]

valuation (val_of_var=None)

Return the valuation of self.

INPUT:

self -- a p-adic polynomial
val_of_var -- None or a rational (default None).

OUTPUT:

If val_of_var == None, returns the largest power of the variable dividing self. Otherwise, returns the valuation of self where the variable is assigned valuation val_of_var

EXAMPLES:

sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.valuation()
0

valuation_of_coefficient (n=None)

Return valuation information about self’s coefficients.

INPUT:

self -- a p-adic polynomial
n -- None or an integer (default None).

OUTPUT:

If n == None, returns a list of valuations of coefficients. Otherwise, returns the valuation of the coefficient of x^n.

EXAMPLES:

sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.valuation_of_coefficient(1)
0

xgcd (right)

Extended gcd of self and other.
INPUT:
• \texttt{other} – an element with the same parent as \texttt{self}

OUTPUT:
Polynomials \(g, u, \text{ and } v\) such that \(g = u*\text{self} + v*\text{other}\)

**Warning:** The computations are performed using the standard Euclidean algorithm which might produce mathematically incorrect results in some cases. See trac ticket \#13439.

**EXAMPLES:**

```python
sage: R.<x> = Qp(3,3)[]
sage: f = x + 1
sage: f.xgcd(f^2)
((1 + O(3^3))*x + 1 + O(3^3), 1 + O(3^3), 0)
```

In these examples the results are incorrect, see trac ticket \#13439:

```python
sage: R.<x> = Qp(3,3)[]
sage: f = 3*x + 7
sage: g = 5*x + 9
sage: f.xgcd(f*g)  # known bug
((3 + O(3^4))*x + (1 + 2*3 + O(3^3)), (1 + O(3^3)), 0)
```

```
sage: R.<x> = Qp(3,3)[]
sage: f = 490473657*x + 257392844/729
sage: g = 225227399/59049*x - 8669753175
sage: f.xgcd(f*g)  # known bug
((3^3 + 3^5 + 2*3^6 + 2*3^7 + 3^8 + 2*3^10 + 2*3^11 + 3^12 + 3^13 + 3^15 + \ldots)\times x + (2*3^-6 + 2*3^-5 + 3^-3 + 2*3^-2 + 3^-1 + 2*3^-14)), (1 + O(3^20)), 0)
```

```python
sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.make_padic_poly(parent, x, version)
```

### 2.1.16 \(p\)-adic Flat Polynomials

```python
class sage.rings.polynomial.padics.polynomial_padic_flat.Polynomial_padic_flat(parent, x=None, check=True, is_gen=False, construct=False, absprec=None)
```

**Bases:**
- `sage.rings.polynomial.polynomial_element.Polynomial_generic_dense`
- `sage.rings.polynomial.padics.polynomial_padic.Polynomial_padic`
- `sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense`
- `sage.rings.polynomial.padics.polynomial_padic_capped_absolute_dense`
2.1.17 Univariate Polynomials over GF(p^e) via NTL’s ZZ_pEX.

AUTHOR:

• Yann Laigle-Chapuy (2010-01) initial implementation

class sage.rings.polynomial.polynomial_zz_pex.Polynomial_ZZ_pEX
Bases: sage.rings.polynomial.polynomial_zz_pex.Polynomial_template

Univariate Polynomials over GF(p^n) via NTL’s ZZ_pEX.

EXAMPLES:

```
sage: K.<a>=GF(next_prime(2**60)**3)
sage: R.<x> = PolynomialRing(K,implementation='NTL')
sage: (x^3 + a*x^2 + 1) * (x + a)
x^4 + 2*a*x^3 + a^2*x^2 + x + a
```

```
is_irreducible(algorithm='fast_when_false', iter=1)
Returns True precisely when self is irreducible over its base ring.

INPUT:

Parameters

• algorithm – a string (default “fast_when_false”), there are 3 available algorithms: “fast_when_true”, “fast_when_false” and “probabilistic”.

• iter – (default: 1) if the algorithm is “probabilistic” defines the number of iterations. The error probability is bounded by q ** -iter for polynomials in GF(q)[x].

EXAMPLES:

```
sage: K.<a>=GF(next_prime(2**60)**3)
sage: R.<x> = PolynomialRing(K,implementation='NTL')
sage: P = x^3+(2-a)*x+1
sage: P.is_irreducible(algorithm="fast_when_false")
True
sage: P.is_irreducible(algorithm="fast_when_true")
True
sage: P.is_irreducible(algorithm="probabilistic")
True
sage: Q = (x^2+a)*(x+a^3)
sage: Q.is_irreducible(algorithm="fast_when_false")
False
sage: Q.is_irreducible(algorithm="fast_when_true")
False
sage: Q.is_irreducible(algorithm="probabilistic")
False
```

```
list(copy=True)
Returns the list of coefficients.

EXAMPLES:

```
sage: K.<a> = GF(5^3)
sage: P = PolynomialRing(K, 'x')
sage: f = P.random_element(100)
sage: f.list() == [f[i] for i in range(f.degree()+1)]
True
```
```
resultant (other)

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

INPUT:

Parameters other – a polynomial

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

EXAMPLES:

```
sage: K.<a>=GF(next_prime(2**60)**3)
sage: R.<x> = PolynomialRing(K,implementation='NTL')
sage: f=(x-a)*(x-a^2)*(x+1)
sage: g=(x-a^3)*(x-a^4)*(x+a)
sage: r = f.resultant(g)
sage: r == prod(u-v for (u,eu) in f.roots() for (v,ev) in g.roots())
True
```

shift (n)

EXAMPLES:

```
sage: K.<a>=GF(next_prime(2**60)**3)
sage: R.<x> = PolynomialRing(K,implementation='NTL')
sage: f = x^3 + x^2 + 1
sage: f.shift(1)
x^4 + x^3 + x
sage: f.shift(-1)
x^2 + x
```
```python
sage: P.<x> = GF(2)[]
sage: x.degree()
1
sage: P(1).degree()
0
sage: P(0).degree()
-1
```

**gcd**(other)

Return the greatest common divisor of self and other.

**EXAMPLES:**

```python
sage: P.<x> = GF(2)[]
sage: f = x*(x+1)
sage: f.gcd(x+1)
x + 1
sage: f.gcd(x^2)
x
```

**get_cparent()**

**is_gen()**

**EXAMPLES:**

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

**is_one()**

**EXAMPLES:**

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

**is_zero()**

**EXAMPLES:**

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

**list**(copy=True)

**EXAMPLES:**

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

**quo_rem**(right)

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

shift \((n)\)

EXAMPLES:

sage: P.<x> = GF(2)[]
sage: f = x^3 + x^2 + 1
sage: f.shift(1)
x^4 + x^3 + x
sage: f.shift(-1)
x^2 + x

truncate \((n)\)

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

EXAMPLES:

```python
sage: R.<x> =GF(2)[]
sage: f = sum(x^n for n in range(10)); f
x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
sage: f.truncate(6)
x^5 + x^4 + x^3 + x^2 + x + 1
```

If the precision is higher than the degree of the polynomial then the polynomial itself is returned:

```python
sage: f.truncate(10) == f
True
```

xgcd \((other)\)

Computes extended gcd of self and other.

EXAMPLES:

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

sage.rings.polynomial.polynomial_zz_pex.make_element(parent, args)

2.1.18 Isolate Real Roots of Real Polynomials

AUTHOR:

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

This is an implementation of real root isolation. That is, given a polynomial with exact real coefficients, we compute isolating intervals for the real roots of the polynomial. (Polynomials with integer, rational, or algebraic real coefficients are supported.)

We convert the polynomials into the Bernstein basis, and then use de Casteljau’s algorithm and Descartes’ rule of signs on the Bernstein basis polynomial (using interval arithmetic) to locate the roots. The algorithm is similar to that in
“A Descartes Algorithm for Polynomials with Bit-Stream Coefficients”, by Eigenwillig, Kettner, Krandick, Mehlhorn, Schmitt, and Wolpert, but has three crucial optimizations over the algorithm in that paper:

- **Precision reduction**: at certain points in the computation, we discard the low-order bits of the coefficients, widening the intervals.
- **Degree reduction**: at certain points in the computation, we find lower-degree polynomials that are approximately equal to our high-degree polynomial over the region of interest.
- **When the intervals are too wide to continue (either because of a too-low initial precision, or because of precision or degree reduction), and we need to restart with higher precision, we recall which regions have already been proven not to have any roots and do not examine them again.

The best description of the algorithms used (other than this source code itself) is in the slides for my Sage Days 4 talk, currently available from [https://wiki.sagemath.org/days4schedule](https://wiki.sagemath.org/days4schedule).

```python
exception sage.rings.polynomial.real_roots.PrecisionError
    Bases: ValueError

sage.rings.polynomial.real_roots.bernstein_down(d1, d2, s)
    Given polynomial degrees d1 and d2 (where d1 < d2), and a number of samples s, computes a matrix bd.
    If you have a Bernstein polynomial of formal degree d2, and select s of its coefficients (according to subsample_vec), and multiply the resulting vector by bd, then you get the coefficients of a Bernstein polynomial of formal degree d1, where this second polynomial is a good approximation to the first polynomial over the region of the Bernstein basis.

    EXAMPLES:
    sage: from sage.rings.polynomial.real_roots import *
    sage: bernstein_down(3, 8, 5)
    [[ 612/245  -348/245  -37/49  338/245 -172/245]
     [-724/441  132/49  395/441 -290/147  452/441]
     [ 452/441 -290/147  395/441  132/49 -724/441]
     [-172/245  338/245 -37/49 -348/245  612/245]]
```

```python
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, -1190, -1595, -2032, -2500, -3000), 1)
```

```python
class sage.rings.polynomial.real_roots.bernstein_polynomial_factory
    Bases: object
```

Chapter 2. Univariate Polynomials
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.

```python
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.)

**coeffs_bitsize()**
Computes the approximate log2 of the maximum of the absolute values of the coefficients.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(AA)
sage: p = (x - 1) * (x - sqrt(AA(2))) * (x - 2)
sage: 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>
```

```
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.

**coeffs_bitsize()**
Computes the approximate log2 of the maximum of the absolute values of the coefficients.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(AA)
sage: p = (x - 1) * (x - sqrt(AA(2))) * (x - 2)
sage: bernstein_polynomial_factory_ar(p, False).coeffs_bitsize()
1
```
**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
```

(continues on next page)
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, Strzeboński, and Vigklas, and picks the better result.

EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *
sage: cl_maximum_root([RIF(-1), RIF(0), RIF(1)])
1.00000000000000

sage.rings.polynomial.real_roots.cl_maximum_root_first_lambda(cl)

Given a polynomial represented by a list of its coefficients (as RealIntervalFieldElements), compute an upper bound on its largest real root.


EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *
sage: cl_maximum_root_first_lambda([RIF(-1), RIF(0), RIF(1)])
1.00000000000000

sage.rings.polynomial.real_roots.cl_maximum_root_local_max(cl)

Given a polynomial represented by a list of its coefficients (as RealIntervalFieldElements), compute an upper bound on its largest real root.


EXAMPLES:
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()

de_casteljau_doublevec(c, x)
Given a polynomial in Bernstein form with floating-point coefficients over the region [0 .. 1], and a split point x, use de Casteljau’s algorithm to give polynomials in Bernstein form over [0 .. x] and [x .. 1].

This function will work for an arbitrary rational split point x, as long as 0 < x < 1; but it has a specialized code path for x==1/2.

INPUT:

• c – vector of coefficients of polynomial in Bernstein form
• x – rational splitting point; 0 < x < 1

OUTPUT:

• c1 – coefficients of polynomial over range [0 .. x]
• c2 – coefficients of polynomial over range [x .. 1]
• err_inc – number of half-ulps by which error intervals widened

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: c = vector(RDF, [0.7, 0, 0, 0, 0, 0])
sage: de_casteljau_doublevec(c, 1/2)
((0.7, 0.35, 0.175, 0.0875, 0.04375, 0.021875), (0.021875, 0.0, 0.0, 0.0, 0.0, 0.
˓→0), 5)
sage: de_casteljau_doublevec(c, 1/3)  # rel tol
((0.7, 0.4666666666666667, 0.31111111111111117, 0.20740740740740746, 0.
˓→13827160493827165, 0.092180106995884777), (0.092180106995884777, 0.0, 0.0, 0.0, 0.
˓→0, 0.0), 15)
sage: de_casteljau_doublevec(c, 7/22)  # rel tol
((0.7, 0.4772727272727273, 0.3254132231404959, 0.2218726521424724, 0.
˓→15127680827812312, 0.10314327837144759), (0.10314327837144759, 0.0, 0.0, 0.0, 0.
˓→0, 0.0), 15)
```

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 \( x = \frac{a}{a + b} \) and \( y = \frac{c}{c + d} \), where \( \min(a, b) \) and \( \min(c, d) \) fit in the same number of machine words and \( a + b \) and \( c + d \) are both powers of two, then \( x \) and \( y \) should be equally fast split points.

If use_ints is nonzero, then instead of checking whether numerators and denominators fit in machine words, we check whether they fit in ints (32 bits, even on 64-bit machines). This slows things down, but allows for identical results across machines.

**INPUT:**

- \( c \) – vector of coefficients of polynomial in Bernstein form
- \( c\_bitsize \) – approximate size of coefficients in \( c \) (in bits)
- \( x \) – rational splitting point; \( 0 < x < 1 \)

**OUTPUT:**

- \( c1 \) – coefficients of polynomial over range \([0 .. x] \)
- \( c2 \) – coefficients of polynomial over range \([x .. 1] \)
- \( err\_inc \) – amount by which error intervals widened

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: c = vector(ZZ, [1048576, 0, 0, 0, 0, 0])
sage: de_casteljau_intvec(c, 20, 1/2, 1)
((1048576, 524288, 262144, 131072, 65536, 32768), (32768, 0, 0, 0, 0, 0), 1)
sage: de_casteljau_intvec(c, 20, 1/3, 1)
((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)
```

**degree_reduction_next_size(n)**

Given \( n \) (a polynomial degree), returns either a smaller integer or None. This defines the sequence of degrees followed by our degree reduction implementation.

**EXAMPLES:**

```python
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
```

**dprod_imatrow_vec(m, v, k)**

Computes the dot product of row \( k \) of the matrix \( m \) with the vector \( v \) (that is, compute one element of the product \( m^*v \)).

If \( v \) has more elements than \( m \) has columns, then elements of \( v \) are selected using subsample_vec.

**EXAMPLES:**

```python
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:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: get_realfield_rndu(20)
Real Field with 20 bits of precision and rounding RNDU
sage: get_realfield_rndu(53)
Real Field with 53 bits of precision and rounding RNDU
sage: get_realfield_rndu(20)
Real Field with 20 bits of precision and rounding RNDU
```

class sage.rings.polynomial.real_roots.interval_bernstein_polynomial
Bases: object

An interval_bernstein_polynomial is an approximation to an exact polynomial. This approximation is in the form of a Bernstein polynomial (a polynomial given as coefficients over a Bernstein basis) with interval coefficients.

The Bernstein basis of degree n over the region [a .. b] is the set of polynomials

\[
\binom{n}{k} (x - a)^k (b - x)^{n-k} / (b - a)^n
\]

for 0 ≤ k ≤ 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 $P_1$ over $[a .. b]$ and $P_2$ over $[b .. c]$, where $P_1$ and $P_2$ contain (or bound) all polynomials that $P$ contains (or bounds).

- Given a degree-$n$ interval Bernstein polynomial $P$ over $[a .. b]$, and $m < n$, we can compute a degree-$m$ interval Bernstein polynomial $P'$ over $[a .. b]$ that bounds all polynomials that $P$ bounds.

- It is sometimes possible to prove that no polynomial bounded by $P$ over $[a .. b]$ has any roots in $[a .. b]$. (Roughly, this is possible when no polynomial contained by $P$ has any complex roots near the line segment $[a .. b]$, where “near” is defined relative to the length $b-a$.)

- It is sometimes possible to prove that every polynomial bounded by $P$ over $[a .. b]$ with slope error $E$ has exactly one root in $[a .. b]$. (Roughly, this is possible when every polynomial contained by $P$ over $[a .. b]$ has exactly one root in $[a .. b]$, there are no other complex roots near the line segment $[a .. b]$, and every polynomial contained in $P$ has a derivative which is bounded away from zero over $[a .. b]$ by an amount which is large relative to $E$.)

- Starting from a sufficiently precise interval Bernstein polynomial, it is always possible to split it into polynomials which provably have 0 or 1 roots (as long as your original polynomial has no multiple real roots).

So a rough outline of a family of algorithms would be:

- Given a polynomial $p$, compute a region $[a .. b]$ in which any real roots must lie.

- Compute an interval Bernstein polynomial $P$ containing $p$ over $[a .. b]$.

- Keep splitting $P$ until you have isolated all the roots. Optionally, reduce the degree or the precision of the interval Bernstein polynomials at intermediate stages (to reduce computation time). If this seems not to be working, go back and try again with higher precision.

Obviously, there are many details to be worked out to turn this into a full algorithm, like:

- What initial precision is selected for computing $P$?

- How do you decide when to reduce the degree of intermediate polynomials?

- How do you decide when to reduce the precision of intermediate polynomials?

- How do you decide where to split the interval Bernstein polynomial regions?

- How do you decide when to give up and start over with higher precision?

Each set of answers to these questions gives a different algorithm (potentially with very different performance characteristics), but all of them can use this `interval_bernstein_polynomial` class as their basic building block.

To save computation time, all coefficients in an `interval_bernstein_polynomial` share the same interval width. (There is one exception: when creating an `interval_bernstein_polynomial`, the first and last coefficients can be marked as “known positive” or “known negative”. This has some of the same effect as having a (potentially) smaller interval width for these two coefficients, although it does not affect de Casteljau splitting.) To allow for widely varying coefficient magnitudes, all coefficients in an `interval_bernstein_polynomial` are scaled by $2^n$ (where $n$ may be positive, negative, or zero).

There are two representations for `interval_bernstein_polynomials`, integer and floating-point. These are the two subclasses of this class; `interval_bernstein_polynomial` itself is an abstract class.
interval_bernstein_polynomial and its subclasses are not expected to be used outside this file.

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
def from sage.rings.polynomial.real_roots import *
def bp = mk_ibpi([50, 20, -90, -70, 200], error=5)
def bp1, bp2, _ = bp.try_rand_split(mk_context(), None)
def bp1
<IBP: (50, 29, -27, -56, -11) + [0 .. 6) over [0 .. 43/64]>
def bp2
<IBP: (-11, 10, 49, 111, 200) + [0 .. 6) over [43/64 .. 1]>
def bp1, bp2, _ = bp.try_rand_split(mk_context(seed=42), None)
def bp1
<IBP: (50, 32, -11, -41, -29) + [0 .. 6) over [0 .. 583/1024]>
def bp2
<IBP: (-29, -20, 13, 83, 200) + [0 .. 6) over [583/1024 .. 1]>
def bp = mk_ibpf([0.5, 0.2, -0.9, -0.7, 0.99], neg_err=-0.1, pos_err=0.01)
def bp1, bp2, _ = bp.try_rand_split(mk_context(), None)
def bp1
<IBP: (0.5, 0.2984375, -0.2642578125, -0.5511661529541015, -0.3145806974172592) + [-0.10000000000000069 .. 0.010000000000000677] over [0 .. 1/2]>
def bp2
<IBP: (-0.3145806974172592, -0.19903896331787108, 0.04135986328125002, 0.99) + [-0.10000000000000069 .. 0.010000000000000677] over [1/2 .. 1]>
```

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
def from sage.rings.polynomial.real_roots import *
def bp = mk_ibpi([50, 20, -90, -70, 200], error=5)
def bp1, bp2, _ = bp.try_split(mk_context(), None)
def bp1
<IBP: (50, 35, 0, -29, -31) + [0 .. 6) over [0 .. 1/2]>
def bp2
<IBP: (-31, -33, -8, 65, 200) + [0 .. 6) over [1/2 .. 1]>
def bp = mk_ibpf([0.5, 0.2, -0.9, -0.7, 0.99], neg_err=-0.1, pos_err=0.01)
def bp1, bp2, _ = bp.try_split(mk_context(), None)
def bp1
<IBP: (0.5, 0.35, 0, -0.2875, -0.369375) + [-0.10000000000000023 .. 0.010000000000000226] over [0 .. 1/2]>
def bp2
<IBP: (-0.369375, -0.45125, -0.3275, 0.14500000000000002, 0.99) + [-0.10000000000000023 .. 0.010000000000000226] over [1/2 .. 1]>
```
Consider a polynomial (written in either the normal power basis or the Bernstein basis). Take its list of coefficients, omitting zeroes. Count the number of positions in the list where the sign of one coefficient is opposite the sign of the next coefficient.

This count is the number of sign variations of the polynomial. According to Descartes’ rule of signs, the number of real roots of the polynomial (counted with multiplicity) in a certain interval is always less than or equal to the number of sign variations, and the difference is always even. (If the polynomial is written in the power basis, the region is the positive reals; if the polynomial is written in the Bernstein basis over a particular region, then we count roots in that region.)

In particular, a polynomial with no sign variations has no real roots in the region, and a polynomial with one sign variation has one real root in the region.

In an interval Bernstein polynomial, we do not necessarily know the signs of the coefficients (if some of the coefficient intervals contain zero), so the polynomials contained by this interval polynomial may not all have the same number of sign variations. However, we can compute a range of possible numbers of sign variations.

This function returns the range, as a 2-tuple of integers.

```python
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: (0, 0)
sage: bp = mk_ibpf([-0.3, -0.1, 0.1, -0.1, -0.3, -0.1], lower=1, upper=5/4, usign=1, pos_err=0.2, scale_log2=-3, level=2, slope_err=RIF(pi)); print(bp)
degree 5 IBP with floating-point coefficients
sage: 1 IBP: ((-0.3, -0.1, 0.1, -0.1, -0.3, -0.1) + [0.0 .. 0.2]) * 2^-3 over [1 .. 5/4]; usign 1; level 2; slope_err 3.141592653589794?
```

EXAMPLES:

```python
sage: as_float()
```

**de_casteljau**(ctx, mid, msign=0)

Uses de Casteljau’s algorithm to compute the representation of this polynomial in a Bernstein basis over new regions.

INPUT:
• mid – where to split the Bernstein basis region; 0 < mid < 1
• msign – default 0 (unknown); the sign of this polynomial at mid

OUTPUT:
• bp1, bp2 – the new interval Bernstein polynomials
• ok – a boolean; True if the sign of the original polynomial at mid is known

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: ctx = mk_context()
sage: bp = mk_ibpf([0.5, 0.2, -0.9, -0.7, 0.99], neg_err=-0.1, pos_err=0.01)
sage: bp1, bp2, ok = bp.de_casteljau(ctx, 1/2)
sage: bp1
<IBP: (0.5, 0.35, 0.0, -0.2875, -0.369375) + [-0.10000000000000023 .. 0.010000000000000226] over [0 .. 1/2]>
sage: bp2
<IBP: (-0.369375, -0.45125, -0.3275, 0.14500000000000002, 0.99) + [-0.10000000000000023 .. 0.010000000000000226] over [1/2 .. 1]>
sage: bp1 # rel tol 2e-16
<IBP: (0.5, 0.30000000000000004, -0.2555555555555555, -0.5444444444444444, -0.32172839506172846) + [-0.100000000000000069 .. 0.010000000000000677] over [0 .. 2/3]>
sage: bp2 # rel tol 3e-15
<IBP: (-0.32172839506172846, -0.21037037037037046, 0.028888888888888797, 0.99) + [-0.100000000000000069 .. 0.010000000000000677] over [2/3 .. 1]>
sage: bp1, bp2, ok = bp.de_casteljau(ctx, 2/3)
sage: bp1
# rel tol
<IBP: (0.5, 0.4461538461538461, 0.36653517422748183, 0.27328680523946786, 0.176592706232836) + [-0.100000000000000069 .. 0.010000000000000677] over [0 .. 7/39]>
sage: bp2
# rel tol
<IBP: (0.176592706232836, -0.26556803047927313, -0.780203812807364, -0.3966666666666666, 0.99) + [-0.100000000000000069 .. 0.010000000000000677] over [7/39 .. 1]>
```

get_msb_bit()
Returns an approximation of the log2 of the maximum of the absolute values of the coefficients, as an integer.

slope_range()
Compute a bound on the derivative of this polynomial, over its region.

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpf([0.5, 0.2, -0.9, -0.7, 0.99], neg_err=-0.1, pos_err=0.01)
sage: bp.slope_range().str(style='brackets')
'[-4.8400000000000017 .. 7.2000000000000011]'
```

class sage.rings.polynomial.real_roots.interval_bernstein_polynomial_integer
Bases: 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 \( A \cdot 2^n \leq c < (A+E) \cdot 2^n \).

(Note that \texttt{mk_ibpi} is a simple helper function for creating elements of interval_bernstein_polynomial_integer in doctests.)

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([1, 2, 3], error=5); print(bp)
degree 2 IBP with 2-bit coefficients
sage: bp
<IBP: (1, 2, 3) + [0 .. 5)>
sage: bp.variations()
(0, 0)
sage: bp = mk_ibpi([-3, -1, 1, -1, -3, -1], lower=1, upper=5/4, usign=1, error=2, scale_log2=-3, level=2, slope_err=RIF(pi)); print(bp)
degree 5 IBP with 2-bit coefficients
sage: bp
<IBP: ((-3, -1, 1, -1, -3, -1) + [0 .. 2)) \cdot 2^{-3} over [1 .. 5/4]; usign 1; level \rightarrow 2; slope_err 3.141592653589794?>
sage: bp.variations()
(3, 3)
```

**as_float()**

Compute an interval_bernstein_polynomial_float which contains (or bounds) all the polynomials this interval polynomial contains (or bounds).

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([50, 20, -90, -70, 200], error=5)
sage: print(bp.as_float())
degree 4 IBP with floating-point coefficients
sage: bp.as_float()
<IBP: ((0.1953125, 0.078125, -0.3515625, -0.2734375, 0.78125) + [-1.2^{1275705389246e-16 .. 0.01953125000000017]) \cdot 2^8>
```

**de_casteljau(ctx, mid, msign=0)**

Uses de Casteljau’s algorithm to compute the representation of this polynomial in a Bernstein basis over new regions.

**INPUT:**

- \( \text{mid} \) – where to split the Bernstein basis region; \( 0 < \text{mid} < 1 \)
- \( \text{msign} \) – default 0 (unknown); the sign of this polynomial at \( \text{mid} \)

**OUTPUT:**

- \( \text{bp1, bp2} \) – the new interval Bernstein polynomials
- \( \text{ok} \) – a boolean; True if the sign of the original polynomial at \( \text{mid} \) is known

**EXAMPLES:**

```python
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)
```
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” \(\text{expected\_err}\), 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 \(\text{max\_err}\), and by \(\text{expected\_err} < \text{exp\_err\_shift}\). 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
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 .. 9); 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 \(1 << \text{max\_scale}\).

EXAMPLES:

```sage
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\((\text{bits})\)

Compute an interval_bernstein_polynomial_integer which contains (or bounds) all the polynomials this interval polynomial contains (or bounds), but uses “bits” fewer bits.

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([0, 100, 400, 903], error=2)
sage: bp.downscale(5)
<IBP: ((0, 3, 12, 28) + [0 .. 1)) \times 2^5>
```

get_msb_bit

Returns an approximation of the log2 of the maximum of the absolute values of the coefficients, as an integer.

slope_range

Compute a bound on the derivative of this polynomial, over its region.

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([0, 100, 400, 903], error=2)
sage: bp.slope_range().str(style='brackets')
'[294.00000000000000 .. 1515.0000000000000]'
```

```python
sage.rings.polynomial.real_roots.intvec_to_doublevec\((b, err)\)

Given a vector of integers \(A = [a_1, \ldots, a_n]\), and an integer error bound \(E\), returns a vector of floating-point numbers \(B = [b_1, \ldots, b_n]\), lower and upper error bounds \(F_1\) and \(F_2\), and a scaling factor \(d\), such that

\[
(b_k + F_1) \times 2^d \leq a_k
\]

and

\[
ak + E \leq (b_k + F_2) \times 2^d
\]

If \(b_j\) is the element of \(B\) with largest absolute value, then \(0.5 \leq \text{abs}(b_j) < 1.0\).

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: intvec_to_doublevec(vector(ZZ, [1, 2, 3, 4, 5]), 3)
((0.125, 0.25, 0.375, 0.5, 0.625), -1.1275702593849246e-16, 0.37500000000000017, \rightarrow 3)
```

class sage.rings.polynomial.real_roots.island

Bases: object

This implements the island portion of my ocean-island root isolation algorithm. See the documentation for class ocean, for more information on the overall algorithm.

Island root refinement starts with a Bernstein polynomial whose region is the whole island (or perhaps slightly more than the island in certain cases). There are two subalgorithms; one when looking at a Bernstein polynomial covering a whole island (so we know that there are gaps on the left and right), and one when looking at a Bernstein polynomial covering the left segment of an island (so we know that there is a gap on the left, but the right is in the middle of an island). An important invariant of the left-segment subalgorithm over the region \([l .. r]\) is that it always finds a gap \([r_0 .. r]\) ending at its right endpoint.

Ignoring degree reduction, downscaling (precision reduction), and failures to split, the algorithm is roughly:

Whole island:
1. If the island definitely has exactly one root, then return.

2. Split the island in (approximately) half.

3. If both halves definitely have no roots, then remove this island from its doubly-linked list (merging its left and right gaps) and return.

4. If either half definitely has no roots, then discard that half and call the whole-island algorithm with the other half, then return.

5. If both halves may have roots, then call the left-segment algorithm on the left half.

6. We now know that there is a gap immediately to the left of the right half, so call the whole-island algorithm on the right half, then return.

**Left segment:**

1. Split the left segment in (approximately) half.

2. If both halves definitely have no roots, then extend the left gap over the segment and return.

3. If the left half definitely has no roots, then extend the left gap over this half and call the left-segment algorithm on the right half, then return.

4. If the right half definitely has no roots, then split the island in two, creating a new gap. Call the whole-island algorithm on the left half, then return.

5. Both halves may have roots. Call the left-segment algorithm on the left half.

6. We now know that there is a gap immediately to the left of the right half, so call the left-segment algorithm on the right half, then return.

Degree reduction complicates this picture only slightly. Basically, we use heuristics to decide when degree reduction might be likely to succeed and be helpful; whenever this is the case, we attempt degree reduction.

Precision reduction and split failure add more complications. The algorithm maintains a stack of different-precision representations of the interval Bernstein polynomial. The base of the stack is at the highest (currently known) precision; each stack entry has approximately half the precision of the entry below it. When we do a split, we pop off the top of the stack, split it, then push whichever half we're interested in back on the stack (so the different Bernstein polynomials may be over different regions). When we push a polynomial onto the stack, we may heuristically decide to push further lower-precision versions of the same polynomial onto the stack.

In the algorithm above, whenever we say “split in (approximately) half”, we attempt to split the top-of-stack polynomial using try_split() and try_rand_split(). However, these will fail if the sign of the polynomial at the chosen split point is unknown (if the polynomial is not known to high enough precision, or if the chosen split point actually happens to be a root of the polynomial). If this fails, then we discard the top-of-stack polynomial, and try again with the next polynomial down (which has approximately twice the precision). This next polynomial may not be over the same region; if not, we split it using de Casteljau’s algorithm to get a polynomial over (approximately) the same region first.

If we run out of higher-precision polynomials (if we empty out the entire stack), then we give up on root refinement for this island. The ocean class will notice this, provide the island with a higher-precision polynomial, and restart root refinement. Basically the only information kept in that case is the lower and upper bounds on the island. Since these are updated whenever we discover a “half” (of an island or a segment) that definitely contains no roots, we never need to re-examine these gaps. (We could keep more information. For example, we could keep a record of split points that succeeded and failed. However, a split point that failed at lower precision is likely to succeed at higher precision, so it’s not worth avoiding. It could be useful to select split points that are known to succeed, but starting from a new Bernstein polynomial over a slightly different region, hitting such split points would require de Casteljau splits with non-power-of-two denominators, which are much much slower.)
bp_done \((bp)\)
Examine the given Bernstein polynomial to see if it is known to have exactly one root in its region. (In addition, we require that the polynomial region not include 0 or 1. This makes things work if the user gives explicit bounds to real_roots(), where the lower or upper bound is a root of the polynomial. real_roots() deals with this by explicitly detecting it, dividing out the appropriate linear polynomial, and adding the root to the returned list of roots; but then if the island considers itself “done” with a region including 0 or 1, the returned root regions can overlap with each other.)

done \((ctx)\)
Check to see if the island is known to contain zero roots or is known to contain one root.

has_root ()
Assuming that the island is done (has either 0 or 1 roots), reports whether the island has a root.

less_bits \((ancestors, bp)\)
Heuristically pushes lower-precision polynomials on the polynomial stack. See the class documentation for class island for more information.

more_bits \((ctx, ancestors, bp, rightmost)\)
Find a Bernstein polynomial on the “ancestors” stack with more precision than bp; if it is over a different region, then shrink its region to (approximately) match that of bp. (If this is rightmost – if bp covers the whole island – then we only require that the new region cover the whole island fairly tightly; if this is not rightmost, then the new region will have exactly the same right boundary as bp, although the left boundary may vary slightly.)

refine \((ctx)\)
Attempts to shrink and/or split this island into sub-island that each definitely contain exactly one root.

refine_recurse \((ctx, bp, ancestors, history, rightmost)\)
This implements the root isolation algorithm described in the class documentation for class island. This is the implementation of both the whole-island and the left-segment algorithms; if the flag rightmost is True, then it is the whole-island algorithm, otherwise the left-segment algorithm.

The precision-reduction stack is \((ancestors + [bp])\); that is, the top-of-stack is maintained separately.

reset_root_width \((target_width)\)
Modify the criteria for this island to require that it is not “done” until its width is less than or equal to target_width.

shrink_bp \((ctx)\)
If the island’s Bernstein polynomial covers a region much larger than the island itself (in particular, if either the island’s left gap or right gap are totally contained in the polynomial’s region) then shrink the polynomial down to cover the island more tightly.

class sage.rings.polynomial.real_roots.linear_map \((lower, upper)\)
Bases: object
A simple class to map linearly between original coordinates (ranging from \([lower .. upper]\)) and ocean coordinates (ranging from \([0 .. 1]\)).

from_ocean \((region)\)

to_ocean \((region)\)
sage.rings.polynomial.real_roots.max_abs_doublevec \((c)\)
Given a floating-point vector, return the maximum of the absolute values of its elements.

EXAMPLES:
sage: from sage.rings.polynomial.real_roots import *
sage: max_abs_doublevec(vector(RDF, [0.1, -0.767, 0.3, 0.693]))
0.767

sage.rings.polynomial.real_roots.max_bitsize_intvec_doctest(b)
sage.rings.polynomial.real_roots.maximum_root_first_lambda(p)
Given a polynomial with real coefficients, computes an upper bound on its largest real root, using the first-lambda algorithm from “Implementations of a New Theorem for Computing Bounds for Positive Roots of Polynomials”, by Akritas, Strzeboński, and Vigklas.

EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: maximum_root_first_lambda((x-1)*(x-2)*(x-3))
6.00000000000001
sage: maximum_root_first_lambda((x+1)*(x+2)*(x+3))
0.000000000000000
sage: maximum_root_first_lambda(x^2 - 1)
1.00000000000000

sage.rings.polynomial.real_roots.maximum_root_local_max(p)
Given a polynomial with real coefficients, computes an upper bound on its largest real root, using the local-max algorithm from “Implementations of a New Theorem for Computing Bounds for Positive Roots of Polynomials”, by Akritas, Strzeboński, and Vigklas.

EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: maximum_root_local_max((x-1)*(x-2)*(x-3))
12.0000000000001
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 \( b = (b_0, \ldots, b_n) \), 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.

**approx_bp(scale_log2)**

Returns an approximation to our Bernstein polynomial with the given scale_log2.

**find_roots()**

Isolate all roots in this ocean.

**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().)
from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1/3, -22/7, 193/71, -140/99]), lmap)
sage: oc
ocean with precision 120 and 1 island(s)
sage: oc.increase_precision()
sage: oc.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, -140/99]), lmap)
sage: oc
ocean with precision 120 and 1 island(s)
sage: oc.refine_all()
sage: oc
ocean with precision 120 and 3 island(s)

reset_root_width(isle_num, target_width)
Require that the isle_num island have a width at most target_width.
If this is followed by a call to find_roots(), then the corresponding root will be refined to the specified width.

EXAMPLES:

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: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1, 0, -1111/2, 0, 11108889/14, 0, 0, 0, 0, -1]), lmap)

sage: oc.find_roots()

sage: oc.roots()

[(95761241267509487747625/9671406556917033397649408, 191522482605387719863145/19342813113834066795298816), (1496269395904347376805/151115727451828646838272, 374067366568272936175/37778931862957161709568), (-31/32, 63/64)]

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, \qquad \text{bounds} = \text{None}, \quad \text{seed} = \text{None}, \\
\text{skip\_squarefree} = \text{False}, \quad \text{do\_logging} = \text{False}, \\
\text{wordsize} = 32, \quad \text{retval} = \text{\textquoteleft rational\textquoteright}, \\
\text{strategy} = \text{None}, \quad \text{max\_diameter} = \text{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\textquoteleft=\textquoteleft rational\textquoteright, then it is returned as a pair of rationals that define a region that includes exactly one root. If retval\textquoteleft=\textquoteleft interval\textquoteright, then it is returned as a \texttt{RealIntervalFieldElement} that includes exactly one root. If retval\textquoteleft=\textquoteleft algebraic\_real\textquoteright, then it is returned as an \texttt{AlgebraicReal}. In the former two cases, all the intervals are disjoint.

An alternate high-level algorithm can be used by selecting strategy\textquoteleft=\textquoteleft warp\textquoteright. This affects the conversion into Bernstein polynomial form, but still uses the same ocean-island algorithm as the default algorithm. The \textquoteleft warp\textquoteright algorithm performs the conversion into Bernstein polynomial form much more quickly, but performs the rest of the computation slightly slower in some benchmarks. The \textquoteleft warp\textquoteright 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\textquoteleft=\textquoteleft warp\textquoteright, 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\texttt() 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\textquoteleft=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:
\texttt{sage: real_roots((x-1)*(x-2)*(x-3)*(x-5)*(x-8)*(x-13)*(x-21)*(x-34))}
\texttt{[((7/4, 19/8), 1), ((11/16, 33/32), 1), ((11/8, 33/16), 1), ((11/4, 55/16), 1), ((77/16, 165/32), 1), ((11/2, 33/4), 1), ((11, 55/4), 1), ((165/8, 341/16), 1), ((22, 44), 1)]}

\texttt{sage: real_roots(x^5 * (x^2 - 9999)^2 - 1)}
\texttt{[((-29274496381311/9007199254740992, 41960125186091/2251799813685248), 1), ((11/16, 33/32), 1), ((11/8, 33/16), 1), ((11/4, 55/16), 1), ((77/16, 165/32), 1), ((11/2, 33/4), 1), ((11, 55/4), 1), ((165/8, 341/16), 1), ((22, 44), 1)]}

\texttt{sage: real_roots(x^5 * (x^2 - 9999)^2 - 1, seed=42)}
\texttt{[((-123196838480289/18014398509481984, 293964743458749/9007199254740992), 1), ((11/16, 33/32), 1), ((11/8, 33/16), 1), ((11/4, 55/16), 1), ((77/16, 165/32), 1), ((11/2, 33/4), 1), ((11, 55/4), 1), ((165/8, 341/16), 1), ((22, 44), 1)]}

\texttt{sage: real_roots(x^5 * (x^2 - 9999)^2 - 1, wordsize=64)}
\texttt{[((-6286503803202151050003/19342813113834066795928816, 90108554512564177624143/483570327845816698824704), 1), ((-4444256323733151241990897292809050101157/5444518787073501541541399371898291383292, 1), 10888491729660194637118845654929604385439/1088903574170030830827987437816582766592, 1), ((-52721281929661439711063928686060007142141/27222589333765707707069685945145691648, 1), 136106141275823501951003939373685486562633/13611294676837538535498429727072845824, 1)]}

\texttt{sage: real_roots(x)}
\texttt{[((-47/256, 81/512), 1), (1, 1)]}

\texttt{sage: real_roots(x * (x-1))}
\texttt{[((-47/256, 81/512), 1), ((1/2, 1201/1024), 1)]}

\texttt{sage: real_roots(x-1)}
\texttt{[((209/256, 593/512), 1)]}

\texttt{sage: real_roots(x*(x-1)*(x-2), bounds=(0, 2))}
\texttt{[((0, 0), 1), ((81/128, 337/256), 1), ((2, 2), 1)]}

\texttt{sage: real_roots(x*(x-1)*(x-2), bounds=(0, 2), retval='algebraic_real')}
\texttt{[(0, 1), (1, 1), (2, 1)]}

\texttt{sage: v = 2^40}
\texttt{sage: real_roots((x^2-1)^2 * (x^2 - (v+1)/v))}
\texttt{[((-6427749906842725/562949953421275/56294995342132, 2), 0), ((-22, 44), 2), ((50925899408362152157114221034542056268267098416484016265690351233859532494083417654894344, 2571100870815553642177003804234824013625707403573393563630791, 25711008708143844040867139347745786016435254790052568354622016, 1)]}

\texttt{sage: real_roots(x^2 - 2)}
\texttt{((50925899408362152157114221034542056268267098416484016265690351233859532494083417654894344, 2571100870815553642177003804234824013625707403573393563630791, 25711008708143844040867139347745786016435254790052568354622016, 1)]}
[((-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)
[(-2.?, 1), (2.?, 1)]
sage: real_roots(x^2 - 2, max_diameter=1/2^30)
[((-
˓→22506280506048040147267537595888654364534879097901251919845680573131269246430553365310109/
˓→15914435651131725489723194069826683214596825515126958094847260581103904401068017057792,
˓→1), (45012561012096082945350759197773086542448972309421182031053065959548949685605179935498343/
˓→3182687130226345097944463881396533766429193651030259161896945211262207808802136034115584),
˓→1)]
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')
[(-1.000000000000000000?, 1), (1.000000000000000000?, 1)]
sage: v = 2^60
sage: real_roots((x-1) * (x-(v+1)/v), retval='interval')
[(-1.000000000000000000000000000?, 1), (1.000000000000000000000000000?, 1)]
sage: real_roots((x+3)*(x+1)*x*(x-1)*(x-2), strategy='warp')
[((499/525, 1173/875), 1), ((337/175, 849/175), 1)]
sage: real_roots((x+3)*(x+1)*x*(x-1)*(x-2), strategy='warp', retval='algebraic_real')
[(-3.000000000000000?, 1), (-1.000000000000000000?, 1), (0, 1), (1.000000000000000000?, 1),
˓→1), (2.000000000000000000?, 1)]

If the polynomial has no real roots, we get an empty list.

sage: (x^2 + 1).real_root_intervals()
[
]

We can compute Conway’s constant (see http://mathworld.wolfram.com/ConwaysConstant.html) to arbitrary precision.

sage: p = x^71 - x^69 - 2*x^68 - x^67 + 2*x^66 + 2*x^65 + x^64 - x^63 - x^62 - x^61 - x^60 - x^59 + 2*x^58 + 5*x^57 + 3*x^56 - 2*x^55 - 10*x^54 - 3*x^53 - 2*x^52 + 6*x^51 + 6*x^50 + 9*x^49 - 3*x^48 - 3*x^47 + 6*x^46 + 8*x^45 + 8*x^44 + 10*x^43 + 6*x^42 + 8*x^41 - 5*x^40 - 12*x^39 + 7*x^38 - 7*x^37 + 7*x^36 + x^35 - 3*x^34 + 10*x^33 + x^32 - 6*x^31 - 2*x^30 - 10*x^29 - 3*x^28 + 2*x^27 + 9*x^26 - 3*x^25 + 14*x^24 - 8*x^23 - 7*x^21 + 9*x^20 + 3*x^19 - 4*x^18 - 10*x^17 - 7*x^16 + 12*x^15 + 7*x^14 + 2*x^13 - 12*x^12 - 4*x^11 - 2*x^10 + 5*x^9 + x^7 - 7*x^6 + 7*x^5 - 4*x^4 + 12*x^3 - 6*x^2 + 3*x - 6
(continues on next page)
Now we play with algebraic real coefficients.

```python
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 + 63.98526763257801?*x^2 - 35.37613490585595?*x + 8.028284271247462?
sage: real_roots(p2, retval='interval')
[(1.00?, 1), (1.1?, 1), (1.38?, 1), (1.5?, 1), (2.00?, 1), (2.1?, 1)]
sage: p = (x - 1) * (x - sqrt(AA(2)))^2 * (x - 2)^3 * sqrt(AA(3))
sage: real_roots(p, retval='interval')
[(1.000000000000000?, 1), (1.414213562373095?, 2), (2.000000000000000?, 3)]
```

```
 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)
```

```
 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)
```
Given a polynomial with real coefficients, computes a lower and upper bound on its real roots. Uses algorithms of Akritas, Strzeboński, and Vigklas.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: root_bounds((x-1)*(x-2)*(x-3))
(0.545454545454545, 6.00000000000001)
sage: root_bounds(x^2)
(0.000000000000000, 0.000000000000000)
sage: root_bounds(x*(x+1))
(-1.000000000000000, 0.000000000000000)
sage: root_bounds((x+2)*(x-3))
(-2.44948974278317, 3.46410161513776)
sage: root_bounds(x^995 * (x^2 - 9999) - 1)
(-99.9949998749937, 141.414284992713)
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.

```python
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^n) * (kn^(n-i)).

Modifies the input vector; has no return value.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: v = vector(ZZ, [1, 1, 1, 1])
sage: scale_intvec_var(v, 3/4)
sage: v
(64, 48, 36, 27)
```

sage.rings.polynomial.real_roots.split_for_targets(ctx, bp, target_list, precise=False)

Given an interval Bernstein polynomial over a particular region (assumed to be a (not necessarily proper) sub-region of [0 .. 1]), and a list of targets, uses de Casteljau’s method to compute representations of the Bernstein polynomial over each target. Uses degree reduction as often as possible while maintaining the requested precision.

Each target is of the form (lgap, ugap, b). Suppose lgap.region() is (l1, l2), and ugap.region() is (u1, u2). Then we will compute an interval Bernstein polynomial over a region [l .. u], where l1 <= l <= l2 and u1 <= u <= u2. (split_for_targets() is free to select arbitrary region endpoints within these bounds; it picks endpoints which make the computation easier.) The third component of the target, b, is the maximum allowed scale_log2 of the result; this is used to decide when degree reduction is allowed.
The pair \((l_1, l_2)\) can be replaced by None, meaning \([-\infty .. 0]\); or, \((u_1, u_2)\) can be replaced by None, meaning \([1 .. \infty]\).

There is another constraint on the region endpoints selected by \texttt{split_for_targets()} for a target \(((l_1, l_2), (u_1, u_2), b)\). We set a size goal \(g\), such that \((u - l) \leq g \ast (u_1 - l_2)\). Normally \(g\) is 256/255, but if \texttt{precise} is True, then \(g\) is 65536/65535.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([1000000, -2000000, 3000000, -4000000, -5000000, -6000000])
sage: ctx = mk_context()
sage: bps = split_for_targets(ctx, bp, [(rr_gap(1/1234567893, 1/1234567892, 1), rr_gap(1/1234567891, 1/1234567890, 1), 12), (rr_gap(1/3, 1/2, -1), rr_gap(2/3, 3/4, -1), 6)])
sage: bps[0]
<IBP: (999992, 999992, 999992) + [0 .. 15) over [86133974711464784778830327/106338239627932698323045648224756608 .. 5919081680259349481383652749593829787/730750818665451459101842163581459827966271488]; level 2; slope_err 0.?e12>
sage: bps[1]
<IBP: (-1562500, -1875001, -2222223, -2592593, -2969137, -3337450) + [0 .. 4) over [1/2 .. 2863311531/4294967296]>
```

```python
sage.rings.polynomial.real_roots.subsample_vec_doctest(a, slen, llen)
sage.rings.polynomial.real_roots.taylor_shift1_intvec(c)
```

Given a vector of integers \(c\) of length \(d+1\), representing the coefficients of a degree-\(d\) polynomial \(p\), modify the vector to perform a Taylor shift by 1 (that is, \(p\) becomes \(p(x+1)\)).

This is the straightforward algorithm, which is not asymptotically optimal.

Modifies the input vector; has no return value.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: p = (x-1)*(x-2)*(x-3)
sage: v = vector(ZZ, p.list())
sage: p, v
(x^3 - 6*x^2 + 11*x - 6, (-6, 11, -6, 1))
sage: taylor_shift1_intvec(v)
sage: p(x+1), v
(x^3 - 3*x^2 + 2*x, (0, 2, -3, 1))
```

```python
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 \texttt{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 \(p\ast scale\). (If you only care about the roots of the polynomial, then of course scale can be ignored.)

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: to_bernstein(x)
([0, 1], 1)
```

(continues on next page)

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)

Bases: object

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
... k. 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

(continues on next page)
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

2.1.19 Isolate Complex Roots of Polynomials

AUTHOR:

• Carl Witty (2007-11-18): initial version

This is an implementation of complex root isolation. That is, given a polynomial with exact complex coefficients, we compute isolating intervals for the complex roots of the polynomial. (Polynomials with integer, rational, Gaussian rational, or algebraic coefficients are supported.)

We use a simple algorithm. First, we compute a squarefree decomposition of the input polynomial; the resulting polynomials have no multiple roots. Then, we find the roots numerically, using NumPy (at low precision) or Pari (at high precision). Then, we verify the roots using interval arithmetic.

EXAMPLES:

```python
sage: x = polygen(ZZ)
sage: (x^5 - x - 1).roots(ring=CIF)
[(1.167303978261419?, 1), (-0.764884433600585? - 0.352471546031727?*I, 1), (-0.764884433600585? + 0.352471546031727?*I, 1), (0.181232444469876? - 1.083954101317711?*I, 1), (0.181232444469876? + 1.083954101317711?*I, 1)]
```

sage.rings.polynomial.complex_roots.complex_roots(p, skip_squarefree=False, retval='interval', min_prec=0)

Compute the complex roots of a given polynomial with exact coefficients (integer, rational, Gaussian rational, and algebraic coefficients are supported). Returns a list of pairs of a root and its multiplicity.

Roots are returned as a ComplexIntervalFieldElement; each interval includes exactly one root, and the intervals are disjoint.

By default, the algorithm will do a squarefree decomposition to get squarefree polynomials. The skip_squarefree parameter lets you skip this step. (If this step is skipped, and the polynomial has a repeated root, then the algorithm will loop forever!)

You can specify retval='interval' (the default) to get roots as complex intervals. The other options are retval='algebraic' to get elements of QQbar, or retval='algebraic_real' to get only the real roots, and to get them as elements of AA.

EXAMPLES:

```python
sage: from sage.rings.polynomial.complex_roots import complex_roots
sage: x = polygen(ZZ)
```
sage: complex_roots(x^5 - x - 1)
[(-1.167303978261419?, 1), (-0.76488443600585? - 0.352471546031727?*I, 1), (-0.76488443600585? + 0.352471546031727?*I, 1), (0.181232444469876? - 1, 0.181232444469876? + 0.352471546031727?*I, 1), (0.183954101317711?*I, 1)]

Unfortunately due to numerical noise there can be a small imaginary part to each root depending on CPU, compiler, etc, and that affects the printing order. So we verify the real part of each root and check that the imaginary part is small in both cases:

```python
sage: v = complex_roots(x^2 + 27*x + 181)

[-14.61803398874989?, -12.3819660112501?...]
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
```

This polynomial actually has all-real coefficients, and is very, very close to (x-1)^5:

```python
sage: [RR(QQ(a)) for a in list(p - (x-1)^5)]
[3.87259191484932e-121, -3.87259191484932e-121]
sage: rms = complex_roots(p)
sage: [ComplexIntervalField(10)(rt[0] - 1) for rt in rms]
```

We can get roots either as intervals, or as elements of QQbar or AA.

```python
sage: p = (x^2 + x - 1)
sage: p = p * p(x*im)
sage: p
-x^4 + (im - 1)*x^3 + im*x^2 + (-im - 1)*x + 1
```

Two of the roots have a zero real component; two have a zero imaginary component. These zero components will be found slightly inaccurately, and the exact values returned are very sensitive to the (non-portable) results of NumPy. So we post-process the roots for printing, to get predictable doctest results.

```python
sage: def tiny(x):
....:     return x.contains_zero() and x.absolute_diameter() < 1e-14
sage: def smash(x):
....:     x = CIF(x[0]) # discard multiplicity
....:     if tiny(x.imag()):
....:         return x.real()
....:     if tiny(x.real()):
....:         return CIF(0, x.imag())

sage: rms = complex_roots(p)
sage: [rt for rt in sorted(map(smash, rms))]
[-1.618033988749895?, -0.618033988749895?*I, 1.618033988749895?*I, 0.618033988749895?]
```
sage: rts = complex_roots(p, retval='algebraic_real'); type(rts[0][0]), rts
(<class 'sage.rings.qqbar.AlgebraicReal'>, \((-1.618033988749895?, 1), (0.618033988749895?, 1)\))

sage.rings.polynomial.complex_roots.interval_roots\(\) 
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 \(\text{prec}\). 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.
  →86602540378443864676372317052936183471402626905190314027904?*I]
sage: interval_roots(p, rts, 200)
[1, -0.500000000000000000000000000000000000000000000000000000000000? + 0.
  →86602540378443864676372317052936183471402626905190314027904?*I, -0.
  →500000000000000000000000000000000000000000000000000000000000? - 0.
  →86602540378443864676372317052936183471402626905190314027904?*I]

sage.rings.polynomial.complex_roots.intervals_disjoint\(\) 
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

2.1.20 Refine polynomial roots using Newton–Raphson

This is an implementation of the Newton–Raphson algorithm to approximate roots of complex polynomials. The implementation is based on interval arithmetic

AUTHORS:

• Carl Witty (2007-11-18): initial version

sage.rings.polynomial.refine_root.refine_root\(\) 
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.refine_root import refine_root
sage: x = polygen(ZZ)
sage: p = x^9 - 1
sage: ip = CIF['x'](p); ip
x^9 - 1
sage: ipd = CIF['x'](p.derivative()); ipd
9*x^8
sage: irt = CIF(CC(cos(2*pi/9), sin(2*pi/9))); irt
0.7660444311897802? + 0.64278760968653926?*I
sage: ip(irt)
0.?e-14 + 0.?e-14*I
sage: ipd(irt)
6.89439998807080? - 5.78508848717885?*I
sage: refine_root(ip, ipd, irt, CIF)
0.766044443118978? + 0.642787609686540?*I

2.1.21 Ideals in Univariate Polynomial Rings.

AUTHORS:

• David Roe (2009-12-14) – initial version.

class sage.rings.polynomial.ideal.Ideal_1poly_field(ring, gen)

Bases: sage.rings.ideal.Ideal_pid

An ideal in a univariate polynomial ring over a field.

groebner_basis(algorithm=None)

Return a Gröbner basis for this ideal.

The Gröbner basis has 1 element, namely the generator of the ideal. This trivial method exists for compatibility with multi-variate polynomial rings.

INPUT:

• algorithm – ignored

EXAMPLES:

sage: R.<x> = QQ[]
sage: I = R.ideal([x^2 - 1, x^3 - 1])
sage: G = I.groebner_basis(); G
[x - 1]
sage: type(G)
<class 'sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_˓
generic'>
sage: list(G)
[x - 1]

residue_class_degree()

Returns the degree of the generator of this ideal.

This function is included for compatibility with ideals in rings of integers of number fields.

EXAMPLES:

sage: R.<t> = GF(5)[]
sage: P = R.ideal(t^4 + t + 1)
residue_field \((\text{names=}\text{None, check=True})\)

If this ideal is \(P \subset \mathbb{F}_p[t]\), returns the quotient \(\mathbb{F}_p[t]/P\).

EXAMPLES:

```python
sage: R.<t> = GF(17)[]; P = R.ideal(t^3 + 2*t + 9)
sage: k.<a> = P.residue_field(); k
Residue field in a of Principal ideal (t^3 + 2*t + 9) of Univariate Polynomial Ring in t over Finite Field of size 17
```

## 2.1.22 Quotients of Univariate Polynomial Rings

EXAMPLES:

```python
sage: R.<x> = QQ[]
sage: S = R.quotient(x^3-3*x+1, 'alpha')
sage: S.gen()**2 in S
True
sage: x in S
True
sage: S.gen() in R
False
sage: 1 in S
True
```

class sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRingFactory

Bases: sage.structure.factory.UniqueFactory

Create a quotient of a polynomial ring.

INPUT:

- ring - a univariate polynomial ring
- polynomial - an element of ring with a unit leading coefficient
- names - (optional) name for the variable

OUTPUT: Creates the quotient ring \(R/I\), where \(R\) is the ring and \(I\) is the principal ideal generated by polynomial.

EXAMPLES:

We create the quotient ring \(\mathbb{Z}[x]/(x^3 + 7)\), and demonstrate many basic functions with it:

```python
sage: Z = IntegerRing()
sage: R = PolynomialRing(Z, 'x'); x = R.gen()
sage: S = R.quotient(x^3 + 7, 'a'); a = S.gen()
sage: S
Univariate Quotient Polynomial Ring in a over Integer Ring with modulus x^3 + 7
sage: a^3
-7
sage: S.is_field()
False
sage: a in S
```

(continues on next page)
We create the “iterated” polynomial ring quotient

\[ R = \left( \mathbb{F}_2[y]/(y^2 + y + 1) \right)[x]/(x^3 - 5). \]

Next we create a number field, but viewed as a quotient of a polynomial ring over \( \mathbb{Q} \):

There are conversion functions for easily going back and forth between quotients of polynomial rings over \( \mathbb{Q} \) and number fields:

The leading coefficient must be a unit (but need not be 1).
Traceback (most recent call last):
...
TypeError: polynomial must have unit leading coefficient

Another example:

```
sage: R.<x> = PolynomialRing(IntegerRing())
sage: f = x^2 + 1
sage: R.quotient(f)
Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 + 1
```

This shows that the issue at trac ticket #5482 is solved:

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

create_key (ring, polynomial, names=None)

Return a unique description of the quotient ring specified by the arguments.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: PolynomialQuotientRing.create_key(R, x + 1)
(Univariate Polynomial Ring in x over Rational Field, x + 1, ('xbar',))
```

create_object (version, key)

Return the quotient ring specified by key.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: PolynomialQuotientRing.create_object((8, 0, 0), (R, x^2 - 1, ('xbar')))
Univariate Quotient Polynomial Ring in xbar over Rational Field with modulus x^2 - 1
```

class sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_coercion

```
Bases: sage.structure.coerce_maps.DefaultConvertMap_unique

A coercion map from a PolynomialQuotientRing to a PolynomialQuotientRing that restricts to the coercion map on the underlying ring of constants.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: S.<x> = QQ[]
sage: f = S.quo(x^2 + 1).coerce_map_from(R.quo(x^2 + 1)); f
Coercion map:
  From: Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 + 1
  To:   Univariate Quotient Polynomial Ring in xbar over Rational Field with modulus x^2 + 1
```

is_injective ()

Return whether this coercion is injective.

EXAMPLES:
If the modulus of the domain and the codomain is the same and the leading coefficient is a unit in the domain, then the map is injective if the underlying map on the constants is:

```sage
r.<x> = ZZ[]
s.<x> = QQ[]
f = S.quo(x^2 + 1).coerce_map_from(R.quo(x^2 + 1))
f.is_injective()
True
```

**is_surjective()**
Return whether this coercion is surjective.

**EXAMPLES:**
If the underlying map on constants is surjective, then this coercion is surjective since the modulus of the codomain divides the modulus of the domain:

```sage
r.<x> = ZZ[]
f = R.quo(x).coerce_map_from(R.quo(x^2))
f.is_surjective()
True
```

If the modulus of the domain and the codomain is the same, then the map is surjective iff the underlying map on the constants is:

```sage
A.<a> = ZqCA(9)
r.<x> = A[]
s.<x> = A.fraction_field()[]
f = S.quo(x^2 + 2).coerce_map_from(R.quo(x^2 + 2))
f.is_surjective()
False
```

```sage
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

EXAM...e:
s.<x> = PolynomialRing(ZZ)
s.<xbar> = R.quotient(x^2 + 1)
s.<xbar>
Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 + 1
loads(S.dumps()) == S
True
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
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
sage: R.<x> = GF(25,'f25')['x']
sage: S.<a> = R.quo(x^3 - 2)
sage: F, g, h = S.field_extension('b')
Traceback (most recent call last):
...
AttributeError: 'PolynomialQuotientRing_generic_with_category' object has no
˓attribute 'field_extension'
```

Over a finite field, the corresponding field extension is not a number field:

```sage
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
sage: R.<x> = QQ['x']
sage: K.<a> = NumberField(x^3 - 2)
sage: S.<X> = K['X']
sage: Q.<b> = S.quo(X^3 + 2*X + 1)
sage: Q.field_extension('b')
(Number Field in b with defining polynomial X^3 + 2*X + 1 over its base field,
  Defn: b |--> b, Relative number field morphism:
  From: Number Field in b with defining polynomial X^3 + 2*X + 1 over its
  base field
  To:   Univariate Quotient Polynomial Ring in b over Number Field in a with
  defining polynomial x^3 - 2 with modulus X^3 + 2*X + 1
  Defn: b |--> b
  a |--> a)
```

We slightly change the example above so it works.
```python
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)
```

```python
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:
```
```
class sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic(ring, polynomial, name=None, category=None)

Bases: sage.rings.ring.CommutativeRing

Quotient of a univariate polynomial ring by an ideal.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(8)); R
Univariate Polynomial Ring in x over Ring of integers modulo 8
sage: S.<xbar> = R.quotient(x^2 + 1); S
Univariate Quotient Polynomial Ring in xbar over Ring of integers modulo 8 with modulus x^2 + 1
```

We demonstrate object persistence.

```
sage: loads(S.dumps()) == S
True
sage: loads(xbar.dumps()) == xbar
True
```

We create some sample homomorphisms;

```
sage: R.<x> = PolynomialRing(ZZ)
sage: S = R.quo(x^2-4)
sage: f = S.hom([2])
sage: f
Ring morphism:
  From: Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 - 4
  To:   Integer Ring
  Defn: xbar |--> 2
sage: f(x)
2
sage: f(x^2 - 4)
0
sage: f(x^2)
4
```

Element

alias of sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement

S_class_group (S, proof=True)

If self is an étale algebra $D$ over a number field $K$ (i.e. a quotient of $K[x]$ by a squarefree polynomial) and $S$ is a finite set of places of $K$, return a list of generators of the $S$-class group of $D$.

NOTE:

Since the ideal function behaves differently over number fields than over polynomial quotient rings (the
quotient does not even know its ring of integers), we return a set of pairs \((\text{gen}, \text{order})\), where \(\text{gen}\) is a tuple of generators of an ideal \(I\) and \(\text{order}\) is the order of \(I\) in the \(S\)-class group.

**INPUT:**

- \(S\) - a set of primes of the coefficient ring
- \(\text{proof}\) - if False, assume the GRH in computing the class group

**OUTPUT:**

A list of generators of the \(S\)-class group, in the form \((\text{gen}, \text{order})\), where \(\text{gen}\) is a tuple of elements generating a fractional ideal \(I\) and \(\text{order}\) is the order of \(I\) in the \(S\)-class group.

**EXAMPLES:**

A trivial algebra over \(\mathbb{Q}(\sqrt{-5})\) has the same class group as its base:

```python
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:

```python
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:

```python
sage: K.<a> = QuadraticField(-5)
sage: K.class_group()
Class group of order 2 with structure C2 of Number Field in a with defining polynomial x^2 + 5 with a = 2.236067977499790?*I
sage: R.<x> = K[]
sage: S.<xbar> = R.quotient(x^2 + 23)
sage: S.S_class_group([])
[((2, -a + 1, 1/2*xbar + 1/2, -1/2*a*xbar + 1/2*a + 1), 6)]
sage: S.S_class_group([K.ideal(3, a-1)])
[]
sage: S.S_class_group([K.ideal(2, a+1)])
[]
sage: S.S_class_group([K.ideal(a)])
[((2, -a + 1, 1/2*xbar + 1/2, -1/2*a*xbar + 1/2*a + 1), 6)]
```

Now we take an example over a nontrivial base with two factors, each contributing to the class group:

```python
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 + 31/16*xbar + 31/16,
  -1/16*a*xbar^3 + (1/16*a + 1/8)*xbar^2 - 31/16*a*xbar + 31/16*a + 31/8),
  6),
  ((-1/4*xbar^2 - 23/4,
    (1/8*a - 1/8)*xbar^2 + 23/8*a - 23/8,
    -1/16*xbar^3 - 1/16*xbar^2 - 23/16*xbar - 23/16),
    6))
```

(continues on next page)
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 + 31/16*xbar + 31/16, -1/16*a*xbar^3 + (1/16*a + 1/8)*xbar^2 - 31/16*a*xbar + 31/16*a + 31/8), 6), ((-1/4*xbar^2 - 23/4, (1/8*a - 1/8)*xbar^2 - 23/8*a + 23/8, -1/16*xbar^3 - 1/16*xbar^2 - 23/16*xbar - 23/16, 1/16*a*xbar^3 - (-1/16*a - 1/8)*xbar^2 + 23/16*a*xbar - 23/16*a - 23/8), 2)]
```

Note that all the returned values live where we expect them to:

```
sage: CG = S.S_class_group([])
sage: type(CG[0][0][1])
<class 'sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic_with_category.element_class'>
sage: type(CG[0][1])
<type 'sage.rings.integer.Integer'>
```

**S_units** $(S, proof=True)$

If self is an étale algebra $D$ over a number field $K$ (i.e. a quotient of $K[x]$ by a squarefree polynomial) and $S$ is a finite set of places of $K$, return a list of generators of the group of $S$-units of $D$.

**INPUT:**

- **$S$** - a set of primes of the base field
- **proof** - if False, assume the GRH in computing the class group

**OUTPUT:**

A list of generators of the $S$-unit group, in the form $(\text{gen}, \text{order})$, where gen is a unit of order order.

**EXAMPLES:**

```
sage: K.<a> = QuadraticField(-3)
sage: u = K.unit_group()
Unit group with structure C6 of Number Field in a with defining polynomial x^2 + 3 with a = 1.732050807568878?*I
sage: u, o = K.S_units([[]][0]); o
6
sage: 2*u - 1 in {a, -a}
True
sage: u^6
1
sage: u^3
-1
sage: 2*u^2 + 1 in {a, -a}
True
```
```python
sage: K.<a> = QuadraticField(-3)
sage: y = polygen(K)
sage: L.<b> = K['y'].quotient(y^3 + 5); L
Univariate Quotient Polynomial Ring in b over Number Field in a with defining polynomial x^2 + 3 with a = 1.732050807568878?*I with modulus y^3 + 5
sage: [u for u, o in L.S_units([]) if o is Infinity]
[(-1/3*a - 1)*b^2 - 4/3*a*b - 5/6*a + 7/2,
 2/3*a+b^2 + (2/3*a - 2)*b - 5/6*a - 7/2]
sage: [u for u, o in L.S_units([K.ideal(1/2*a - 3/2)]) if o is Infinity]
[(-1/6*a - 1/2)*b^2 - (1/3*a - 1)*b + 4/3*a,
 2/3*a+b^2 + (2/3*a - 2)*b - 5/6*a - 7/2]
sage: [u for u, o in L.S_units([K.ideal(2)]) if o is Infinity]
[(1/2*a - 1/2)*b^2 + (a + 1)*b + 3,
 (1/6*a + 1/2)*b^2 + (-1/3*a + 1)*b - 5/6*a + 1/2,
 (1/6*a + 1/2)*b^2 + (-1/3*a + 1)*b - 5/6*a - 1/2,
 (1/6*a + 1/2)*b^2 - 4/3*a*b - 5/6*a + 7/2,
 2/3*a+b^2 + (2/3*a - 2)*b - 5/6*a - 7/2]
```

Note that all the returned values live where we expect them to:

```python
sage: U = L.S_units([])
sage: type(U[0][0])
<class 'sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_field_with_category.element_class'>
sage: type(U[0][1])
<type 'sage.rings.integer.Integer'>
sage: type(U[1][1])
<class 'sage.rings.infinity.PlusInfinity'>
```

### ambient()

**base_ring()**

Return the base ring of the polynomial ring, of which this ring is a quotient.

**EXAMPLES:**

The base ring of \( \mathbb{Z}[z]/(z^3 + z^2 + z + 1) \) is \( \mathbb{Z} \).

```python
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.

```python
sage: T.<t> = PolynomialRing(S)
sage: W = T.quotient(t^99 + 99)
sage: W.base_ring()
Univariate Quotient Polynomial Ring in beta over Integer Ring with modulus z^99 + z^98 + z^97 + \ldots + z^3 + z^2 + z + 1
```

### cardinality()

Return the number of elements of this quotient ring.

order is an alias of cardinality.

**EXAMPLES:**
characteristic()

Return the characteristic of this quotient ring.

This is always the same as the characteristic of the base ring.

EXAMPLES:

```python
sage: R.<z> = PolynomialRing(ZZ)
sage: S.<a> = R.quo(z - 19)
sage: S.characteristic()
0
```

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 \((\text{gen}, \text{order})\), where \text{gen} is a tuple of generators of an ideal \( I \) and \text{order} is the order of \( I \) in the class group.

INPUT:

- proof - if False, assume the GRH in computing the class group

OUTPUT:

A list of pairs \((\text{gen}, \text{order})\), where \text{gen} is a tuple of elements generating a fractional ideal and \text{order} is the order of \( I \) in the class group.

EXAMPLES:

```python
sage: K.<a> = QuadraticField(-3)
sage: K.class_group()
Class group of order 1 of Number Field in a with defining polynomial x^2 + 3
with a = 1.732050807568878?*I
```
A trivial algebra over $\mathbb{Q}(\sqrt{-5})$ has the same class group as its base:

```python
sage: K.<a> = QuadraticField(-5)
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)]
```

The same algebra constructed in a different way:

```python
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:

```python
sage: K.<a> = QuadraticField(-5)
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),
 ((-1/2*xbar^2 - 23/12, -1/48*xbar^3 - 1/48*xbar^2 - 23/48*xbar - 23/48), 5),
 ((-5/4*xbar^2 - 115/4, 1/4*a*xbar^2 + 23/4*a), 6)]
```

Here is an example of a product of number fields, both of which contribute to the class group:

```python
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, (-1/8*a + 1/8)*xbar^2 - 31/8*a + 31/8), 3),
 ((-1/4*xbar^2 - 23/4, (1/8*a - 1/8)*xbar^2 + 23/8*a - 23/8), 5),
 ((-5/4*xbar^2 - 115/4, 1/4*a*xbar^2 + 23/4*a), 6)]
```

Now we take an example over a nontrivial base with two factors, each contributing to the class group:

```python
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 + 31/16*xbar + 31/16, -1/16*a*xbar^3 + (1/16*a + 1/8)*xbar^2 - 31/16*a*xbar + 31/16*a + 31/8), 6),
 ((-1/4*xbar^2 - 23/4, 1/8*a - 1/8)*xbar^2 + 23/8*a - 23/8, -1/16*xbar^3 - 1/16*xbar^2 - 23/16*xbar - 23/16),
 ((1/16*a*xbar^3 + (-1/16*a - 1/8)*xbar^2 + 23/16*a*xbar - 23/16*a - 23/8), 6),
 ((-5/4*xbar^2 - 115/4, 1/4*a*xbar^2 + 23/4*a, 6)),
 ((continues on next page)
Note that all the returned values live where we expect them to:

```python
sage: CG = S.class_group()

sage: type(CG[0][0][1])
<class 'sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing___generic_with_category.element_class'>

sage: type(CG[0][1])
<type 'sage.rings.integer.Integer'>
```

**construction()**

Functorial construction of self

**EXAMPLES:**

```python
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:**

```python
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:**

```python
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:**
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.

**is_field** \((proof=True)\)

Return whether or not this quotient ring is a field.

If proof is True, requires the is_irreducible method of the modulus to be implemented:

```
sage: F1 = R1.quotient_ring(x^2+x+1)
sage: F2.is_field()  # Traceback (most recent call last):
   ...  
   NotImplementedError: can not rewrite Univariate Quotient Polynomial Ring in xbar over Z-adic Field with capped relative precision 20 with modulus (1 + O(2^20))*x^2 + (1 + O(2^20))*x + 1 + O(2^20) as an isomorphic ring
sage: F2.is_field(proof = False)
False
```
is_finite()
Return whether or not this quotient ring is finite.

EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: R.quotient(x^2).is_finite()  # Not finite
True
sage: R.quotient(x^3-2).is_finite()  # Finite
False

sage: R.<x> = GF(9,'a')[

```

```python
sage: R.quotient(2*x^3+x+1).is_finite()  # Finite
True
sage: R.quotient(2).is_finite()  # Infinite
True

sage: P.<v> = GF(2)[

```

```python
sage: P.quotient(v^2-v).is_finite()  # Finite
True
```

krull_dimension()

lift(x)
Return an element of the ambient ring mapping to the given argument.

EXAMPLES:

```python
sage: P.<x> = QQ[

```

```python
sage: Q = P.quotient(x^2+2)
sage: Q.lift(Q.0^3)
-2*x
sage: Q(-2*x)
-2*xbar
sage: Q.0^3
-2*xbar
```

modulus()
Return the polynomial modulus of this quotient ring.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(GF(3))

```

```python
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:

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

```

```python
sage: S.<y> = PolynomialRing(R)
sage: T.<z> = S.quotient(y + x)
sage: T
Univariate Quotient Polynomial Ring in z over Univariate Polynomial Ring in x
˓
over Rational Field with modulus y + x
```

(continues on next page)
number_field()
Return the number field isomorphic to this quotient polynomial ring, if possible.

EXAMPLES:

sage: R.<x> = PolynomialRing(QQ)
sage: S.<alpha> = R.quotient(x^29 - 17*x - 1)
sage: K = S.number_field()
sage: K
Number Field in alpha with defining polynomial x^29 - 17*x - 1
sage: alpha = K.gen()
sage: alpha^29
17*alpha + 1

order()
Return the number of elements of this quotient ring.

order is an alias of cardinality.

EXAMPLES:

sage: R.<x> = ZZ
sage: R.quo(1).cardinality()
1
sage: R.quo(x^3-2).cardinality()
+Infinity
sage: R.quo(1).order()
1
sage: R.quo(x^3-2).order()
+Infinity

sage: R.<x> = GF(9,'a')

sage: R.quo(2*x^3+x+1).cardinality()
729
sage: GF(9,'a').extension(2*x^3+x+1).cardinality()
729
sage: R.quo(2).cardinality()
1

polynomial_ring()
Return the polynomial ring of which this ring is the quotient.

EXAMPLES:

sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^2-2)
sage: S.polynomial_ring()
Univariate Polynomial Ring in x over Rational Field

random_element (**args, **kwds)
Return a random element of this quotient ring.

INPUT:
*args, **kwds - Arguments for randomization that are passed on to the \texttt{random_element} method of the polynomial ring, and from there to the base ring

**OUTPUT:**
- Element of this quotient ring

**EXAMPLES:**

```python
sage: F1.<a> = GF(2^7)
sage: F1.<x> = F1[]
sage: F2 = F1.extension(x^2+x+1, 'u')
sage: F2.random_element()
(a^6 + a^5 + a^2 + a)*u + a^6 + a^4 + a^3 + a^2 + 1
```

\texttt{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

\texttt{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 \(D(\sqrt[m]{a})/\sqrt[m]{a}\) 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
- \texttt{proof} - if False, assume the GRH in computing the class group

**OUTPUT:**
A list of generators of \(D(S, m)\).

**EXAMPLES:**

```python
sage: K.<a> = QuadraticField(-5)
sage: R.<x> = K[]
sage: D.<T> = R.quotient(x)
sage: D.selmer_group(), 2)
[2, -1]
sage: D.selmer_group([K.ideal(2, -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]
```
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:

- \(\text{proof}\) - if False, assume the GRH in computing the class group

OUTPUT:

A list of generators of the unit group, in the form \((\text{gen}, \text{order})\), where \(\text{gen}\) is a unit of order \(\text{order}\).

EXAMPLES:

```python
sage: K.<a> = QuadraticField(-3)
sage: K.unit_group()
Unit group with structure C6 of Number Field in a with defining polynomial \(x^2 + 3\) with \(a = 1.732050807568878?*I\)
sage: K.<a> = QQ['x'].quotient(x^2 + 3)
sage: u = K.units()[0][0]
sage: 2*u - 1 in {a, -a}
True
sage: u^6
1
sage: u^3
-1
sage: 2*u^2 + 1 in {a, -a}
True
sage: K.<a> = 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

\(\text{polynomial} x^2 + 3\) with \(a = 1.732050807568878?*I\) with modulus \(y^3 + 5\)

sage: [u for u, o in L.units() if o is Infinity]
[(-1/3*a - 1)*b^2 - 4/3*a*b - 5/6*a + 7/2, 2/3*a*b^2 + (2/3*a - 2)*b - 5/6*a - 7/2]
```

Note that all the returned values live where we expect them to:

```python
sage: L.<b> = K['y'].quotient(y^3 + 5)
sage: U = L.units()
sage: type(U[0][0])
<type 'sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_field_with_category.element_class'>
sage: type(U[0][1])
<type 'sage.rings.integer.Integer'>
```

2.1.23 Elements of Quotients of Univariate Polynomial Rings

EXAMPLES: We create a quotient of a univariate polynomial ring over \( \mathbb{Z} \).

\[
\text{sage: } R.<x> = \mathbb{Z}[x]
\]

\[
\text{sage: } S.<a> = R.quotient(x^3 + 3x -1)
\]

\[
\text{sage: } 2 * a^3 -6*a + 2
\]

Next we make a univariate polynomial ring over \( \mathbb{Z}[x]/(x^3 + 3x - 1) \).

\[
\text{sage: } S1.<y> = S[y]
\]

And, we quotient out that by \( y^2 + a \).

\[
\text{sage: } T.<z> = S1.quotient(y^2+a)
\]

In the quotient \( z^2 \) is \(-a\).

\[
\text{sage: } z^2
\]

\[-a
\]

And since \( a^3 = -3x + 1 \), we have:

\[
\text{sage: } z^6
\]

\[3*a - 1
\]

For the purposes of comparison in Sage the quotient element \( a^3 \) is equal to \( x^3 \). This is because when the comparison is performed, the right element is coerced into the parent of the left element, and \( x^3 \) coerces to \( a^3 \).

\[
\text{sage: } a == x
\]

True

\[
\text{sage: } a^3 == x^3
\]

True

\[
\text{sage: } x^3
\]

\[x^3
\]

\[
\text{sage: } S(x^3)
\]

2
AUTHORS:
• William Stein

```python
class sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement


Element of a quotient of a polynomial ring.

EXAMPLES:

```python
code
```

```python
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:

```python
code
```

```python
fcp(var='x')
```

Return the factorization of the characteristic polynomial of this element.

EXAMPLES:

```python
code
```

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:**

```python
sage: R.<x> = QQ[]
sage: S.<y> = R.quotient(x^2 + 2*x + 1)
sage: (2*y).is_unit()
True
sage: (y+1).is_unit()
False
```

**lift()**

Return lift of this polynomial quotient ring element to the unique equivalent polynomial of degree less than the modulus.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3-2)
sage: b = a^2 - 3
sage: b
a^2 - 3
sage: b.lift()
x^2 - 3
```

**list** *(copy=True)*

Return list of the elements of `self`, of length the same as the degree of the quotient polynomial ring.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 + 2*x - 5)
sage: a^10
-134*a^2 - 35*a + 300
sage: (a^10).list()
[300, -35, -134]
```

**matrix()**

The matrix of right multiplication by this element on the power basis for the quotient ring.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 + 2*x - 5)
sage: a.matrix()
[ 0 1 0]
[ 0 0 1]
[ 5 -2 0]
```

**minpoly()**

The minimal polynomial of this element, which is by definition the minimal polynomial of right multiplication by this element.

**norm()**

The norm of this element, which is the determinant of the matrix of right multiplication by this element.
EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 -389*x^2 + 2*x - 5)
sage: a.norm()
5
```

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 -389*x^2 + 2*x - 5)
sage: a.trace()
389
```

**2.1.24 Polynomial Compilers**

AUTHORS:

- Tom Boothby, initial design & implementation
- Robert Bradshaw, bug fixes / suggested & assisted with significant design improvements

**class** `sage.rings.polynomial.polynomial_compiled.CompiledPolynomialFunction`

**Bases:** object

Builds a reasonably optimized directed acyclic graph representation for a given polynomial. A CompiledPolynomialFunction is callable from python, though it is a little faster to call the eval function from pyrex.

This class is not intended to be called by a user, rather, it is intended to improve the performance of immutable polynomial objects.

**Todo:**

- Recursive calling
- Faster casting of coefficients / argument
- Multivariate polynomials
- Cython implementation of Pippenger’s Algorithm that doesn’t depend heavily upon dicts.
- Computation of parameter sequence suggested by Pippenger
- Univariate exponentiation can use Brauer’s method to improve extremely sparse polynomials of very high degree

**class** `sage.rings.polynomial.polynomial_compiled.abc_pd`

**Bases:** `sage.rings.polynomial.polynomial_compiled.binary_pd`

**class** `sage.rings.polynomial.polynomial_compiled.add_pd`

**Bases:** `sage.rings.polynomial.polynomial_compiled.binary_pd`

**class** `sage.rings.polynomial.polynomial_compiled.binary_pd`

**Bases:** `sage.rings.polynomial.polynomial_compiled.generic_pd`

**class** `sage.rings.polynomial.polynomial_compiled.coeff_pd`

**Bases:** `sage.rings.polynomial.polynomial_compiled.generic_pd`

---

Chapter 2. Univariate Polynomials
class sage.rings.polynomial.polynomial_compiled.dummy_pd
    Bases: sage.rings.polynomial.polynomial_compiled.generic_pd

class sage.rings.polynomial.polynomial_compiled.generic_pd
    Bases: object

class sage.rings.polynomial.polynomial_compiled.mul_pd
    Bases: sage.rings.polynomial.polynomial_compiled.binary_pd

class sage.rings.polynomial.polynomial_compiled.pow_pd
    Bases: sage.rings.polynomial.polynomial_compiled.unary_pd

class sage.rings.polynomial.polynomial_compiled.sqr_pd
    Bases: sage.rings.polynomial.polynomial_compiled.unary_pd

class sage.rings.polynomial.polynomial_compiled.unary_pd
    Bases: sage.rings.polynomial.polynomial_compiled.generic_pd

class sage.rings.polynomial.polynomial_compiled.univar_pd
    Bases: sage.rings.polynomial.polynomial_compiled.generic_pd

class sage.rings.polynomial.polynomial_compiled.var_pd
    Bases: sage.rings.polynomial.polynomial_compiled.generic_pd

\section{Polynomial multiplication by Kronecker substitution}

\subsection{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 \texttt{convolution()}.

\begin{example}
\begin{verbatim}
sage: convolution([1, 2, 3, 4, 5], [6, 7])
[6, 19, 32, 45, 58, 35]
\end{verbatim}
\end{example}

The convolution function is reasonably fast, even though it is written in pure Python. For example, the following takes less than a second:

\begin{verbatim}
sage: v = convolution(list(range(1000)), list(range(1000)))
\end{verbatim}

**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\(L1, L2\)
\hspace{1cm} 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)
True
sage: len(L3) == 1000 + 3756 - 1
True
```

2.3 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)
```

Reference:
Bateman, P. T.; Pomerance, C.; Vaughan, R. C. *On the size of the coefficients of the cyclotomic polynomial*.

```
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 Möbius function that is 1 if \( d \) has an even number of distinct prime divisors, -1 if it has an odd number of distinct prime divisors, and 0 if \( d \) is not squarefree.

Multiplications and divisions by polynomials of the form \( 1 - x^n \) can be done very quickly in a single pass.

If sparse is True, the result is returned as a dictionary of the non-zero entries, otherwise the result is returned as a list of python ints.

EXAMPLES:

```
sage: from sage.rings.polynomial.cyclotomic import cyclotomic_coeffs
sage: cyclotomic_coeffs(30)
[1, 1, 0, -1, -1, -1, 0, 1, 1]
sage: cyclotomic_coeffs(10^5)
{0: 1, 10000: -1, 20000: 1, 30000: -1, 40000: 1}
sage: R = QQ['x']
sage: R(cyclotomic_coeffs(30))
x^8 + x^7 - x^5 - x^4 - x^3 + x + 1
```

Check that it has the right degree:

```
sage: euler_phi(30)
8
sage: R(cyclotomic_coeffs(14)).factor()
x^6 - x^5 + x^4 - x^3 + x^2 - x + 1
```
The coefficients are not always +/-1:

```python
sage: cyclotomic_coeffs(105)
[1, 1, 1, 0, 0, -1, -1, -2, -1, -1, 0, 1, 1, 1, 1, 1, 0, -1, 0, -1, 0, -
-1, 0, -1, 0, -1, 0, 0, 1, 1, 1, 1, 1, 0, 0, -1, -2, -1, -1, 0, 0, 1, 1, 1]
```

In fact the height is not bounded by any polynomial in n (Erdos), although takes a while just to exceed linear:

```python
sage: v = cyclotomic_coeffs(1181895)
sage: max(v)
14102773
```

The polynomial is a palindrome for any n:

```python
sage: n = ZZ.random_element(50000)
sage: factor(n)
3 * 10009
sage: v = cyclotomic_coeffs(n, sparse=False)
sage: v == list(reversed(v))
True
```

AUTHORS:

• Robert Bradshaw (2007-10-27): initial version (inspired by work of Andrew Arnold and Michael Mongan)

```python
sage.rings.polynomial.cyclotomic.
cyclotomic_value(n, x)
```

Return the value of the $n$-th cyclotomic polynomial evaluated at $x$.

**INPUT:**

- n – an Integer, specifying which cyclotomic polynomial is to be evaluated.
- x – an element of a ring.

**OUTPUT:**

- the value of the cyclotomic polynomial $\Phi_n$ at $x$.

**ALGORITHM:**

- Reduce to the case that $n$ is squarefree: use the identity
  \[ \Phi_n(x) = \Phi_q(x^{n/q}) \]
  where $q$ is the radical of $n$.
  - Use the identity
    \[ \Phi_n(x) = \prod_{d|n} (x^d - 1)^{\mu(n/d)}, \]
    where $\mu$ is the Möbius function.
  - Handles the case that $x^d = 1$ for some $d$, but not the case that $x^d - 1$ is non-invertible: in this case polynomial evaluation is used instead.

**EXAMPLES:**

```python
sage: cyclotomic_value(51, 3)
1282860140677441
sage: cyclotomic_polynomial(51)(3)
1282860140677441
```

2.3. Fast calculation of cyclotomic polynomials 239
It works for non-integral values as well:

```
sage: cyclotomic_value(144, 4/3)
79148745433504023621920372161/79766443076872509863361
sage: cyclotomic_polynomial(144)(4/3)
79148745433504023621920372161/79766443076872509863361
```
3.1 Multivariate Polynomials and Polynomial Rings

Sage implements multivariate polynomial rings through several backends. The most generic implementation uses the classes `sage.rings.polynomial.polydict.PolyDict` and `sage.rings.polynomial.polydict.ETuple` to construct a dictionary with exponent tuples as keys and coefficients as values.

Additionally, specialized and optimized implementations over many specific coefficient rings are implemented via a shared library interface to SINGULAR; and polynomials in the boolean polynomial ring

\[ \mathbb{F}_2[x_1, \ldots, x_n]/(x_1^2 + x_1, \ldots, x_n^2 + x_n) \]

are implemented using the PolyBoRi library (cf. `sage.rings.polynomial.pbori`).

### 3.1.1 Term orders

Sage supports the following term orders:

**Lexicographic (lex)** \( x^a < x^b \) if and only if there exists \( 1 \leq i \leq n \) such that \( a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i < b_i \). This term order is called ‘lp’ in Singular.

**Degree reverse lexicographic (degrevlex)** Let \( \deg(x^a) = a_1 + a_2 + \cdots + a_n \), then \( x^a < x^b \) if and only if \( \deg(x^a) < \deg(x^b) \) or \( \deg(x^a) = \deg(x^b) \) and there exists \( 1 \leq i \leq n \) such that \( a_i = b_i, \ldots, a_{i+1} = b_{i+1}, a_i > b_i \). This term order is called ‘dp’ in Singular.

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
sage: x > y
True
sage: x > y^2
True
sage: x > 1
True
sage: x^1*y^2 > y^3*z^4
True
sage: x^3*y^2*z^4 < x^3*y^2*z^1
False
```

```python
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='degrevlex')
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 lexicographic (deglex) Let $\deg(x^a) = a_1 + a_2 + \cdots + a_n$, then $x^a < x^b$ if and only if $\deg(x^a) < \deg(x^b)$ or $\deg(x^a) = \deg(x^b)$ and there exists $1 \leq i \leq n$ such that $a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i < b_i$. This term order is called ‘Dp’ in Singular.

EXAMPLES:

```python
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^4*y^1*z^3 > x^3*y^2*z^0  # True
sage: x^2*y*z^2 > x*y^3*z  # False
```

Inverse lexicographic (invlex) $x^a < x^b$ if and only if there exists $1 \leq i \leq n$ such that $a_n = b_n, \ldots, a_{i+1} = b_{i+1}, a_i < b_i$. This order is called ‘rp’ in Singular.

EXAMPLES:

```python
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='invlex')
sage: x > y  # False
sage: y > x^2  # True
sage: x > 1  # True
sage: x*y > z  # False
```

This term order only makes sense in a non-commutative setting because if P is the ring $k[x_1, \ldots, x_n]$ and term order ‘invlex’ then it is equivalent to the ring $k[x_n, \ldots, x_1]$ with term order ‘lex’.

Negative lexicographic (neglex) $x^a < x^b$ if and only if there exists $1 \leq i \leq n$ such that $a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i > b_i$. This term order is called ‘ls’ in Singular.

EXAMPLES:

```python
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='neglex')
sage: x > y  # False
sage: x > 1  # False
```

(continues on next page)
Negative degree reverse lexicographic (negdegrevlex)  Let \( \deg(x^a) = a_1 + a_2 + \cdots + a_n \), then \( x^a < x^b \) if and only if \( \deg(x^a) > \deg(x^b) \) or \( \deg(x^a) = \deg(x^b) \) and there exists \( 1 \leq i \leq n \) such that \( a_n = b_n, \ldots, a_{i+1} = b_{i+1}, a_i > b_i \). This term order is called ‘ds’ in Singular.

**EXAMPLES:**

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='negdegrevlex')
sage: x > y
True
sage: x > x^2
True
sage: x > 1
False
sage: x^1*y^2 > y^3*z^4
True
sage: x^2*y*z^2 > x*y^3*z
False
```

Negative degree lexicographic (negdeglex)  Let \( \deg(x^a) = a_1 + a_2 + \cdots + a_n \), then \( x^a < x^b \) if and only if \( \deg(x^a) > \deg(x^b) \) or \( \deg(x^a) = \deg(x^b) \) and there exists \( 1 \leq i \leq n \) such that \( a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i < b_i \). This term order is called ‘Ds’ in Singular.

**EXAMPLES:**

```
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 \( \deg_w(x^a) = a_1w_1 + a_2w_2 + \cdots + a_nw_n \) with weights \( w \), then \( x^a < x^b \) if and only if \( \deg_w(x^a) > \deg_w(x^b) \) or \( \deg_w(x^a) = \deg_w(x^b) \) and there exists \( 1 \leq i \leq n \) such that \( a_n = b_n, \ldots, a_{i+1} = b_{i+1}, a_i > b_i \). 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
```

(continues on next page)
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 \( 1 \leq i \leq n \) such that \( a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i < b_i \). This term order is called ‘Wp’ in Singular.

EXAMPLES:

```
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 \( \deg_w(x^a) = a_1 w_1 + a_2 w_2 + \cdots + a_n w_n \) with weights \( w \), then \( x^a < x^b \) if and only if \( \deg_w(x^a) > \deg_w(x^b) \) or \( \deg_w(x^a) = \deg_w(x^b) \) and there exists \( 1 \leq i \leq n \) such that \( a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i > b_i \). This term order is called ‘ws’ in Singular.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order=TermOrder('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 \( 1 \leq i \leq n \) such that \( a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i > b_i \). This term order is called ‘dp_asc’ in PolyBoRi. Singular has the extra weight vector ordering ‘(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 \( \deg_w(x^a) = a_1 w_1 + a_2 w_2 + \cdots + a_n w_n \) with weights \( w \), then \( x^a < x^b \) if and only if \( \deg_w(x^a) > \deg_w(x^b) \) or \( \deg_w(x^a) = \deg_w(x^b) \) and there exists \( 1 \leq i \leq n \) such that \( a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i < b_i \). This term order is called ‘Ws’ in Singular.
EXAMPLES:

```python
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order=TermOrder('negdeglex',(1,2,3)))
sage: x > y
True
sage: x > x^2
True
sage: x > 1
False
sage: x^1*y^2 > x^2*z
False
sage: y*z > x^3*y
False
```

Of these, only ‘degrevlex’, ‘deglex’, ‘degneglex’, ‘wdegrevlex’, ‘wdeglex’, ‘invlex’ and ‘lex’ are global orders.

Sage also supports matrix term order. Given a square matrix $A$, $x^a <_A x^b$ if and only if $Aa < Ab$

where $<$ is the lexicographic term order.

EXAMPLES:

```python
sage: m = matrix(2,[[2,3,0,1]]); m
[2 3]
[0 1]
sage: T = TermOrder(m); T
Matrix term order with matrix
[2 3]
[0 1]
sage: P.<a,b> = PolynomialRing(QQ,2,order=T)
sage: P
Multivariate Polynomial Ring in a, b over Rational Field
sage: a > b
False
sage: a^3 < b^2
True
sage: S = TermOrder('M(2,3,0,1)')
sage: T == S
True
```

Additionally all these monomial orders may be combined to product or block orders, defined as:

Let $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_m)$ be two ordered sets of variables, $<_1$ a monomial order on $k[x]$ and $<_2$ a monomial order on $k[y]$.

The product order (or block order) $< := (<_1, <_2)$ on $k[x,y]$ is defined as: $x^a y^b < x^A y^B$ if and only if $x^a <_1 x^A$ or ($x^a = x^A$ and $y^b <_2 y^B$).

These block orders are constructed in Sage by giving a comma separated list of monomial orders with the length of each block attached to them.

EXAMPLES:

As an example, consider constructing a block order where the first four variables are compared using the degree reverse lexicographical order while the last two variables in the second block are compared using negative lexicographical order.

```python
sage: P.<a,b,c,d,e,f> = PolynomialRing(QQ, 6,order='degrevlex(4),neglex(2)')
sage: a > c^4
```

(continues on next page)
False
\begin{Verbatim}
sage: a > e^4
True
sage: e > f^2
False
\end{Verbatim}
The same result can be achieved by:
\begin{Verbatim}
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
\end{Verbatim}
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:
\begin{Verbatim}
sage: T = TermOrder("royalorder")
Traceback (most recent call last):
  ... ValueError: unknown term order 'royalorder'
sage: T = TermOrder("royalorder",force=True)
sage: T
royalorder term order
sage: T.singular_str()
'royalorder'
\end{Verbatim}

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

\begin{Verbatim}
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.

EXAMPLES:
sage: t = TermOrder('deglex', 2) + TermOrder('lex', 2)
sage: t.blocks()

(Degree lexicographic term order, Lexicographic term order)

greater_tuple
The default greater_tuple method for this term order.

EXAMPLES:

```python
sage: O = TermOrder()
sage: O.greater_tuple.__func__ is O.greater_tuple_lex.__func__
True
sage: O = TermOrder('deglex')
sage: O.greater_tuple.__func__ is O.greater_tuple_deglex.__func__
True
```

greater_tuple_block \((f, g)\)
Return the greater exponent tuple with respect to the block order as specified when constructing this element.

This method is called by the \texttt{lm/lc/lt} methods of \texttt{MPolynomial_polydict}.

INPUT:
- \(f\) - exponent tuple
- \(g\) - exponent tuple

EXAMPLES:

```python
sage: P.<a,b,c,d,e,f> = PolynomialRing(QQbar, 6, order='degrevlex(3),
˓→degrevlex(3)')
sage: f = a + c^4; f.lm() # indirect doctest
c^4
sage: g = a + e^4; g.lm()
a
```

greater_tuple_deglex \((f, g)\)
Return the greater exponent tuple with respect to the total degree lexicographical term order.

INPUT:
- \(f\) - exponent tuple
- \(g\) - exponent tuple

EXAMPLES:

```python
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 \texttt{lm/lc/lt} methods of \texttt{MPolynomial_polydict}.

greater_tuple_degneglex \((f, g)\)
Return the greater exponent tuple with respect to the degree negative lexicographical term order.

INPUT:
- \(f\) - exponent tuple
- \(g\) - exponent tuple

EXAMPLES:

```python
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
```

3.1. Multivariate Polynomials and Polynomial Rings 247
• g - exponent tuple

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='degrevlex')
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

EXAMPLES:

```
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
```

Chapter 3. Multivariate Polynomials
This method is called by the \texttt{lm/lc/lt} methods of \texttt{MPolynomial\_polydict}.

\textbf{greater\_tuple\_matrix}(f, g)

Return the greater exponent tuple with respect to the matrix term order.

INPUT:

- \texttt{f} - exponent tuple
- \texttt{g} - exponent tuple

EXAMPLES:

\begin{verbatim}
 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
\end{verbatim}

\textbf{greater\_tuple\_negdeglex}(f, g)

Return the greater exponent tuple with respect to the negative degree lexicographical term order.

INPUT:

- \texttt{f} - exponent tuple
- \texttt{g} - exponent tuple

EXAMPLES:

\begin{verbatim}
 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
\end{verbatim}

This method is called by the \texttt{lm/lc/lt} methods of \texttt{MPolynomial\_polydict}.

\textbf{greater\_tuple\_negdegrevlex}(f, g)

Return the greater exponent tuple with respect to the negative degree reverse lexicographical term order.

INPUT:

- \texttt{f} - exponent tuple
- \texttt{g} - exponent tuple

EXAMPLES:

\begin{verbatim}
 sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='negdegrevlex')
 sage: f = x + y; f.lm() # indirect doctest
 x
 sage: f = x + x^2; f.lm()
 x
 sage: f = x^2*y*z^2 + x*y^3*z; f.lm()
 x*y^3*z
\end{verbatim}

This method is called by the \texttt{lm/lc/lt} methods of \texttt{MPolynomial\_polydict}.

\textbf{greater\_tuple\_neglex}(f, g)

Return the greater exponent tuple with respect to the negative lexicographical term order.
This method is called by the \texttt{lm/lc.lt} methods of \texttt{MPolynomial_polydict}.

INPUT:
- \( f \) - exponent tuple
- \( g \) - exponent tuple

EXAMPLES:

```python
tsage: R.<a,b,c,d,e,f> = PolynomialRing(QQbar, 6, order='degrevlex(3), negdeglex(3))
tsage: f = a + c^4; f.lm() # indirect doctest
c^4
tsage: g = a + e^4; g.lm()
a```

\textbf{greater\_tuple\_negwdeglex}(\( f, g \))

Return the greater exponent tuple with respect to the negative weighted degree lexicographical term order.

INPUT:
- \( f \) - exponent tuple
- \( g \) - exponent tuple

EXAMPLES:

```python
tsage: t = TermOrder('negwdeglex', (1,2,3))
tsage: R.<x,y,z> = PolynomialRing(QQbar, 3, order=t)
tsage: f = x + y; f.lm() # indirect doctest
x
tsage: f = x + x^2; f.lm()
x
tsage: f = x^3 + z; f.lm()
x^3```

This method is called by the \texttt{lm/lc.lt} methods of \texttt{MPolynomial_polydict}.

\textbf{greater\_tuple\_negwdegrevlex}(\( f, g \))

Return the greater exponent tuple with respect to the negative weighted degree reverse lexicographical term order.

INPUT:
- \( f \) - exponent tuple
- \( g \) - exponent tuple

EXAMPLES:

```python
tsage: t = TermOrder('negwdegrevlex', (1,2,3))
tsage: R.<x,y,z> = PolynomialRing(QQbar, 3, order=t)
tsage: f = x + y; f.lm() # indirect doctest
x
tsage: f = x + x^2; f.lm()
x
tsage: f = x^3 + z; f.lm()
x^3```

This method is called by the \texttt{lm/lc.lt} methods of \texttt{MPolynomial_polydict}.
**greater_tuple_wdeglex** \((f, g)\)
Return the greater exponent tuple with respect to the weighted degree lexicographical term order.

**INPUT:**
- \(f\) - exponent tuple
- \(g\) - exponent tuple

**EXAMPLES:**
```python
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 \texttt{MPolynomial_polydict}.

**greater_tuple_wdegrevlex** \((f, g)\)
Return the greater exponent tuple with respect to the weighted degree reverse lexicographical term order.

**INPUT:**
- \(f\) - exponent tuple
- \(g\) - exponent tuple

**EXAMPLES:**
```python
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 \texttt{MPolynomial_polydict}.

**is_block_order**
Return true if self is a block term order.

**EXAMPLES:**
```python
sage: t = TermOrder('deglex', 2) + TermOrder('lex', 2)
sage: t.is_block_order()
True
```

**is_global**
Return true if this term order is definitely global. Return false otherwise, which includes unknown term orders.

**EXAMPLES:**
```python
sage: T = TermOrder('degrevlex', 3) + TermOrder('negdegrevlex', 3)
sage: T.is_global()
True
```
(continues on next page)
sage: T.is_global()
False
sage: T = TermOrder('degneglex', 3)
sage: T.is_global()
True

\textbf{is\_local()}

Return true if this term order is definitely local. Return false otherwise, which includes unknown term orders.

EXAMPLES:

\begin{verbatim}
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
\end{verbatim}

\textbf{is\_weighted\_degree\_order()}

Return true if self is a weighted degree term order.

EXAMPLES:

\begin{verbatim}
sage: t=TermOrder('wdeglex', (2,3))
sage: t.is_weighted_degree_order()
True
\end{verbatim}

\textbf{macaulay2\_str()}

Return a Macaulay2 representation of self.

Used to convert polynomial rings to their Macaulay2 representation.

EXAMPLES:

\begin{verbatim}
sage: P = PolynomialRing(GF(127), 8,names='x',order='degrevlex(3),lex(5)')
sage: T = P.term_order()
sage: T.macaulay2_str()
'{GRevLex => 3,Lex => 5}'
sage: P._macaulay2_().options()
# optional - macaulay2
{MonomialSize => 16 }
{GRevLex => {1, 1, 1}}
{Lex => 5 }
{Position => Up }
\end{verbatim}

\textbf{magma\_str()}

Return a MAGMA representation of self.

Used to convert polynomial rings to their MAGMA representation.

EXAMPLES:

\begin{verbatim}
sage: P = PolynomialRing(GF(127), 10,names='x',order='degrevlex')
sage: magma(P)
\end{verbatim}
Polynomial ring of rank 10 over GF(127)
Order: Graded Reverse Lexicographical
Variables: x0, x1, x2, x3, x4, x5, x6, x7, x8, x9

```python
sage: T = P.term_order()
sage: T.magma_str()  
"grevlex"
```

**matrix()**

Return the matrix defining matrix term order.

**EXAMPLES:**

```python
sage: t = TermOrder("M(1,2,0,1)")
sage: t.matrix()
[1 2]
[0 1]
```

**name()**

**EXAMPLES:**

```python
sage: TermOrder('lex').name()
'lex'
```

**singular_moreblocks()**

Return a the number of additional blocks SINGULAR needs to allocate for handling non-native orderings like `degneglex`.

**EXAMPLES:**

```python
sage: P = PolynomialRing(GF(127),10,names='x',order='lex(3),deglex(5),lex(2)')
sage: T = P.term_order()
sage: T.singular_moreblocks()
0
sage: P = PolynomialRing(GF(127),10,names='x',order='lex(3),degneglex(5),
    \rightarrow 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)
    \rightarrow')
sage: T = P.term_order()
sage: T.singular_moreblocks()
2
```

**singular_str()**

Return a SINGULAR representation of self.

Used to convert polynomial rings to their SINGULAR representation.

**EXAMPLES:**

```python
sage: P = PolynomialRing(GF(127),10,names='x',order='lex(3),deglex(5),lex(2)')
sage: T = P.term_order()
sage: T.singular_str()  
'(lp(3),Dp(5),lp(2))'
```
polynomial ring, over a field, global ordering
// coefficients: ZZ/127
// number of vars : 10
// block 1 : ordering lp
//   : names x0 x1 x2
// block 2 : ordering Dp
//   : names x3 x4 x5 x6 x7
// block 3 : ordering lp
//   : names x8 x9
// block 4 : ordering C

sortkey
The default sortkey method for this term order.

EXAMPLES:

sage: O = TermOrder()
sage: O.sortkey.__func__ is O.sortkey_lex.__func__
True
sage: O = TermOrder('deglex')
sage: O.sortkey.__func__ is O.sortkey_deglex.__func__
True

sortkey_block(f)
Return the sortkey of an exponent tuple with respect to the block order as specified when constructing this element.

INPUT:

• f – exponent tuple

EXAMPLES:

sage: P.<a,b,c,d,e,f>=PolynomialRing(QQbar, 6, order='degrevlex(3),
˓→degrevlex(3)')
sage: a > c^4 # indirect doctest
False
sage: a > e^4
True

sortkey_deglex(f)
Return the sortkey of an exponent tuple with respect to the degree lexicographical term order.

INPUT:

• f – exponent tuple

EXAMPLES:

sage: P.<x,y> = PolynomialRing(QQbar, 2, order='deglex')
sage: x > y^2 # indirect doctest
False
sage: x > 1
True

sortkey_degneglex(f)
Return the sortkey of an exponent tuple with respect to the degree negative lexicographical term order.

INPUT:
• \( \cdot \) – exponent tuple

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='degneglex')
sage: x*y > y*z  # indirect doctest
False
sage: x*y > x
True
```

**sortkey_degrevlex** \((f)\)

Return the sortkey of an exponent tuple with respect to the degree reversed lexicographical term order.

INPUT:

• \( \cdot \) – exponent tuple

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='degrevlex')
sage: x > y^2  # indirect doctest
False
sage: x > 1
True
```

**sortkey_invlex** \((f)\)

Return the sortkey of an exponent tuple with respect to the inversed lexicographical term order.

INPUT:

• \( \cdot \) – exponent tuple

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='invlex')
sage: x > y^2  # indirect doctest
False
sage: x > 1
True
```

**sortkey_lex** \((f)\)

Return the sortkey of an exponent tuple with respect to the lexicographical term order.

INPUT:

• \( \cdot \) – exponent tuple

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='lex')
sage: x > y^2  # indirect doctest
True
sage: x > 1
True
```

**sortkey_matrix** \((f)\)

Return the sortkey of an exponent tuple with respect to the matrix term order.

INPUT:

• \( \cdot \) – 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

sortkey_negdeglex (f)
Return the sortkey of an exponent tuple with respect to the negative degree lexicographical term order.

INPUT:

• f – exponent tuple

EXAMPLES:

sage: P.<x,y> = PolynomialRing(QQbar, 2, order='negdeglex')
sage: x > y^2  # indirect doctest
True
sage: x > 1
False

sortkey_negdegrevlex (f)
Return the sortkey of an exponent tuple with respect to the negative degree reverse lexicographical term order.

INPUT:

• f – exponent tuple

EXAMPLES:

sage: P.<x,y> = PolynomialRing(QQbar, 2, order='negdegrevlex')
sage: x > y^2  # indirect doctest
True
sage: x > 1
False

sortkey_neglex (f)
Return the sortkey of an exponent tuple with respect to the negative lexicographical term order.

INPUT:

• f – exponent tuple

EXAMPLES:

sage: P.<x,y> = PolynomialRing(QQbar, 2, order='neglex')
sage: x > y^2  # indirect doctest
False
sage: x > 1
False

sortkey_negwdeglex (f)
Return the sortkey of an exponent tuple with respect to the negative weighted degree lexicographical term order.

INPUT:

• f – exponent tuple

EXAMPLES:
sage: t = TermOrder('negwdeglex',(3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)
sage: x > y^2 # indirect doctest
True
sage: x^2 > y^3
True

sortkey_negwdegrevlex \( f \)
Return the sortkey of an exponent tuple with respect to the negative weighted degree reverse lexicographical term order.

INPUT:

• \( f \) – exponent tuple

EXAMPLES:

sage: t = TermOrder('negwdegrevlex',(3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)
sage: x > y^2 # indirect doctest
True
sage: x^2 > y^3
True

sortkey_wdeglex \( f \)
Return the sortkey of an exponent tuple with respect to the weighted degree lexicographical term order.

INPUT:

• \( f \) – exponent tuple

EXAMPLES:

sage: t = TermOrder('wdeglex',(3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)
sage: x > y^2 # indirect doctest
False
sage: x > y
True

sortkey_wdegrevlex \( f \)
Return the sortkey of an exponent tuple with respect to the weighted degree reverse lexicographical term order.

INPUT:

• \( f \) – exponent tuple

EXAMPLES:

sage: t = TermOrder('wdegrevlex',(3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)
sage: x > y^2 # indirect doctest
False
sage: x^2 > y^3
True

tuple_weight \( f \)
Return the weight of tuple \( f \).

INPUT:
• \( f \) - exponent tuple

EXAMPLES:

```python
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.

EXAMPLES:

```python
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.

EXAMPLES:

```python
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)

### 3.1.2 Base class for multivariate polynomial rings

class `sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base`

Bases: `sage.rings.ring.CommutativeRing`

Create a polynomial ring in several variables over a commutative ring.

EXAMPLES:

```python
sage: R.<x,y> = ZZ['x,y']; R
Multivariate Polynomial Ring in x, y over Integer Ring
sage: class CR(CommutativeRing):
....:     def __init__(self):
....:         CommutativeRing.__init__(self, self)
```

(continues on next page)
....:     def __call__(self, x):
....:         return None
sage: cr = CR()
sage: cr.is_commutative()
True
sage: cr['x,y']
Multivariate Polynomial Ring in x, y over <__main__.CR_with_category object at ...>

`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:**

```python
sage: P.<x,y,z> = PolynomialRing(GF(127),3,order='lex')
sage: x > y^2
True
sage: Q.<x,y,z> = P.change_ring(order='degrevlex')
sage: x > y^2
False
```

`characteristic()`

Return the characteristic of this polynomial ring.

**EXAMPLES:**

```python
sage: R = PolynomialRing(QQ, 'x', 3)
sage: R.characteristic()
0
sage: R = PolynomialRing(GF(7),'x', 20)
sage: R.characteristic()
7
```

`completion` *(names, prec=20)*

Return the completion of self with respect to the ideal generated by the variable(s) `names`.

**INPUT:**

- `names` – variable or list/tuple of variables (given either as elements of the polynomial ring or as strings)
- `prec` – default precision of resulting power series ring
- `extras` – deprecated and ignored

**EXAMPLES:**

```python
sage: P.<x,y,z,w> = PolynomialRing(ZZ)
sage: P.completion('w')
```

(continues on next page)
sage: P.completion((w,x,y))
Multivariate Power Series Ring in w, x, y over Univariate Polynomial Ring in z
   → over Integer Ring
sage: Q.<w,x,y,z> = P.completion(); Q
Multivariate Power Series Ring in w, x, y, z over Integer Ring

sage: H = PolynomialRing(PolynomialRing(ZZ,3,'z'),4,'f'); H
Multivariate Polynomial Ring in f0, f1, f2, f3 over Multivariate Polynomial Ring in z0, z1, z2 over Integer Ring

sage: H.completion(H.gens())
Multivariate Power Series Ring in f0, f1, f2, f3 over Multivariate Polynomial Ring in z0, z1, z2 over Integer Ring
sage: H.completion(H.gens()[2])
Power Series Ring in f2 over Multivariate Polynomial Ring in f0, f1, f3 over Multivariate Polynomial Ring in z0, z1, z2 over Integer Ring

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

flattening_morphism()
Return the flattening morphism of this polynomial ring

EXAMPLES:

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

gen(n=0)

irrelevant_ideal()
Return the irrelevant ideal of this multivariate polynomial ring.

This is the ideal generated by all of the indeterminate generators of this ring.

EXAMPLES:

```python
sage: R.<x,y,z> = QQ[]
sage: R.irrelevant_ideal()
Ideal (x, y, z) of Multivariate Polynomial Ring in x, y, z over Rational Field
```

**is_exact()**
Test whether this multivariate polynomial ring is defined over an exact base ring.

EXAMPLES:

```python
sage: PolynomialRing(QQ, 2, 'x').is_exact()
True
sage: PolynomialRing(RDF, 2, 'x').is_exact()
False
```

**is_field**(proof=True)
Test whether this multivariate polynomial ring is a field.
A polynomial ring is a field when there are no variable and the base ring is a field.

EXAMPLES:

```python
sage: PolynomialRing(QQ, 'x', 2).is_field()
False
sage: PolynomialRing(QQ, 'x', 0).is_field()
True
sage: PolynomialRing(ZZ, 'x', 0).is_field()
False
sage: PolynomialRing(Zmod(1), names=['x','y']).is_finite()
True
```

**is_integral_domain**(proof=True)

EXAMPLES:

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

**is_noetherian()**

EXAMPLES:

```python
sage: ZZ['x,y'].is_noetherian()
True
sage: Integers(8)['x,y'].is_noetherian()
True
```

**krull_dimension()**

**macaulay_resultant**(*args, **kwds)**
This is an implementation of the Macaulay Resultant. It computes the resultant of universal polynomials as well as polynomials with constant coefficients. This is a project done in sage days 55. It’s based on the implementation in Maple by Manfred Minimair, which in turn is based on the references listed below: It calculates the Macaulay resultant for a list of polynomials, up to sign!

REFERENCES:

- [CLO2005]
- [Can1990]

3.1. Multivariate Polynomials and Polynomial Rings 261
• [Mac1916]

AUTHORS:
• Hao Chen, Solomon Vishkautsan (7-2014)

INPUT:
• **args** – a list of \( n \) homogeneous polynomials in \( n \) variables. works when \( \text{args}[0] \) is the list of polynomials, or \( \text{args} \) is itself the list of polynomials

kwds:
• **sparse** – boolean (optional - default: **False**) if \( \text{True} \) function creates sparse matrices.

OUTPUT:
• the macaulay resultant, an element of the base ring of \( \text{self} \)

**Todo:** Working with sparse matrices should usually give faster results, but with the current implementation it actually works slower. There should be a way to improve performance with regards to this.

**EXAMPLES:**
The number of polynomials has to match the number of variables:

```sage
R.<x,y,z> = PolynomialRing(QQ,3)
sage: R.macaulay_resultant([y,x+z])
Traceback (most recent call last):
  ... TypeError: number of polynomials(= 2) must equal number of variables (= 3)
```

The polynomials need to be all homogeneous:

```sage
R.<x,y,z> = PolynomialRing(QQ,3)
sage: R.macaulay_resultant([y, x+z, z+x^3])
Traceback (most recent call last):
  ... TypeError: resultant for non-homogeneous polynomials is not supported
```

All polynomials must be in the same ring:

```sage
S.<x,y> = PolynomialRing(QQ, 2)
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: S.macaulay_resultant([y, z+x])
Traceback (most recent call last):
  ... TypeError: not all inputs are polynomials in the calling ring
```

The following example recreates Proposition 2.10 in Ch.3 in [CLO2005]:

```sage
K.<x,y> = PolynomialRing(ZZ, 2)
sage: flist,R = K._macaulay_resultant_universal_polynomials([1,1,2])
sage: R.macaulay_resultant(flist)
u2^2*u4^2*u6 - 2*u1*u2*u4*u5*u6 + u1^2*u5^2*u6 - u2^2*u3*u4*u7 + u1*u2*u3*u5*u7 + u0*u2*u4*u5*u7 - u0*u1*u5^2*u7 + u1*u2*u3*u4*u8 - u0*u2*u4^2 - u1^2*u3*u5*u8 + u0*u1*u4*u5*u8 + u2^2*u3^2*u9 - 2*u0*u2*u3*u5*u9 + u0^2*u4^2*u11
```

262 Chapter 3. Multivariate Polynomials
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 be 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:
```python
sage: R.<x> = PolynomialRing(QQ,1)
sage: f = x^2+1; g = x^5+1
sage: fh = f.homogenize()
sage: gh = g.homogenize()
sage: RH = fh.parent()
sage: f.resultant(g) == RH.macaulay_resultant([fh,gh])
True
```

**monomial** (*exponents*)

Return the monomial with given exponents.

**EXAMPLES:**

```python
sage: R.<x,y,z> = PolynomialRing(ZZ, 3)
sage: R.monomial(1,1,1)
x*y*z
sage: e=(1,2,3)
sage: R.monomial(*e)
x*y^2*z^3
sage: m = R.monomial(1,2,3)
sage: R.monomial(*m.degrees()) == m
True
```

**ngens** ()

**random_element** (*degree=2, terms=None, choose_degree=False, *args, **kwargs*)

Return a random polynomial of at most degree $d$ and at most $t$ terms.

First monomials are chosen uniformly random from the set of all possible monomials of degree up to $d$ (inclusive). This means that it is more likely that a monomial of degree $d$ appears than a monomial of degree $d-1$ because the former class is bigger.

Exactly $t$ distinct monomials are chosen this way and each one gets a random coefficient (possibly zero) from the base ring assigned.

The returned polynomial is the sum of this list of terms.

**INPUT:**

- *degree* – maximal degree (likely to be reached) (default: 2)
- *terms* – number of terms requested (default: 5). If more terms are requested than exist, then this parameter is silently reduced to the maximum number of available terms.
- *choose_degree* – choose degrees of monomials randomly first rather than monomials uniformly random.
- **kwargs** – passed to the random element generator of the base ring

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: 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)

\[
\begin{bmatrix}
357*x^2 + 1/4*y^2 + 2*y*z + 2*z^2 + 28*x^
\rightarrow 2 - z] \\
x*y - y*z + 2*z^2 - x^2 - 4/3*x*z + 2*z^2 - x
\end{bmatrix}
\]

sage: random_matrix(QQ['x,y,z'], 2, 2, terms=1, degree=2)

\[
\begin{bmatrix}
1/2*x*y -1/4*x \\
1/2 1/3*x
\end{bmatrix}
\]

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()
2*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 - 5*y^2 + 15*y^3 + 61*x^2*z - 12*x*y*z + 20*y^2*z - 61*x*z^2

\rightarrow 2 - 5*y*z^2 + 62*z^3 + 15*x^2 - 47*x*y + 31*y^2 - 14*x*z + 29*y*z + 13*z^2

\rightarrow 61*x - 40*y - 49*z + 30

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

\rightarrow 2 + 7*y*z^2 + 6*z^3 - 38*x^2 - 18*x*y - 52*y^2 + 27*x*z + 4*y*z - 51*z^2 -

\rightarrow 63*x + 7*y + 48*z + 14

The number of terms is silently reduced to the maximum available if more terms are requested:

sage: P.<x,y,z> = GF(127)[[]]

sage: P.random_element(degree=2, terms=1000)
5*x^2 - 10*x*y + 10*x^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)

(continues on next page)
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
sage: P.remove_var(y,z,x,w)
Removing all variables results in the base ring:
sage: P.remove_var(y,z,x,w)
Integer Ring
If possible, the term order is kept:

sage: R.<x,y,z,w> = PolynomialRing(ZZ, order='deglex')
sage: R.remove_var(y).term_order()
Degree lexicographic term order
sage: R.<x,y,z,w> = PolynomialRing(ZZ, order='lex')
sage: R.remove_var(y).term_order()
Lexicographic term order
Be careful with block orders when removing variables:

sage: R.<x,y,z,u,v> = PolynomialRing(ZZ, order='deglex(2),lex(3)')
sage: R.remove_var(x,y,z)  
Traceback (most recent call last):
  ...  
ValueError: impossible to use the original term order (most likely because it_  
→was a block order). Please specify the term order for the subring
sage: R.remove_var(x,y,z, order='degrevlex')
Multivariate Polynomial Ring in u, v over Integer Ring

repr_long()
Return structured string representation of self.

EXAMPLES:

sage: P.<x,y,z> = PolynomialRing(QQ,order=TermOrder('degrevlex',1)+TermOrder( ˓→'lex',2))
sage: print(P.repr_long())
Polynomial Ring
  Base Ring : Rational Field
  Size : 3 Variables
  Block 0 : Ordering : degrevlex
    Names : x
  Block 1 : Ordering : lex
    Names : y, z

term_order()
univariate_ring(x)
Return a univariate polynomial ring whose base ring comprises all but one variables of self.

INPUT:
  • x – a variable of self.

EXAMPLES:
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']
sage: R.variable_names_recursive()
('x', 'y')
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
```

3.1.3 Base class for elements of multivariate polynomial rings

class sage.rings.polynomial.multi_polynomial.MPolynomial
Bases: sage.structure.element.CommutativeRingElement

args()
Returns the named of the arguments of self, in the order they are accepted from call.

EXAMPLES:
```
sage: R.<x,y> = ZZ[]
sage: x.args()
(x, y)
```

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

INPUT:
- \( R \) – a ring or morphism.

**EXAMPLES:**

```python
sage: R.<x,y> = QQ[]
sage: f = x^3 + 3/5*y + 1
sage: f.change_ring(GF(7))
x^3 + 2*y + 1

sage: R.<x,y> = GF(9,'a')[]
sage: (x+2*y).change_ring(GF(3))
x - y

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

### coefficients()()

Return the nonzero coefficients of this polynomial in a list. The returned list is decreasingly ordered by the term ordering of `self.parent()`, i.e., the list of coefficients matches the list of monomials returned by `sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular.monomials()`.

**EXAMPLES:**

```python
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 \( \mathbb{Z} \) – different implementation:

```python
sage: R.<x,y,z> = PolynomialRing(ZZ,3,order='degrevlex')
sage: f=23*x^6*y^7 + x^3*y+6*x^7*z
sage: f.coefficients()
[23, 6, 1]

sage: R.<x,y,z> = PolynomialRing(ZZ,3,order='lex')
sage: f=23*x^6*y^7 + x^3*y+6*x^7*z
sage: f.coefficients()
[6, 23, 1]
```

**AUTHOR:**

- Didier Deshommes

### content()

Returns the content of this polynomial. Here, we define content as the \( \gcd \) of the coefficients in the base ring.

**See also:**

- `content_ideal()`

**EXAMPLES:**

```python
```
sage: R.<x,y> = ZZ[
]
sage: f = 4*x+6*y
sage: f.content()
2
sage: f.content().parent()
Integer Ring
```

**content_ideal()**

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

See also:

- **content()**

**EXAMPLES:**

```
sage: R.<x,y> = ZZ[
]
sage: f = 2*x*y + 6*x - 4*y + 2
sage: f.content_ideal()
Principal ideal (2) of Integer Ring
sage: S.<z,t> = R[
]
sage: g = x*z + y*t
sage: g.content_ideal()
Ideal (x, y) of Multivariate Polynomial Ring in x, y over Integer Ring
```

**denominator()**

Return a denominator of self.

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

Note that some subclasses may implement its own denominator function.

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

**EXAMPLES:**

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

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

Next we compute the denominator of a polynomial over a number field.

```
sage: R.<x,y> = NumberField(symbolic_expression(x^2+3) , 'a')['x,y']
sage: f = (1/17)*x^19 + (1/6)*y - (2/3)*x + 1/3; f
1/17*x^19 - 2/3*x + 1/6*y + 1/3
sage: f.denominator()
102
```

Finally, we try to compute the denominator of a polynomial with coefficients in the real numbers, which is a ring whose elements do not have a denominator method.
sage: R.<a,b,c> = RR[]
sage: f = a + b + RR('0.3'); f
a + b + 0.300000000000000
sage: f.denominator()
1

Check that the denominator is an element over the base whenever the base has no denominator function. This closes trac ticket #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)
(continues on next page)
Polynomials over the symbolic ring (just for fun...):

```sage
sage: x = var("x")
sage: S.<u, v> = PolynomialRing(SR)
sage: f = u*v*x
sage: f.derivative(x) == u*v
True
sage: f.derivative(u) == v*x
True
```

**discriminant** *(variable)*

Returns the discriminant of self with respect to the given variable.

**INPUT:**

- **variable** - The variable with respect to which we compute the discriminant

**OUTPUT:**

- An element of the base ring of the polynomial ring.

**EXAMPLES:**

```sage
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
sage: R.<x,y>=QQ[]
sage: f=x^5*y+3*x^2*y^2-2*x+y-1
sage: f.discriminant(y)
x^10 + 2*x^5 + 24*x^3 + 12*x^2 + 1
sage: f.polynomial(y).discriminant()
x^10 + 2*x^5 + 24*x^3 + 12*x^2 + 1
sage: f.discriminant(y).parent()==f.polynomial(y).discriminant().parent()
False
```

**AUTHOR:** Miguel Marco

**gcd** *(other)*

Return a greatest common divisor of this polynomial and other.

**INPUT:**

```sage
(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
```
• other – a polynomial with the same parent as this polynomial

**EXAMPLES:**

```python
sage: Q.<z> = Frac(QQ['z'])
sage: R.<x,y> = Q[]
sage: r = x*y - (2*z-1)/(z^2+z+1) * x + y/z
sage: q = r * (x+y*z + 1)
sage: gcd(p,q)
(z^3 + z^2 + z)*x*y + (-2*z^2 + z)*x + (z^2 + z + 1)*y
```

Polynomials over polynomial rings are converted to a simpler polynomial ring with all variables to compute the gcd:

```python
sage: A.<z,t> = ZZ[]
sage: B.<x,y> = A[]
sage: r = x*y*z*t+1
sage: p = r * (x - y + z - t + 1)
sage: q = r * (x*z - y*t)
sage: gcd(p,q)
z*t*x*y + 1
```

Some multivariate polynomial rings have no gcd implementation:

```python
sage: R.<x,y> = GaussianIntegers()[]
sage: x.gcd(x)
Traceback (most recent call last):
  ... NotImplementedError: GCD is not implemented for multivariate polynomials over Gaussian Integers in Number Field in I with defining polynomial x^2 + 1 with I = 1*I
```

**gradient()**

Return a list of partial derivatives of this polynomial, ordered by the variables of self.parent().

**EXAMPLES:**

```python
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()
sage: 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
sage: 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
sage: 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
sage: x^2 + y^2
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
sage: 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)
-1 + x^2
```

If it is present already, this might not be the case:

```
sage: g = f.homogenize(x); g
sage: x^2 - x*y
sage: g(1,y,z)
-1 + y
```

In particular, this can be surprising in positive characteristic:
```python
sage: R.<x,y> = GF(2)[]
sage: f = x + 1
sage: f.homogenize(x)
0
```

**inverse_mod(I)**

Returns an inverse of self modulo the polynomial ideal $I$, namely a multivariate polynomial $f$ such that $\text{self} \ast f - 1$ 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:**

```python
sage: R.<x1,x2> = QQ[]
sage: I = R.ideal(x2**2 + x1 - 2, x1**2 - 1)
sage: f = x1 + 3*x2^2; g = f.inverse_mod(I); g
1/16*x1 + 3/16
sage: (f*g).reduce(I)
1
```

Test a non-invertible element:

```python
sage: R.<x1,x2> = QQ[]
sage: I = R.ideal(x2**2 + x1 - 2, x1**2 - 1)
sage: f = x1 + x2
sage: f.inverse_mod(I)
Traceback (most recent call last):
 ... ArithmeticError: element is non-invertible
```

**is_generator()**

Returns True if this polynomial is a generator of its parent.

**EXAMPLES:**

```python
sage: R.<x,y>=ZZ[]
sage: x.is_generator()
True
sage: (x+y-y).is_generator()
True
sage: (x*y).is_generator()
False
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()**

Returns True if self is a homogeneous polynomial.
**is_nilpotent()**

Return True if self is nilpotent, i.e., some power of self is 0.

**EXAMPLES:**

```python
sage: R.<x,y> = QQbar[]
sage: (x+y).is_nilpotent()
False
sage: R(0).is_nilpotent()
True
sage: _.<x,y> = Zmod(4)[]
sage: (2*x).is_nilpotent()
True
sage: (2+y*x).is_nilpotent()
False
sage: _.<x,y> = Zmod(36)[]
sage: (4+6*x).is_nilpotent()
False
sage: (6*x + 12*y + 18*x*y + 24*(x^2+y^2)).is_nilpotent()
True
```

**is_square(root=False)**

Test whether this polynomial is a square root.

**INPUT:**

- root - if set to True return a pair (True, root) where root is a square root or (False, None) if it is not a square.

**EXAMPLES:**

```python
sage: R.<a,b> = QQ[]
sage: a.is_square()
False
sage: ((1+a*b^2)^2).is_square()
True
sage: ((1+a*b^2)^2).is_square(root=True)
(True, a*b^2 + 1)
```

**is_unit()**

Return True if self is a unit, that is, has a multiplicative inverse.

**EXAMPLES:**

```python
sage: R.<x,y> = QQbar[]
sage: (x+y).is_unit()
False
sage: R(0).is_unit()
False
sage: R(-1).is_unit()
True
sage: R(-1 + x).is_unit()
False
sage: R(2).is_unit()
True
```
Check that trac ticket #22454 is fixed:

```
sage: _.<x,y> = Zmod(4)[]
sage: (1 + 2*x).is_unit ()
True
sage: (x+y).is_unit ()
False
sage: _.<x,y> = Zmod(36)[]
sage: (7+ 6*x + 12*y - 18*x*y).is_unit ()
True
```

**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()**

given an ideal \( I = (f_1, \ldots, f_r) \) and some \( g \) (== self) in \( I \), find \( s_1, \ldots, s_r \) such that \( g = s_1 f_1 + \ldots + s_r f_r \).

**EXAMPLES:**

```
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:
The polynomials need to be all homogeneous:

```
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)
```

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*u5*u7 - u0*u1*u5^2*u7 + u1*u2*u3*u4*u8 - u0*u2*u4^ 
   2*u8 - u1^2*u3*u5*u8 + u0*u1*u4*u5*u8 + u2^2*u3^2*u9 - 2*u0*u2*u3*u5*u9 + 
   u0^2*u5^2*u9 - u1*u2*u3^2*u10 + u0*u2*u3*u4*u10 + u0*u1*u3*u5*u10 - u0^ 
   2*u4*u5*u10 + u1^2*u3^2*u11 - 2*u0*u1*u3*u4*u11 + u0^2*u4^2*u11
```

The following example degenerates into the determinant of a 3 * 3 matrix:

```
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: y.macaulay_resultant(x+z)
Traceback (most recent call last):
  ...  
TypeError: resultant for non-homogeneous polynomials is not supported
```

3.1. Multivariate Polynomials and Polynomial Rings
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:

```python
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:**

```python
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
```

`nth_root(n)`
Return a \(n\)-th root of this element.

If there is no such root, a `ValueError` is raised.

**EXAMPLES:**

```python
sage: R.<x,y,z> = QQ[]
sage: a = 32 * (x*y + 1)^5 * (x+y+z)^5
sage: a.nth_root(5)
2*x^2*y + 2*x*y^2 + 2*x*y*z + 2*x + 2*y + 2*z
sage: b = x + 2*y + 3*z
sage: b.nth_root(42)
Traceback (most recent call last):
  ... ValueError: not a 42nd power
```

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

\[(x*z*t + x)*u + (z*t + 1)*v + (y + 1)*z*t + y + 1\]

```
sage: (p**3).nth_root(3).parent() is p.parent()
True
sage: ((1+x+z+t)**2).nth_root(3)
Traceback (most recent call last):
...
ValueError: not a 3rd power
```

### numerator()

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

Note that some subclasses may implement its own numerator function.

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

#### EXAMPLES:

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

```
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 \( (\text{var}) \)

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

EXAMPLES:

```python
sage: R.<x,w,z> = QQ[]
sage: f = x^3 + 3*w*x + w^5 + (17*w^3)*x + z^5
sage: f.polynomial(x)
x^3 + (17*w^3 + 3*w)*x + w^5 + z^5

sage: parent(f.polynomial(x))
Univariate Polynomial Ring in x over Multivariate Polynomial Ring in w, z
  over Rational Field
```

```python
sage: R.<x,w,z,k> = ZZ[

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
```

reduced_form (**\text{kwsds})

Return a reduced form of this polynomial.

The algorithm is from Stoll and Cremona’s “On the Reduction Theory of Binary Forms” [CS2003]. This takes a two variable homogenous polynomial and finds a reduced form. This is a \( SL(2,\mathbb{Z}) \)-equivalent binary form whose covariant in the upper half plane is in the fundamental domain. If the polynomial has multiple roots, they are removed and the algorithm is applied to the portion without multiple roots.

This reduction should also minimize the sum of the squares of the coefficients, but this is not always the case. By default the coefficient minimizing algorithm in [HS2018] is applied. The coefficients can be minimized either with respect to the sum of their squares or the maximum of their global heights.

A portion of the algorithm uses Newton’s method to find a solution to a system of equations. If Newton’s method fails to converge to a point in the upper half plane, the function will use the less precise \( z_0 \) covariant from the \( Q_0 \) form as defined on page 7 of [CS2003]. Additionally, if this polynomial has a root with multiplicity at least half the total degree of the polynomial, then we must also use the \( z_0 \) covariant. See [CS2003] for details.

Note that, if the covariant is within error_limit of the boundary but outside the fundamental domain, our function will erroneously move it to within the fundamental domain, hence our conjugation will be off by 1. If you don’t want this to happen, decrease your error_limit and increase your precision.

Implemented by Rebecca Lauren Miller as part of GSOC 2016. Smallest coefficients added by Ben Hutz July 2018.
INPUT:

keywords:

• **prec** – integer, sets the precision (default:300)
• **return_conjugation** – boolean. Returns element of $SL(2, \mathbb{Z})$ (default:True)
• **error_limit** – sets the error tolerance (default:0.000001)
• **smallest_coeffs** – (default: True), boolean, whether to find the model with smallest coefficients
• **norm_type** – either 'norm' or 'height'. What type of norm to use for smallest coefficients
• **emb** – (optional) embedding of based field into CC

OUTPUT:

• a polynomial (reduced binary form)
• a matrix (element of $SL(2, \mathbb{Z})$)

TODO: When Newton's Method doesn't converge to a root in the upper half plane. Now we just return $z_0$. It would be better to modify and find the unique root in the upper half plane.

EXAMPLES:

```sage
R.<x,h> = PolynomialRing(QQ)
sage: f = 19*x^8 - 262*x^7*h + 1507*x^6*h^2 - 4784*x^5*h^3 + 9202*x^4*h^4 - 10962*x^3*h^5 + 7844*x^2*h^6 - 3040*x*h^7 + 475*h^8
sage: f.reduced_form(prec=200, smallest_coeffs=False)
(-x^8 - 2*x^7*h + 7*x^6*h^2 + 16*x^5*h^3 + 2*x^4*h^4 - 2*x^3*h^5 + 4*x^2*h^6 - 5*h^8,
[ 1 -2]
[ 1 -1])
```

An example where the multiplicity is too high:

```sage
R.<x,y> = PolynomialRing(QQ)
sage: f = x^3 + 378666*x^2*y - 12444444*x*y^2 + 1234567890*y^3
sage: j = f * (x-545*y)^9
sage: j.reduced_form(prec=200, smallest_coeffs=False)
Traceback (most recent call last):
  ... ValueError: cannot have a root with multiplicity >= 12/2
```

An example where Newton's Method does not find the right root:

```sage
R.<x,y> = PolynomialRing(QQ)
sage: F = x^6 + 3*x^5*y - 8*x^4*y^2 - 2*x^3*y^3 - 44*x^2*y^4 - 8*x*y^5
sage: F.reduced_form(smallest_coeffs=False, prec=400)
Traceback (most recent call last):
  ... ArithmeticError: Newton's method converged to z not in the upper half plane
```

An example with covariant on the boundary, therefore a non-unique form:
```python
sage: R.<x,y> = PolynomialRing(QQ)
sage: F = 5*x^2*y - 5*x*y^2 - 30*y^3
sage: F.reduced_form(smallest_coeffs=False)
(5*x^2*y + 5*x*y^2 - 30*y^3, [0 1])
```

An example where precision needs to be increased:

```python
sage: R.<x,y> = PolynomialRing(QQ)
sage: F=-16*x^7 - 114*x^6*y - 345*x^5*y^2 - 599*x^4*y^3 - 666*x^3*y^4 - 481*x^2*y^5 - 207*x*y^6 - 40*y^7
sage: F.reduced_form(prec=50, smallest_coeffs=False)
Traceback (most recent call last):
  ... Value Error: accuracy of Newton's root not within tolerance(0.0000124... > 1e-06), increase precision
sage: F.reduced_form(prec=100, smallest_coeffs=False)
([-1 -1]
-2*x^5*y^2 - 24*x^3*y^4 - 3*x^2*y^5 - 2*x*y^6 + 16*y^7, [1 0])
```

```python
sage: R.<x,y> = PolynomialRing(QQ)
sage: F = - 8*x^4 - 3933*x^3*y - 725085*x^2*y^2 - 59411592*x*y^3 - 99*y^6
```

3.1. Multivariate Polynomials and Polynomial Rings 283
specialization(D=None, phi=None)

Specialization of this polynomial.

Given a family of polynomials defined over a polynomial ring. A specialization is a particular member of that family. The specialization can be specified either by a dictionary or a SpecializationMorphism.

INPUT:

- D – dictionary (optional)
- phi – SpecializationMorphism (optional)

OUTPUT: a new polynomial

EXAMPLES:

```python
sage: R.<c> = PolynomialRing(QQ)
sage: S.<x,y> = PolynomialRing(R)
sage: F = x^2 + c*y^2
sage: F.specialization({c:2})
x^2 + 2*y^2
```
Check that we preserve multi- versus uni-variate:

```
sage: R.<l> = PolynomialRing(QQ, 1)
sage: S.<k> = PolynomialRing(R)
sage: K.<a, b, c> = PolynomialRing(S)
sage: F = a*k^2 + b*l + c^2
sage: F.specialization({b:56, c:5}).parent()
Univariate Polynomial Ring in a over Univariate Polynomial Ring in k over Multivariate Polynomial Ring in l over Rational Field
```

**sylvester_matrix** (*right*, *variable=None*)

Given two nonzero polynomials self and right, returns the Sylvester matrix of the polynomials with respect to a given variable.

Note that the Sylvester matrix is not defined if one of the polynomials is zero.

**INPUT:**

- self, right: multivariate polynomials
- variable: optional, compute the Sylvester matrix with respect to this variable. If variable is not provided, the first variable of the polynomial ring is used.

**OUTPUT:**

- The Sylvester matrix of self and right.

**EXAMPLES:**

```
sage: R.<x, y> = PolynomialRing(ZZ)
sage: f = (y + 1)*x + 3*x**2
sage: g = (y + 2)*x + 4*x**2
sage: M = f.sylvester_matrix(g, x)
sage: M
[ 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

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)
6805647384187692694195958937245974528
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: unable to convert non-constant polynomial x to an integer
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

sage.rings.polynomial.multi_polynomial.is_MPolynomial(x)

### 3.1.4 Multivariate Polynomial Rings over Generic Rings

Sage implements multivariate polynomial rings through several backends. This generic implementation uses the classes `PolyDict` and `ETuple` to construct a dictionary with exponent tuples as keys and coefficients as values.

**AUTHORS:**
- David Joyner and William Stein
- Kiran S. Kedlaya (2006-02-12): added Macaulay2 analogues of Singular features
- Martin Albrecht (2006-04-21): reorganize class hierarchy for singular rep
- Martin Albrecht (2007-04-20): reorganized class hierarchy to support Pyrex implementations
- Robert Bradshaw (2007-08-15): Coercions from rings in a subset of the variables

**EXAMPLES:**

We construct the Frobenius morphism on $F_5[x, y, z]$ over $F_5$:

```python
sage: R.<x,y,z> = GF(5)[]
sage: frob = R.hom([x^5, y^5, z^5])
sage: frob(x^2 + 2*y - z^4)
-x^20 + x^10 + 2*y^5
sage: frob((x + 2*y)^3)
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:

```python
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`

Bases: `object`

A mixin class for polynomial rings that support conversion to Macaulay2.

**class** `sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict`

*(base_ring, n, names, order)*

Bases: `sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_macaulay2_repr, sage.rings.polynomial.polynomial_singular_interface.PolynomialRing_singular_repr, sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base`

Multivariable polynomial ring.

**EXAMPLES:**
```python
sage: R = PolynomialRing(Integers(12), 'x', 5); R
Multivariate Polynomial Ring in x0, x1, x2, x3, x4 over Ring of integers modulo 12
sage: loads(R.dumps()) == R
True
```

`monomial_all_divisors(t)`

Return a list of all monomials that divide \( t \), coefficients are ignored.

**INPUT:**
- \( t \) - a monomial.

**OUTPUT:** a list of monomials.

**EXAMPLES:**
```python
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_˓
    →polydict_domain
sage: P.<x,y,z> = MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_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

**OUTPUT:** Boolean

**EXAMPLES:**
```python
sage: P.<x,y,z> = PolynomialRing(ZZ,3, order='degrevlex')
sage: P.monomial_divides(x*y*z, x^3*y^2*z^4)
True
sage: P.monomial_divides(x^3*y^2*z^4, x*y*z)
False
```

`monomial_lcm(f, g)`

LCM for monomials. Coefficients are ignored.

**INPUT:**
- \( f \) - monomial.
- \( g \) - monomial.

**OUTPUT:** monomial.

**EXAMPLES:**
```python
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_˓
    →polydict_domain
sage: P.<x,y,z> = MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_lcm(3/2*x*y, x)
x*y
```
monomial_pairwise_prime(h, g)
Return True if h and g are pairwise prime.
Both are treated as monomials.

INPUT:
• h - monomial.
• g - monomial.
OUTPUT: Boolean.

EXAMPLES:
```python
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domain
sage: P.<x,y,z> = MPolynomialRing_polydict_domain(QQ, 3, order='degrevlex')
sage: P.monomial_pairwise_prime(x^2*z^3, y^4)
True
sage: P.monomial_pairwise_prime(1/2*x^3*y^2, 3/4*y^3)
False
```

monomial_quotient(f, g, coeff=False)
Return f/g, where both f and g are treated as monomials.
Coefficients are ignored by default.

INPUT:
• f - monomial.
• g - monomial.
• coeff - divide coefficients as well (default: False).

OUTPUT: monomial.

EXAMPLES:
```python
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domain
sage: P.<x,y,z> = MPolynomialRing_polydict_domain(QQ, 3, order='degrevlex')
sage: P.monomial_quotient(3/2*x*y, x)
y
sage: P.monomial_quotient(3/2*x*y, 2*x, coeff=True)
3/4*y
```

Note: Assumes that the head term of f is a multiple of the head term of g and return the multiplicant m. If this rule is violated, funny things may happen.

monomial_reduce(f, G)
Try to find a g in G where g.lm() divides f.
If found, (flt, g) is returned, (0, 0) otherwise, where flt is f/g.lm(). It is assumed that G is iterable and contains ONLY elements in this ring.

INPUT:
• \( f \) - monomial
• \( G \) - list/set of \( m \)-polynomials

EXCEPTIONS:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domain
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: f = x*y^2
sage: G = [3/2*x^3 + y^2 + 1/2, 1/4*x*y + 2/7, P(1/2)]
```

```
sage: P.monomial_reduce(f,G)
(y, 1/4*x*y + 2/7)
```

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domain
sage: P.<x,y,z> = MPolynomialRing_polydict_domain(Zmod(23432),3, order='degrevlex')
sage: f = x*y^2
sage: G = [3*x^3 + y^2 + 2, 4*x*y + 7, P(2)]
```

```
sage: P.monomial_reduce(f,G)
(y, 4*x*y + 7)
```

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.MPolynomialRing_polydict

ideal (*gens, **kwds)
Create an ideal in this polynomial ring.

is_field (proof=True)

is_integral_domain (proof=True)

3.1.5 Generic Multivariate Polynomials

AUTHORS:

• David Joyner: first version
• William Stein: use dict’s instead of lists
• Martin Albrecht malb@informatik.uni-bremen.de: some functions added
• Kiran S. Kedlaya (2006-02-12): added Macaulay2 analogues of some Singular features
• William Stein (2006-04-19): added e.g., \texttt{f[1,3]} to get coeff of \( xy^3 \); added examples of the new \texttt{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:

```python
sage: R.<a0,a1,a2,a3,b0,b1,b2,b3> = QQbar[]
sage: (a0^2 + a1^2 + a2^2 + a3^2)*(b0^2 + b1^2 + b2^2 + b3^2) == (a0*b0 - a1*b1 -
˓→a2*b2 - a3*b3)^2 + (a0*b1 + a1*b0 + a2*b3 - a3*b2)^2 + (a0*b2 - a1*b3 + a2*b0 +
˓→a3*b1)^2 + (a0*b3 + a1*b2 - a2*b1 + a3*b0)^2
True
```

```python
class sage.rings.polynomial.multi_polynomial_element.MPolynomial_element (parent, x)

Bases: sage.rings.polynomial.multi_polynomial.MPolynomial

EXAMPLES:

```python
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
```n
direct doctest

```python
change_ring (R)

Change the base ring of this polynomial to \( R \).

INPUT:

- \( R \) – ring or morphism.

OUTPUT: a new polynomial converted to \( R \).

EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: f = x^2 + 5*y
sage: f.change_ring(GF(5))
x^2
```n
```python
sage: K.<w> = CyclotomicField(5)
sage: R.<x,y> = K[]
sage: f = x^2 + w*y
sage: f.change_ring(K.embeddings(QQbar)[1])
x^2 + (-0.8090169943749474? + 0.5877852522924731?*I)*y
```

```python
element ()

```

```python
hamming_weight ()

Return the number of non-zero coefficients of this polynomial.

This is also called weight, hamming_weight() or sparsity.

EXAMPLES:

```python
sage: R.<x, y> = CC[]
sage: f = x^3 - y
sage: f.number_of_terms()
2
sage: R(0).number_of_terms()
0
sage: f = (x+y)^100
```

(continues on next page)
The method `hamming_weight()` is an alias:

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

**number_of_terms()**

Return the number of non-zero coefficients of this polynomial.

This is also called weight, `hamming_weight()` or sparsity.

**EXAMPLES:**

```python
sage: R.<x, y> = CC[]
sage: f = x^3 - y
sage: f.number_of_terms()
2
sage: R(0).number_of_terms()
0
sage: f = (x+y)^100
sage: f.number_of_terms()
101
```

The method `hamming_weight()` is an alias:

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

**class** `sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict`

```
parent, x)
```

Multivariate polynomials implemented in pure python using polydicts.

**coefficient** *(degrees)*

Return the coefficient of the variables with the degrees specified in the python dictionary `degrees`. Mathematically, this is the coefficient in the base ring adjoined by the variables of this ring not listed in `degrees`. However, the result has the same parent as this polynomial.

This function contrasts with the function `monomial_coefficient` which returns the coefficient in the base ring of a monomial.

**INPUT:**

- `degrees` - Can be any of:
  - a dictionary of degree restrictions
  - a list of degree restrictions (with None in the unrestricted variables)
  - a monomial (very fast, but not as flexible)

**OUTPUT:** element of the parent of self

**See also:**

For coefficients of specific monomials, look at `monomial_coefficient()`.
EXAMPLES:

```python
sage: R.<x, y> = QQbar[]
sage: f = 2 * x * y
c = f.coefficient({x:1,y:1}); c
2
sage: c.parent()
Multivariate Polynomial Ring in x, y over Algebraic Field
c in PolynomialRing(QQbar, 2, names = ['x','y'])
True
sage: f = y^2 - x^9 - 7*x + 5*x*y
c = f.coefficient({x:1})
5*x
sage: f = y^2 - x^9 + 7*x + 5*x*y
sage: c = f.coefficient({y:0})
-x^9 + 7*x
sage: f = (1+y+y^2)*(1+x+x^2)
c = f.coefficient({x:0})
y^2 + y + 1
sage: f = (1+y+y^2)*(1+x+x^2)
c = f.coefficient({x:0})
y^2 + y + 1
sage: c = f.coefficient({x})
y^2 + y + 1
sage: # Be aware that this may not be what you think!
sage: # The physical appearance of the variable x is deceiving --
# particularly if the exponent would be a variable.
sage: f = x^0
sage: c = f.coefficient({x:0})
# outputs the full polynomial
x^2*y^2 + x^2*y + x*y^2 + x^2 + x*y + y^2 + x + y + 1
```

```python
sage: R.<x,y> = RR[]
sage: f=x*y+5
c = f.coefficient({x:0,y:0}); c
5.00000000000000
sage: parent(c)
Multivariate Polynomial Ring in x, y over Real Field with 53 bits of precision
```

AUTHORS:

• Joel B. Mohler (2007-10-31)

constant_coefficient()
Return the constant coefficient of this multivariate polynomial.

EXAMPLES:

```python
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: c = f.constant_coefficient()
5
sage: f = 3*x^2
sage: c = f.constant_coefficient()
0
```

degree(x=None, std_grading=False)
Return the degree of self in x, where x must be one of the generators for the parent of self.

INPUT:

• x - multivariate polynomial (a generator of the parent of self). If x is not specified (or is None),
return the total degree, which is the maximum degree of any monomial. Note that a weighted term ordering alters the grading of the generators of the ring; see the tests below. To avoid this behavior, set the optional argument \texttt{std\_grading=True}.

**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 total degree takes into account if we are working in a polynomial ring with a weighted term order.

```
sage: R = PolynomialRing(QQ,'x,y',order=TermOrder('wdeglex',(2,3)))
sage: x,y = R.gens()
sage: x.degree()
2
sage: y.degree()
3
sage: x.degree(y),x.degree(x),y.degree(x),y.degree(y)
(0, 1, 0, 1)
sage: f = (x^2*y+x*y^2)
sage: f.degree(x)
2
sage: f.degree(y)
2
sage: f.degree()
8
sage: f.degree(std\_grading=True)
3
```

Note that if \( x \) is not a generator of the parent of self, for example if it is a generator of a polynomial algebra which maps naturally to this one, then it is converted to an element of this algebra. (This fixes the problem reported in 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:

```python
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
2
```

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:

```python
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:

```python
sage: type(f.exponents()[0])
<type 'sage.rings.polynomial.polydict.ETuple'>
sage: type(f.exponents(as_ETuples=False)[0])
<... 'tuple'>
```

factor (proof=None)
Compute the irreducible factorization of this polynomial.

INPUT:
- proof'' - insist on provably correct results (default: `True unless explicitly disabled for the "polynomial" subsystem with sage.structure.proof.proof.WithProof.)

integral (var=None)
Integrates self with respect to variable var.
Note: The integral is always chosen so the constant term is 0.

If `var` is not one of the generators of this ring, `integral(var)` is called recursively on each coefficient of this polynomial.

EXAMPLES:

On polynomials with rational coefficients:

```python
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:

```python
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
```

`inverse_of_unit()`

Return the inverse of a unit in a ring.

`is_constant()`

True if polynomial is constant, and False otherwise.

EXAMPLES:

```python
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:

```python
sage: R.<x,y>=QQbar[]
sage: x.is_generator()
```

(continues on next page)
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 coefficient be 1.

**EXAMPLES:**
To require leading coefficient 1, use is_monomial():

```python
sage: (2*x+y).is_monomial()
False
sage: (2*x+y).is_term()
True
```

**is_univariate()**

Returns True if this multivariate polynomial is univariate and False otherwise.

**EXAMPLES:**

```python
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:**

```python
sage: R.<x,y,z>=QQbar[]
sage: f=3*x^2-y^2-x*y
sage: f.lc()
3
```

**lift(I)**

given an ideal \( I = (f_1, \ldots, f_r) \) and some \( g \) (== self) in \( I \), find \( s_1, \ldots, s_r \) such that \( g = s_1 f_1 + \ldots + s_r f_r \)

**ALGORITHM:** Use Singular.

**EXAMPLES:**

```python
sage: A.<x,y> = PolynomialRing(CC,2,order='degrevlex')
sage: I = A.ideal([x^10 + x^9*y^2, y^8 - x^2*y^7 ])
sage: f = x*y^13 + y^12
sage: M = f.lift(I)
sage: M
[y^7, x^7*y^2 + x^8 + x^5*y^3 + x^6*y + x^3*y^4 + x^4*y^2 + x*y^5 + x^2*y^3 + y^4, ...
```

(continues on next page)
lm()

Returns the lead monomial of self with respect to the term order of self.parent().

EXAMPLES:

```python
sage: R.<x,y,z>=PolynomialRing(GF(7),3,order='lex')
sage: (x^1*y^2 + y^3*z^4).lm()
x*y^2
sage: (x^3*y^2+z^4 + x^3*y^2*z^1).lm()
x^3*y^2*z^4
```

```python
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
```

```python
sage: R.<x,y,z>=PolynomialRing(QQbar,3,order='degrevlex')
sage: (x^1*y^5*z^2 + x^4*y^1*z^3).lm()
x*y^5*z^2
sage: (x^4*y^7*z^1 + x^4*y^2*z^3).lm()
x^4*y^7*z
```

lt()

Returns the leading term of self i.e., self.lc()*self.lm(). The notion of “leading term” depends on the ordering defined in the parent ring.

EXAMPLES:

```python
sage: R.<x,y,z>=PolynomialRing(QQbar)
sage: f=3*x^2-y^2-x*y
sage: f.lt()
3*x^2
```

```python
sage: R.<x,y,z>=PolynomialRing(QQbar,order="invlex")
sage: f=3*x^2-y^2-x*y
sage: f.lt()
-y^2
```

monomial_coefficient(mon)

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

This function contrasts with the function coefficient which returns the coefficient of a monomial viewing this polynomial in a polynomial ring over a base ring having fewer variables.

INPUT:

- mon - a monomial

OUTPUT: coefficient in base ring

See also:

For coefficients in a base ring of fewer variables, look at coefficient().

EXAMPLES:
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
```

```
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
-fx^3*gx^3 + 3*fx*fy^2*gx^2*gy + -3*fx^2*fy*gx*gy^2 + fx^3*gy^3
sage: F.monomials()
[fx^3*gx^3, fx*fy^2*gx^2*gy, fx^2*fy*gx*gy^2, fx^3*gy^3]
sage: F.coef ficients()
[-1, 3, -3, 1]
sage: sum(map(mul,zip(F.coefficients(),F.monomials()))) == F
True
```

```
nvariables()

Number of variables in this polynomial

EXAMPLES:

```
```
```python
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.nvariables ()
2
sage: g = f.subs({x:10}); g
700*y^2 + (-2)*y + 305
sage: g.nvariables ()
1
```

**quo_rem(right)**

Returns quotient and remainder of self and right.

**EXAMPLES:**

```python
sage: R.<x,y> = CC[]
sage: f = y*x^2 + x + 1
sage: f.quo_rem(x)
(x*y + 1.00000000000000, 1.00000000000000)
sage: R = QQ['a','b']['x','y','z']
sage: p1 = R('a + (1+2*b)*x*y + (3-a^2)*z')
sage: p2 = R('x-1')
sage: p1.quo_rem(p2)
((2*b + 1)*y, (2*b + 1)*y + (-a^2 + 3)*z + a)
sage: R.<x,y> = Qp(5)[]
sage: x.quo_rem(y)
Traceback (most recent call last):
...
TypeError: no conversion of this ring to a Singular ring defined
```

**ALGORITHM:** Use Singular.

**reduce(I)**

Reduce this polynomial by the polynomials in $I$.

**INPUT:**

- $I$ - a list of polynomials or an ideal

**EXAMPLES:**

```python
sage: P.<x,y,z> = QQbar[]
sage: f1 = -2 * x^2 + x^3
sage: f2 = -2 * y + x*y
sage: f3 = -x^2 + y^2
sage: F = Ideal([f1,f2,f3])
sage: g = x*y - 3*x*y^2
sage: g.reduce(F)
(-6)*y^2 + 2*y
sage: g.reduce(F.gens())
(-6)*y^2 + 2*y
sage: f = 3*x
sage: f.reduce([2*x,y])
0
```

3.1. Multivariate Polynomials and Polynomial Rings  301
 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.
For inexact rings or rings not available in Singular, this computes the determinant of the Sylvester matrix.

INPUT:

  • other – polynomial in self.parent()
  • variable – (optional) variable (of type polynomial) in self.parent()

EXAMPLES:

 sage: P.<x,y> = PolynomialRing(QQ, 2)
 sage: a = x + y
 sage: b = x^3 - y^3
 sage: a.resultant(b)
-2*y^3
 sage: a.resultant(b, y)
2*x^3

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))
(continues on next page)
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:**

```python
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:**

```python
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.univariate_polynomial()
Traceback (most recent call last):
...  
TypeError: polynomial must involve at most one variable
sage: g = f.subs({x:10}); g
700*y^2 + (-2)*y + 305
sage: g.univariate_polynomial()
700*y^2 - 2*y + 305
sage: g.univariate_polynomial(PolynomialRing(QQ,'z'))
700*z^2 - 2*z + 305
```

**variable(i)**

Returns i-th variable occurring in this polynomial.

**EXAMPLES:**
\begin{verbatim}
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:
\begin{verbatim}
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,)
\end{verbatim}
\end{verbatim}

\begin{verbatim}
sage.rings.polynomial.multi_polynomial_element.degree_lowest_rational_function(r, x)

Return the difference of valuations of r with respect to variable x.

INPUT:
• r – a multivariate rational function
• x – a multivariate polynomial ring generator x

OUTPUT:
• integer – the difference \text{val}_x(p) - \text{val}_x(q) where r = p/q

Note: This function should be made a method of the FractionFieldElement class.

EXAMPLES:
\begin{verbatim}
sage: R1 = PolynomialRing(FiniteField(5), 3, names = ["a","b","c"])
sage: F = FractionField(R1)
sage: a,b,c = R1.gens()
sage: f = 3*a*b^2*c^3+4*a*b*c
sage: g = a^2*b*c^2+2*a^2*b^4*c^7
sage: r = f/g; r
(-b*c^2 + 2)/(a*b^3*c^6 - 2*a*c)
sage: degree_lowest_rational_function(r,a)
-1
sage: degree_lowest_rational_function(r,b)
0
sage: degree_lowest_rational_function(r,c)
-1
\end{verbatim}
\end{verbatim}

\begin{verbatim}
sage.rings.polynomial.multi_polynomial_element.is_MPolynomial(x)
\end{verbatim}

\end{verbatim}
3.1.6 Ideals in multivariate polynomial rings.

Sage has a powerful system to compute with multivariate polynomial rings. Most algorithms dealing with these ideals are centered on the computation of Groebner bases. Sage mainly uses Singular to implement this functionality. Singular is widely regarded as the best open-source system for Groebner basis calculation in multivariate polynomial rings over fields.

AUTHORS:
- William Stein
- Kiran S. Kedlaya (2006-02-12): added Macaulay2 analogues of some Singular features
- Martin Albrecht (2009): added Groebner basis over rings functionality from Singular 3.1
- John Perry (2012): bug fixing equality & containment of ideals

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 \(\mathbb{Z}/17\mathbb{Z}\):

```
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)
```

(continues on next page)
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 \(\mathbb{Z}/17\mathbb{Z}\) 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 == 16*a*b^2 + 16*b^2
True
sage: (a+b)^17
a*b^16 + b^17
sage: S(17) == 0
True

Working with a polynomial ring over \(\mathbb{Z}\):

sage: R.<x,y,z,w> = ZZ[]
sage: I = ideal(x^2 + y^2 - z^2 - w^2, x-y)
sage: J = I^2
sage: J.groebner_basis()
[4*y^4 - 4*y^2*z^2 + z^4 - 4*y^2*w^2 + 2*z^2*w^2 + w^4,
  2*x*y^2 - 2*y^3 - x*z^2 + y*z^2 - x*w^2 + y*w^2,
  x^2 - 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()
Two examples from the Mathematica documentation (done in Sage):

We compute a Groebner basis:

\begin{verbatim}
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
\end{verbatim}

We show that three polynomials have no common root:

\begin{verbatim}
sage: R.<x,y> = QQ[]
sage: ideal(x+y, x^2 - 1, y^2 - 2*x).groebner_basis()
[1]
\end{verbatim}

The next example shows how we can use Groebner bases over \( \mathbb{Z} \) to find the primes modulo which a system of equations has a solution, when the system has no solutions over the rationals.

We first form a certain ideal \( I \) in \( \mathbb{Z}[x, y, z] \), and note that the Groebner basis of \( I \) over \( \mathbb{Q} \) contains 1, so there are no solutions over \( \mathbb{Q} \) or an algebraic closure of it (this is not surprising as there are 4 equations in 3 unknowns).

\begin{verbatim}
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]
\end{verbatim}

However, when we compute the Groebner basis of \( I \) (defined over \( \mathbb{Z} \)), we note that there is a certain integer in the ideal which is not 1.

\begin{verbatim}
sage: I.groebner_basis()
[ x + 130433*y + 59079*z, y^2 + 3*y + 17220, y*z + 5*y + 14504, 2*y + 158864, \n 2*z + 41856, 164878]
\end{verbatim}

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 \( p \).

\begin{verbatim}
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]
\end{verbatim}
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 + 9*y + 13*z, y^2 + 3*y, y*z + 7*y + 6, 2*y + 6, z^2 + 3, 2*z + 10]
```

Modulo any other prime the Groebner basis is trivial so there are no other solutions. For example:

```
sage: I.change_ring( P.change_ring( GF(3) ) ).groebner_basis()
[1]
```

Note: Sage distinguishes between lists or sequences of polynomials and ideals. Thus an ideal is not identified with a particular set of generators. For sequences of multivariate polynomials see `sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic`.

```python
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
```

```python
basis
    Shortcut to gens().

    EXAMPLES:

    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 g1,...,gn of self as returned by self.gens().
```

308 Chapter 3. Multivariate Polynomials
INPUT:

- P - a multivariate polynomial ring

EXAMPLES:

```python
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]
```

```python
sage: Q.<x,y,z> = P.change_ring(order='degrevlex'); Q
Multivariate Polynomial Ring in x, y, z over Rational Field
sage: Q.term_order()
Degree reverse lexicographic term order

sage: J = I.change_ring(Q); J
Ideal (x + y + z, x*y + x*z + y*z, x*y*z - 1) of
Multivariate Polynomial Ring in x, y, z over Rational Field

sage: J.groebner_basis()
[z^3 - 1, y^2 + y*z + z^2, x + y + z]
```

`degree_of_semi_regularity()`

Return the degree of semi-regularity of this ideal under the assumption that it is semi-regular.

Let \( \{f_1, \ldots, f_m\} \subset K[x_1, \ldots, x_n] \) be homogeneous polynomials of degrees \( d_1, \ldots, d_m \) respectively. This sequence is semi-regular if:

1. \( \{f_1, \ldots, f_m\} \neq K[x_1, \ldots, x_n] \)
2. for all \( 1 \leq i \leq m \) and \( g \in K[x_1, \ldots, x_n] \): \( \deg(g \cdot p_i) < D \) and \( g \cdot f_i \in < f_1, \ldots, f_{i-1} > \) implies that \( g \in < f_1, \ldots, f_{i-1} > \) where \( D \) is the degree of regularity.

This notion can be extended to affine polynomials by considering their homogeneous components of highest degree.

The degree of regularity of a semi-regular sequence \( f_1, \ldots, f_m \) of respective degrees \( d_1, \ldots, d_m \) is given by the index of the first non-positive coefficient of:

\[
\sum_{k} c_k z^k = \prod_{i=1}^{n} \frac{(1-z^{d_i})}{(1-z)^n}
\]

EXAMPLES:

We consider a homogeneous example:

```python
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())
```

(continues on next page)
From this, we expect a Groebner basis computation to reach at most degree 4. For homogeneous systems this is equivalent to the largest degree in the Groebner basis:

```
sage: max(f.degree() for f in I.groebner_basis())
4
```

We increase the number of polynomials and observe a decrease the degree of regularity:

```
sage: for i in range(2*n):
    ....:     f = P.random_element(degree=2, terms=binomial(n,2))
    ....:     f -= f(*s)
    ....:     L.append(f.homogenize())
sage: I = Ideal(L)
sage: I.degree_of_semi_regularity()
3
sage: max(f.degree() for f in I.groebner_basis())
3
```

The degree of regularity approaches 2 for quadratic systems as the number of polynomials approaches $n^2$:

```
sage: for i in range((n-4)*n):
    ....:     f = P.random_element(degree=2, terms=binomial(n,2))
    ....:     f -= f(*s)
    ....:     L.append(f.homogenize())
sage: I = Ideal(L)
sage: I.degree_of_semi_regularity()
2
sage: max(f.degree() for f in I.groebner_basis())
2
```

**Note:** It is unknown whether semi-regular sequences exist. However, it is expected that random systems are semi-regular sequences. For more details about semi-regular sequences see [BFS2004].

```python
gens()  
Return a set of generators / a basis of this ideal. This is usually the set of generators provided during object creation.

**EXAMPLES:**

```
sage: P.<x,y> = PolynomialRing(QQ,2)
sage: I = Ideal([x,y+1]); I
Ideal (x, y + 1) of Multivariate Polynomial Ring in x, y over Rational Field
sage: I.gens()
[x, y + 1]
```

groebner_basis(\textit{algorithm} = \textit{"", deg_bound=\textit{None}, mult_bound=\textit{None}, prot=\textit{False}, *args, **kwds})  
Return the reduced Groebner basis of this ideal.

A Groebner basis $g_1, \ldots, g_n$ for an ideal $I$ is a generating set such that \( <LM(g_i) >= LM(I) \), i.e., the leading monomial ideal of $I$ is spanned by the leading terms of $g_1, \ldots, g_n$. Groebner bases are the key
concept in computational ideal theory in multivariate polynomial rings which allows a variety of problems to be solved.

Additionally, a reduced Groebner basis $G$ is a unique representation for the ideal $< G >$ with respect to the chosen monomial ordering.

**INPUT:**

- **algorithm** - determines the algorithm to use, see below for available algorithms.
- **deg_bound** - only compute to degree $deg\_bound$, 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_\texttt{verbose}().
- ***args** - additional parameters passed to the respective implementations
- ****kwds** - additional keyword parameters passed to the respective implementations

**ALGORITHMS:**

- ‘autoselect’ (default)
- ‘singular:groebner’  Singular’s groebner command
- ‘singular:std’  Singular’s std command
- ‘singular:stdhilb’  Singular’s stdhilb command
- ‘singular:stdfglm’  Singular’s stdfglm command
- ‘singular:slimgb’  Singular’s slimgb command
- ‘libSingular:groebner’  libSingular’s groebner command
- ‘libSingular:std’  libSingular’s std command
- ‘libSingular:slimgb’  libSingular’s slimgb command
- ‘libSingular:stdhilb’  libSingular’s stdhilb command
- ‘libSingular:stdfglm’  libSingular’s stdfglm command
- ‘toy:buchberger’  Sage’s toy/educational buchberger without Buchberger criteria
- ‘toy:buchberger2’  Sage’s toy/educational buchberger with Buchberger criteria
- ‘toy:d\_basis’  Sage’s toy/educational algorithm for computation over PIDs
- ‘macaulay2:gb’  Macaulay2’s gb command (if available)
- ‘macaulay2:f4’  Macaulay2’s GroebnerBasis command with the strategy “F4” (if available)
- ‘macaulay2:mgb’  Macaulay2’s GroebnerBasis command with the strategy “MGB” (if available)
- ‘magma:GroebnerBasis’  Magma’s Groebnerbasis command (if available)
- ‘ginv:TQ’, ‘ginv:TQBlockHigh’, ‘ginv:TQBlockLow’ and ‘ginv:TQDegree’  One of GINV’s implementations (if available)
- ‘giac:gbasis’  Giac’s gbasis command (if available)
If only a system is given - e.g. ‘magma’ - the default algorithm is chosen for that system.

**Note:** The Singular and libSingular versions of the respective algorithms are identical, but the former calls an external Singular process while the latter calls a C function, i.e. the calling overhead is smaller. However, the libSingular interface does not support pretty printing of computation protocols.

**EXAMPLES:**

Consider Katsura-3 over \(\mathbb{Q}\) with lexicographical term ordering. We compute the reduced Groebner basis using every available implementation and check their equality.

```python
sage: P.<a,b,c> = PolynomialRing(QQ,3, order='lex')
sage: I = sage.rings.ideal.Katsura(P,3)  # regenerate to prevent caching
dsage: I.groebner_basis()
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c^2 + 1/84*c]
```

```python
sage: I = sage.rings.ideal.Katsura(P,3)  # regenerate to prevent caching
dsage: I.groebner_basis('libsingular:groebner')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c^2 + 1/84*c]
```

```python
sage: I = sage.rings.ideal.Katsura(P,3)  # regenerate to prevent caching
dsage: I.groebner_basis('libsingular:std')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c^2 + 1/84*c]
```

```python
sage: I = sage.rings.ideal.Katsura(P,3)  # regenerate to prevent caching
dsage: I.groebner_basis('libsingular:stdhilb')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c^2 + 1/84*c]
```

```python
sage: I = sage.rings.ideal.Katsura(P,3)  # regenerate to prevent caching
dsage: I.groebner_basis('libsingular:stdfglm')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c^2 + 1/84*c]
```

```python
sage: I = sage.rings.ideal.Katsura(P,3)  # regenerate to prevent caching
dsage: I.groebner_basis('libsingular:slimgb')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c^2 + 1/84*c]
```

Although Giac does support lexicographical ordering, we use degree reverse lexicographical ordering here, in order to test against trac ticket #21884:

```python
sage: I = sage.rings.ideal.Katsura(P,3)  # regenerate to prevent caching
dsage: J = I.change_ring(P.change_ring(order='degrevlex'))
dsage: gb = J.groebner_basis('giac')  # optional - giacpy_sage, random

d sage: gb = gb.op tional - giacpy_s age
[c^3 - 79/210*c^2 + 1/30*b + 1/70*c, b^2 - 3/5*c^2 - 1/5*b + 1/5*c, b*c + 6/5*c^2 - 1/10*b - 2/5*c, a + 2*b + 2*c - 1]

d sage: J.groebner_basis.set_cache(gb)  # optional - giacpy_sage

d sage: ideal(J.transformed_basis()).change_ring(P).interreduced_basis()  # testing trac 21884
```

(continues on next page)
Giac’s gbasis over $\mathbb{Q}$ can benefit from a probabilistic lifting and multi threaded operations:

```python
sage: A9=PolynomialRing(QQ,9,'x') # optional - giacpy_sage
sage: I9=sage.rings.ideal.Katsura(A9) # optional - giacpy_sage
sage: I9.groebner_basis("giac",proba_epsilon=1e-7) # optional - giacpy_sage,
```

...Running a probabilistic check for the reconstructed Groebner basis...
Polynomial Sequence with 143 Polynomials in 9 Variables

The list of available Giac options is provided at `sage.libs.giac.groebner_basis()`.

Note that `toy:buchberger` does not return the reduced Groebner basis,

```python
sage: gb = I.groebner_basis('toy:buchberger'); gb
```

...but that `toy:buchberger2` does.

```python
sage: gb == gb.reduced()
True
```

Here we use Macaulay2 with three different strategies over a finite field.

```python
sage: R.<a,b,c> = PolynomialRing(GF(101), 3)
sage: I = sage.rings.ideal.Katsura(R,3) # regenerate to prevent caching
sage: I.groebner_basis('macaulay2:gb') # optional - macaulay2
```

Singular and libSingular can compute Groebner basis with degree restrictions.
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.

The list of available options is provided at `LibSingularOptions`.

Note that Groebner bases over \( \mathbb{Z} \) can also be computed.

Groebner bases over \( \mathbb{Z}/n\mathbb{Z} \) are also supported:
.. code-block::

    sage: P.<a,b,c> = PolynomialRing(Zmod(1000),3)
    sage: I = P * (a + 2*b + 2*c - 1, a^2 - a + 2*b^2 + 2*c^2, 2*a*b + 2*b*c - b)
    sage: I.groebner_basis()
    [b*c^2 + 732*b*c + 808*b,
     2*c^3 + 884*b*c + 666*c^2 + 320*b,
     b^2 + 438*b*c + 281*b,
     5*b*c + 156*c^2 + 948*c,
     50*c^2 + 600*b + 650*c,
     a + 2*b + 2*c + 999,
     125*b]

    sage: R.<x,y,z> = PolynomialRing(Zmod(2233497349584))
    sage: I = R.ideal([z*(x-3*y), 3^2*x^2-y*z, z^2+y^2])
    sage: I.groebner_basis()
    [2*z^4, y*z^2 + 81*z^3, 248166372176*z^3, 9*x^2 - y*z, y^2 + z^2, x*z + 2233497349581*y*z, 248166372176*y*z]

Sage also supports local orderings:

    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 \texttt{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 + 95*x*y*z^2
    sage: f.lift(I.gens())
    [2*x + 95*z, 1/2*y^2 - 1/4*y*z, 0]
    sage: f = f.lift(J.gens()); f
    [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 + 95*z]
    sage: sum(map(mul, zip(f,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 - 79/70)*Z^2 + (2/15*t^2 - 74/315*t + 94/105)*Y + (2/35*t^2 - 194/315*t + 1/105)*Y^2 - 4/105*t^2 - 17/210*t + 1/28, Y^2 + (-3/5)*Z^2 + (-2/5*t - 2)*Z + (-2/5*t + 2/5)*Z^2 - 4/15*t + 2/5 + (-2/5*t + 5)*Y + (2/5*t + 3/5)*Y - 4/15*t + 2/5, X + 2*Y + 2*Z - t - 2]

In cases where a characteristic cannot be determined, we use a toy implementation of Buchberger's algorithm (see \texttt{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()

ALGORITHM:
Uses Singular, Magma (if available), Macaulay2 (if available), Giac (if available), or a toy implementation.

groebner_fan (is_groebner_basis=False, symmetry=None, verbose=False)
Return the Groebner fan of this ideal.
The base ring must be \( \mathbb{Q} \) or a finite field \( \mathbb{F}_p \) of with \( p \leq 32749 \).

EXEMPLARY:

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’)

EXEMPLARY:

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 + z + 1, y^2 + x + h^2 + 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
is_homogeneous()  
Return True if this ideal is spanned by homogeneous polynomials, i.e. if it is a homogeneous ideal.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ,3)
sage: I = sage.rings.ideal.Katsura(P)
sage: I
Ideal (x + 2*y + 2*z - 1, x^2 + 2*y^2 + 2*z^2 - x, 2*x*y + 2*y*z - y) of Multivariate Polynomial Ring in x, y, z over Rational Field
```

```
sage: I.is_homogeneous()  
False
```

```
sage: J = I.homogenize()  
sage: J
Ideal (x + 2*y + 2*z - h, x^2 + 2*y^2 + 2*z^2 - x*h, 2*x*y + 2*y*z - y*h) of Multivariate Polynomial Ring in x, y, z, h over Rational Field
```

```
sage: J.is_homogeneous()  
True
```

plot (*args, **kwds)  
Plot the real zero locus of this principal ideal.

INPUT:

- self - a principal ideal in 2 variables
- algorithm - set this to 'surf' if you want 'surf' to plot the ideal (default: None)
- *args - optional tuples (variable, minimum, maximum) for plotting dimensions
- **kwds - optional keyword arguments passed on to implicit_plot

EXAMPLES:

Implicit plotting in 2-d:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: I = R.ideal([y^3 - x^2])
sage: I.plot()  
# cusp
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
```
 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()`.

EXAMPLES:

We compute a uniformly random element up to the provided degree.

```python
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:

```python
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:

```python
sage: F = Sequence(I.random_element(degree=d, compute_gb=True, terms=infinity) for _ in range(n^d))
```

2. We linearize and compute the echelon form:

```python
sage: A,v = F.coefficient_matrix()
sage: A.echelonize()
```

3. The result is the desired Gröbner basis:
We return some element in the ideal with no guarantee on the distribution:

```python
sage: P = PolynomialRing(GF(127), 10, 'x')
sage: I = sage.rings.ideal.Katsura(P)
sage: f = I.random_element(degree=3)
sage: f
# random
-25*x0^2*x1 + 14*x1^3 + 57*x0*x1*x2 + ... + 19*x7*x9 + 40*x8*x9 + 49*x1
sage: f.degree()
3
```

We show that the default method does not sample uniformly at random from the ideal:

```python
sage: P.<x,y,z> = GF(127)[]
sage: G = Sequence([x+7, y-2, z+110])
sage: I = Ideal([sum(P.random_element() * g for g in G) for _ in range(4)])
sage: all(I.random_element(degree=1) == 0 for _ in range(100))
True
```

If degree equals the degree of the generators a random linear combination of the generators is returned:

```python
sage: P.<x,y> = QQ[]
sage: I = P.ideal([x^2,y^2])
sage: set_random_seed(5)
sage: I.random_element(degree=2)
-2*x^2 + 2*y^2
```

**reduce** *(f)*

Reduce an element modulo the reduced Groebner basis for this ideal. This returns 0 if and only if the element is in this ideal. In any case, this reduction is unique up to monomial orders.

**EXAMPLES:**

```python
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: I = (x^3 + y, y)*R
sage: I.reduce(y)
0
sage: I.reduce(x^3)
0
sage: I.reduce(x - y)
x
sage: I = (y^2 - (x^3 + x))*R
sage: I.reduce(x^3)
y^2 - x
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:**

```python
sage: R.<x,y> = PolynomialRing(ZZ,2,'xy')
sage: I = R.ideal(x^5+y^5, x^2 + y + x^2*y^2 + 5); I
Ideal (x^5 + y^5, x^2*y^2 + x^2 + y + 5) of Multivariate Polynomial Ring in x, y over Integer Ring
sage: I.subs(x=y)
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:

```python
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 multivariate polynomial ring:

```python
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
```

Variables that are not substituted remain unchanged:

```python
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 $\text{Res}_{L/k}$ - is a functor which, for any finite extension of fields $L/k$ and any algebraic variety $X$ over $L$, produces another corresponding variety $\text{Res}_{L/k}(X)$, defined over $k$. It is useful for reducing questions about varieties over large fields to questions about more complicated varieties over smaller fields.

This function does not compute this Weil restriction directly but computes on generating sets of polynomial ideals:

Let $d$ be the degree of the field extension $L/k$, let $a$ a generator of $L/k$ and $p$ the minimal polynomial of $L/k$. Denote this ideal by $I$.

Specifically, this function first maps each variable $x$ to its representation over $k$: $\sum_{i=0}^{d-1} a^i x_i$. Then each generator of $I$ is evaluated over these representations and reduced modulo the minimal polynomial $p$. The result is interpreted as a univariate polynomial in $a$ and its coefficients are the new generators of the returned ideal.

If the input and the output ideals are radical, this is equivalent to the statement about algebraic varieties above.

OUTPUT: MPolynomial Ideal

EXAMPLES:

```python
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()  # py2
{(y1: 1, x1: 1, x0: 1, y0: 0)}
sage: J.variety()  # py3
{(y1: 1, y0: 0, x1: 1, x0: 1)}
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
```

```python
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()  # py2
{(y: 0, z: 0, x: 1)}
sage: I.variety()  # py3
{(z: 0, y: 0, x: 1)}
sage: J = I.weil_restriction(); J
```

(continues on next page)
Weil restrictions are often used to study elliptic curves over extension fields so we give a simple example involving those:

```python
sage: K.<a> = QuadraticField(1/3)
sage: E = EllipticCurve(K,[1,2,3,4,5])
```

We pick a point on $E$:

```python
sage: p = E.lift_x(1); p
(1 : 2 : 1)
sage: I = E.defining_ideal(); I
Ideal (-x^3 - 2*x^2*z + x*y*z + y^2*z - 4*x*z^2 + 3*y*z^2 - 5*z^3) of
  Multivariate Polynomial Ring in x, y, z over Number Field in a with
  defining polynomial x^2 - 1/3 with a = 0.5773502691896258?
```

Of course, the point $p$ is a root of all generators of $I$:

```python
sage: I.subs(x=1,y=2,z=1)
Ideal (0) of Multivariate Polynomial Ring in x, y, z over Number Field in a with
  defining polynomial x^2 - 1/3 with a = 0.5773502691896258?
```

$I$ is also radical:

```python
sage: I.radical() == I
True
```

So we compute its Weil restriction:

```python
sage: J = I.weil_restriction()
sage: J
(continues on next page)
```
Ideal \((-x_0^3 - x_0*x_1^2 - 2*x_0^2*z_0 - 2/3*x_1^2*z_0 + x_0*y_0*z_0 + y_0^2*z_0 + \\
1/3*x_1*y_1*z_0 + 1/3*y_1^2*z_0 - 4*x_0*z_0^2 + 3*y_0*z_0^2 - 5*z_0^3 - \\
4/3*x_0*x_1*z_1 + 1/3*x_1*y_0*z_1 + 1/3*x_0*y_1*z_1 + 2/3*y_0*y_1*z_1 - 8/3*x_1*z_0*z_1 \\
+ 2*y_1*z_0*z_1 - 4/3*x_0^2*z_1^2 + y_0*z_1^2 + 5*z_0*z_1^2, -3*x_0*x_1^2 + \\
- 4*x_0*x_1*z_0 + x_1*y_0*z_0 + x_0*y_1*z_0 + 2*y_0*y_1*z_0 - 4*x_1*z_0^2 + 3*y_1*z_0^2 \\
- 2*x_0^2*z_1 - 2/3*x_1^2*z_1 - x_0*y_0*z_1 + y_0^2*z_1 + 1/3*x_1*y_1*z_1 + \\
1/3*y_1^2*z_1 - 8*x_0*z_0*z_1 + 6*y_0*z_0*z_1 - 15*z_0^2*z_1 + 4/3*x_1*z_1^2 + \\
y_1*z_1^2 - 5/3*z_1^3)\) of Multivariate Polynomial Ring in \(x_0, x_1, y_0, y_1, \\
z_0, z_1\) 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 (x^2*y - w*z^2, y^2 - w*x - 1, -3*x^0^2 + x_1^2 + x_1^3 + 2*y_0*y_1)
```

Note: Based on a Singular implementation by Michael Brickenstein

---

### 3.1. Multivariate Polynomials and Polynomial Rings

---
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])

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 \( I = \text{self} \).

An ideal \( Q \) is called primary if it is a proper ideal of the ring \( R \) and if whenever \( ab \in Q \) and \( a \notin Q \) then \( b^n \in Q \) for some \( n \in \mathbb{Z} \).

If \( Q \) is a primary ideal of the ring \( R \), then the radical ideal \( P \) of \( Q \), i.e. \( P = \{a \in R, a^n \in Q \} \) for some \( n \in \mathbb{Z} \), is called the associated prime of \( Q \).

If \( I \) is a proper ideal of the ring \( R \) then there exists a decomposition in primary ideals \( Q_i \) such that

- their intersection is \( I \)
- none of the \( Q_i \) contains the intersection of the rest, and
- the associated prime ideals of \( Q_i \) are pairwise different.

This method returns the associated primes of the \( Q_i \).

INPUT:

- \( \text{algorithm} \) - 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^2 + 1, y + 1) of Multivariate Polynomial Ring in x, y, z over Rational Field,
 Ideal (z^3 + 2, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field]

ALGORITHM:

Uses Singular.
REFERENCES:


**basis_is_groebner**(singular=Singular)

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 $Syz(LM(I))$:

$$S \ast G = \sum_{i=0}^{m} h_i g_i > G 0.$$  

ALGORITHM:

Uses Singular.

EXAMPLES:

```python
sage: R.<a,b,c,d,e,f,g,h,i,j> = PolynomialRing(GF(127),10)
sage: I = sage.rings.ideal.Cyclic(R,4)
sage: I.basis_is_groebner()  
False
sage: I2 = Ideal(I.groebner_basis())
sage: I2.basis_is_groebner()  
True
```

A more complicated example:

```python
sage: R.<U6,U5,U4,U3,U2, u6,u5,u4,u3,u2, h> = PolynomialRing(GF(7583))
sage: I = [u6 + u5 + u4 + u3 + u2 - 3791*h, 
          U6 + U5 + U4 + U3 + U2 - 3791*h, 
          U2*u2 - h^2, U3*u3 - h^2, U4*u4 - h^2, 
          U5*u4 + U5*u3 + U4*u3 + U5*u2 + U4*u2 + U3*u2 - 3791*U5*h - 
          3791*U4*h - 3791*U3*h - 3791*U2*h - 2842*h^2, 
          U4*u5 + U3*u5 + U2*u5 + U3*u4 + U2*u4 + U2*u3 - 3791*u5*h - 
          3791*u4*h - 3791*u3*h - 3791*u2*h - 2842*h^2, 
          U5*u5 - h^2, U4*u2*u3 + U5*u3*u2 + U4*u2*u3 + U3^2*u2 - 
          3791*U5*U4*U3*h - 3791*U4*U3*U2*h - 3791*U5*U2*h - 
          3791*U4*U2*h + U3*U2*h - 3791*U2*U2*h - 3791*U4*U3*h - 
          3791*U4*U2*h - 3791*U3*u2*h - 2843*U5*U3^2*h + 1897*U4*U3*h^2 + 948*U3*u2^2 + 
          947*U2*u2^2 + 2370*h^3, 
          U3*u5*u4 + U2*u5*u4 + U3*u4^2 + U2*u4^2 + U2*u4*u3 - 3791*u5*u4*h - 
          3791*u4^2*h - 3791*u4*u3*h - 3791*u4*u2*h + u5*h^2 - 2842*u4*h^2, 
          U2*u5*u4*u3 + U2*u4^2*u3 + U2*u4*u3^2 - 3791*u5*u4*h + 3791*u4^4 - 
          2*u3*h - 3791*u4*u3^2*h - 3791*u4*u3*u2*h + u5*u4*h^2 + u4^2*h^2 + 2*u5*u3*h^2 - 
          2 - 2842*u4*u3*h^2, 
          U5^2*U4*u3 + U5*U4^2*u3 + U5^2*U4*u2 + U5^4*u2 + U5^2*U3*u2 + 
          2*U5*U4*U3*u2 + U5*U3*u2^2 - 3791*U5*U2*U4*h - 3791*U5*U4^2*h + 3791*U5^2*U4*h + 
          2*U3*h 
          + U5*U4*U3*h - 3791*U5*U3^2*h - 3791*U5*U2*h + U5*U4*U2*h + 
          U5*U3*U2*h - 3791*U5*U2^2*h - 3791*U5*U3*u2*h - 2842*U5*U2*h^2 + 
          1897*U5*U4*h^2 + 
          - U4^2*h^2 - 947*U5*U3^2*h^2 - U4*U3^2*h^2 - 948*U5*U2^2*h^2 - U4*U2*U2*h^2 + 
          - 2 - 1422*U5*U4^3 + 3791*U4^2*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*u4^2*u3*h^2 + 2*u4*u3^2*h^2 + 2*u5*u4*u2*h^2 + 2*u4^2 + 
          2*u2*h^2 + 
          + 2*u5*u3*u2*h^2 + 1899*U4*u3*u2*h^2, 
```

(continues on next page)
U5^2*U4*U3*u2 + U5*U4^2*U3*u2 + U5*U4*U3^2*u2 - 3791*U5^2*U4*U3*h - 
\rightarrow 3791*U5*U4^2*U3*h - 3791*U5*U4*U3^2*h - 3791*U5*U4*U3*u2*h \ 
+ 3791*U5*U4*U3*u2*h + U5^2*U4*h^2 + U5*U4^2*h^2 + U5^2*U3*h^2 - 
\rightarrow U4^2*U3*h^2 - U5*U3^2*h^2 - U4*U3^2*h^2 - U5*U4*U2*h^2 \ 
- U5*U3*U2*h^2 - U4*U3*U2*h^2 + 3791*U5*U4*U3*h^3 + 3791*U5*U3*h^3 + 
\rightarrow 3791*U4*U3*h^3, \ 
\rightarrow u4^2*u3*u2*h^2 + 1515*u5*u3^2*u2*h^2 + u4*u3^2*u2*h^2 + 
\rightarrow 1515*u5*u4*u2^2*h^2 + 1515*u5*u3*u2^2*h^2 + u4*u3*u2^2*h^2 \ 
+ 1521*u5*u4*u3*h^3 - 3028*u4^2*u3*h^3 - 3028*u4*u3^2*h^3 + 
\rightarrow 1521*u5*u4*U2^3*h^3 + 3420*u4*u3*U2^3*h^3 - 3, \ 
\rightarrow U5^2*U4*U3*U2*h + U5*U4^2*U3*U2*h + U5*U4*U3^2*U2*h + U5*U4*U3*U2^ 
\rightarrow 2*h + 2*U5^2*U4*U3*h^2 + 2*U5*U4^2*U3*h^2 + 2*U5*U4*U3^2*h^2 + 
\rightarrow 2*U5*U4*U2*h^2 + 2*U5*U4^2*U2*h^2 + 2*U5*U3*U2*h^2 + 
\rightarrow 2*U5*U3^2*U2*h^2 - 2*U5*U4*U2^2*h^2 - 2*U5*U3*U2^2*h^2 - 
\rightarrow U5*U4*U3^2*h^3 - U5*U4*U2^3*h^3 - U5*U3*U2^3*h^3 - U4*U3*U2^3*h^3 
\rightarrow [\text{sage: } \text{Ideal(l).basis_is_groebner() \nFalse \n\text{sage: } \text{gb = Ideal(l).groebner_basis() \n\text{sage: } \text{Ideal(gb).basis_is_groebner() \nTrue}]

Note: From the Singular Manual for the reduce function we use in this method: ‘The result may have no meaning if the second argument (self) is not a standard basis’. I (malb) believe this refers to the mathematical fact that the results may have no meaning if self is not a standard basis, i.e., Singular doesn’t ‘add’ any additional ‘nonsense’ to the result. So we may actually use reduce to determine if self is a Groebner basis.

**complete_primary_decomposition**(object)
A decorator that creates a cached version of an instance method of a class.

Note: For proper behavior, the method must be a pure function (no side effects). Arguments to the method must be hashable or transformed into something hashable using key or they must define sage.structure.sage_object.SageObject._cache_key().

**EXAMPLES:**

```python
sage: class Foo(object):
    ....: @cached_method
    ....: def f(self, t, x=2):
        ....:     print('computing')
        ....:     return t**x
sage: a = Foo()
```

The example shows that the actual computation takes place only once, and that the result is identical for equivalent input:

```python
sage: res = a.f(3, 2); res
computing 9
sage: a.f(t = 3, x = 2) is res
(continues on next page)```
True

**sage:** `a.f(3)` is `res`  
True

Note, however, that the `CachedMethod` is replaced by a `CachedMethodCaller` or `CachedMethodCallerNoArgs` as soon as it is bound to an instance or class:

```python
sage: P.<a,b,c,d> = QQ[]
sage: I = P*[a,b]
sage: type(I.__class__.gens)
<type 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
```

So, you would hardly ever see an instance of this class alive.

The parameter `key` can be used to pass a function which creates a custom cache key for inputs. In the following example, this parameter is used to ignore the `algorithm` keyword for caching:

```python
sage: class A(object):
....:     def _f_normalize(self, x, algorithm):
....:         return x
....:     @cached_method(key=_f_normalize)
....:     def f(self, x, algorithm='default'):
....:         return x
sage: a = A()
sage: a.f(1, algorithm="default") is a.f(1) is a.f(1, algorithm="algorithm")
True
```

The parameter `do_pickle` can be used to enable pickling of the cache. Usually the cache is not stored when pickling:

```python
sage: class A(object):
....:     @cached_method
....:     def f(self, x):
....:         return None
sage: import __main__
sage: __main__.A = A
sage: a = A()
sage: a.f(1)
len(a.f.cache)
1
b = loads(dumps(a))
len(b.f.cache)
0
```

When `do_pickle` is set, the pickle contains the contents of the cache:

```python
sage: class A(object):
....:     @cached_method(do_pickle=True)
....:     def f(self, x):
....:         return None
sage: __main__.A = A
sage: a = A()
sage: a.f(1)
len(a.f.cache)
1
b = loads(dumps(a))
len(b.f.cache)
1
```

Cached methods can not be copied like usual methods, see trac ticket #12603. Copying them can lead to very surprising results:

3.1. Multivariate Polynomials and Polynomial Rings
sage: class A:
....:   @cached_method
....:   def f(self):
....:       return 1
sage: class B:
....:       g = A.f
....:       def f(self):
....:           return 2
sage: b = B()
sage: b.f()
2
sage: b.g()
1
sage: b.f()
1

dimension (singular='singular_default')
The dimension of the ring modulo this ideal.

EXAMPLES:

sage: P.<x,y,z> = PolynomialRing(GF(32003),order='degrevlex')
sage: I = ideal(x^2-y,x^3)
sage: I.dimension()
1

If the ideal is the total ring, the dimension is \(-1\) by convention.

For polynomials over a finite field of order too large for Singular, this falls back on a toy implementation of Buchberger to compute the Groebner basis, then uses the algorithm described in Chapter 9, Section 1 of Cox, Little, and O’Shea’s “Ideals, Varieties, and Algorithms”.

EXAMPLES:

sage: R.<x,y> = PolynomialRing(GF(2147483659),order='lex')
sage: I = R.ideal([x*y,x*y+1])
sage: I.dimension()
-1
sage: I = ideal([x*(x*y+1),y*(x*y+1)])
sage: I.dimension()
1
sage: I = R.ideal([x^3*y,x*y^2])
sage: I.dimension()
1
sage: R.<x,y> = PolynomialRing(GF(2147483659),order='lex')
sage: I = R.ideal(0)
sage: I.dimension()
2

ALGORITHM:
Uses Singular, unless the characteristic is too large.

**Note:** Requires computation of a Groebner basis, which can be a very expensive operation.

**elimination_ideal**(variables, algorithm=None, *args, **kwds)**

Return the elimination ideal of this ideal with respect to the variables given in variables.

**INPUT:**
- variables - a list or tuple of variables in self.ring()
- algorithm - determines the algorithm to use, see below for available algorithms.

**ALGORITHMS:**
- 'libsingular:eliminate' - libSingular's eliminate command (default)
- 'giac:eliminate' - Giac's eliminate command (if available)

If only a system is given - e.g. 'giac' - the default algorithm is chosen for that system.

**EXAMPLES:**

```python
sage: R.<x,y,t,s,z> = PolynomialRing(QQ,5)
sage: I = R * [x-t,y-t^2,z-t^3,s-x+y^3]
sage: J = I.elimination_ideal([t,s]); J
Ideal (y^2 - x*z, x*y - z, x^2 - y) of Multivariate Polynomial Ring in x, y, t, s, z over Rational Field
```

You can use Giac to compute the elimination ideal:

```python
sage: I.elimination_ideal([t, s], algorithm="giac") == J  # optional - giacpy
\rightarrow sage
... Running a probabilistic check for the reconstructed Groebner basis...
True
```

The list of available Giac options is provided at `sage.libs.giac.groebner_basis()`.

**ALGORITHM:**
Uses Singular, or Giac (if available).

**Note:** Requires computation of a Groebner basis, which can be a very expensive operation.

**genus**(object)

A decorator that creates a cached version of an instance method of a class.

**Note:** For proper behavior, the method must be a pure function (no side effects). Arguments to the method must be hashable or transformed into something hashable using key or they must define `sage.structure.sage_object.SageObject._cache_key()`.

**EXAMPLES:**

```python
sage: class Foo(object):
    ....:     @cached_method
    ....:     def f(self, t, x=2):
```

(continues on next page)
The example shows that the actual computation takes place only once, and that the result is identical for equivalent input:

```
sage: res = a.f(3, 2); res
computing
9
sage: a.f(t = 3, x = 2) == res
True
sage: a.f(3) == res
True
```

Note, however, that the `CachedMethod` is replaced by a `CachedMethodCaller` or `CachedMethodCallerNoArgs` as soon as it is bound to an instance or class:

```
sage: P.<a,b,c,d> = QQ[]
sage: I = P*[a,b]
sage: type(I.__class__.gens)
<type 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
```

So, you would hardly ever see an instance of this class alive.

The parameter `key` can be used to pass a function which creates a custom cache key for inputs. In the following example, this parameter is used to ignore the `algorithm` keyword for caching:

```
sage: class A(object):
    ....:    def _f_normalize(self, x, algorithm):
    ....:        return x
    ....:    @cached_method(key=_f_normalize)
    ....:    def f(self, x, algorithm='default'):
    ....:        return x
sage: a = A()
sage: a.f(1, algorithm="default") == a.f(1) == a.f(1, algorithm="algorithm")
True
```

The parameter `do_pickle` can be used to enable pickling of the cache. Usually the cache is not stored when pickling:

```
sage: class A(object):
    ....:    @cached_method
    ....:    def f(self, x):
    ....:        return None
sage: import __main__
sage: __main__.A = A()
sage: a = A()
sage: a.f(1)
```

```
len(a.f.cache)
1
b = loads(dumps(a))
len(b.f.cache)
0
```

When `do_pickle` is set, the pickle contains the contents of the cache:

```
sage: class A(object):
    ....:    @cached_method(do_pickle=True)
```
Cached methods can not be copied like usual methods, see trac ticket #12603. Copying them can lead to very surprising results:

```python
sage: class A:
    ....:     @cached_method
    ....:     def f(self):
    ....:         return 1
sage: class B:
    ....:     g=A.f
    ....:     def f(self):
    ....:         return 2
sage: b=B()
sage: b.f()
2
sage: b.g()
1
sage: b.f()
1
```

**hilbert_numerator** *(grading=None, algorithm='sage')*

Return the Hilbert numerator of this ideal.

**INPUT:**

- **grading** — (optional) a list or tuple of integers
- **algorithm** — (default: 'sage') must be either 'sage' or 'singular'

Let $I$ (which is `self`) be a homogeneous ideal and $R = \bigoplus_d R_d$ (which is `self.ring()`) be a graded commutative algebra over a field $K$. Then the **Hilbert function** is defined as $H(d) = \dim_K R_d$ and the **Hilbert series** of $I$ is defined as the formal power series $HS(t) = \sum_{d=0}^{\infty} H(d)t^d$.

This power series can be expressed as $HS(t) = Q(t)/(1 - t)^n$ where $Q(t)$ is a polynomial over $Z$ and $n$ the number of variables in $R$. This method returns $Q(t)$, the numerator; hence the name, `hilbert_numerator`. An optional grading can be given, in which case the graded (or weighted) Hilbert numerator is given.

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([x^3*y^2 + 3*x^2*y^2*z + y^3*z^2 + z^5])
sage: I.hilbert_numerator()
-t^5 + 1
sage: R.<a,b> = PolynomialRing(QQ)
sage: J = R.ideal([a^2*b,a*b^2])
sage: J.hilbert_numerator()
t^4 - 2*t^3 + 1
```

(continues on next page)
hilbert_polynomial(algorithm='sage')

Return the Hilbert polynomial of this ideal.

INPUT:

- algorithm — (default: 'sage') must be either 'sage' or 'singular'

Let $I$ (which is self) be a homogeneous ideal and $R = \bigoplus_d R_d$ (which is self.ring()) be a graded commutative algebra over a field $K$. The **Hilbert polynomial** is the unique polynomial $HP(t)$ with rational coefficients such that $HP(d) = \dim_K R_d$ for all but finitely many positive integers $d$.

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([x^3*y^2 + 3*x^2*y^2*z + y^3*z^2 + z^5])
sage: I.hilbert_polynomial()
sage: 5*t - 5
```

Of course, the Hilbert polynomial of a zero-dimensional ideal is zero:

```python
sage: J0 = Ideal([x^3*y^2 + 3*x^2*y^2*z + y^3*z^2 + z^5, y^3-2*x*z^2+x*y,x^4+x*y-y*z^2])
sage: J = P*[m.lm() for m in J0.groebner_basis()]
sage: J.dimension()
sage: 0
sage: J.hilbert_polynomial()
sage: 0
```

It is possible to request a computation using the Singular library:

```python
sage: I.hilbert_polynomial(algorithm = 'singular') == I.hilbert_polynomial()
sage: True
sage: J.hilbert_polynomial(algorithm = 'singular') == J.hilbert_polynomial()
sage: True
```

Here is a bigger examples:

```python
sage: n = 4; m = 11; P = PolynomialRing(QQ, n * m, "x"); x = P.gens(); M = Matrix(n, x)
sage: Minors = P.ideal(M.minors(2))
sage: hp = Minors.hilbert_polynomial(); hp
1/21772800*t^13 + 61/21772800*t^12 + 1661/21772800*t^11 + 26681/21772800*t^10 + 93841/7257600*t^9 + 685421/7257600*t^8 + 1524809/3110400*t^7 + 39780323/21772800*t^6 + 6638071/1360800*t^5 + 12509761/1360800*t^4 + 2689031/226800*t^3 + 1494509/151200*t^2 + 12001/2520*t + 1
```

Because Singular uses 32-bit integers, the above example would fail with Singular. We don’t test it here, as it has a side-effect on other tests that is not understood yet (see trac ticket #26300):

```python
sage: Minors.hilbert_polynomial(algorithm = 'singular') # not tested
Traceback (most recent call last):
  ... RuntimeError: error in Singular function call 'hilbPoly':
```
int overflow in hilb 1
error occurred in or before poly.lib::hilbPoly line 58: ` intvec v=hilb(I,
˓→2);`
expected intvec-expression. type 'help intvec;'

Note that in this example, the Hilbert polynomial gives the coefficients of the Hilbert-Poincaré series in all
degrees:

```
sage: P = PowerSeriesRing(QQ, 't', default_prec = 50)
sage: hs = Minors.hilbert_series()
sage: list(P(hs.numerator()) / P(hs.denominator())) == [hp(t = k) for k in
˓→range(50)]
True
```

**hilbert_series** *(grading=none, algorithm='sage')*

Return the Hilbert series of this ideal.

**INPUT:**

- grading – (optional) a list or tuple of integers
- algorithm – (default: 'sage') must be either 'sage' or 'singular'

Let $I$ (which is self) be a homogeneous ideal and $R = \bigoplus_d R_d$ (which is self.ring()) be a graded
cummutative algebra over a field $K$. Then the *Hilbert function* is defined as $H(d) = \dim_K R_d$ and the
Hilbert *series* of $I$ is defined as the formal power series $HS(t) = \sum_{d=0}^{\infty} H(d)t^d$.

This power series can be expressed as $HS(t) = Q(t)/(1 - t)^n$ where $Q(t)$ is a polynomial over $Z$ and $n$
the number of variables in $R$. This method returns $Q(t)/(1 - t)^n$, normalised so that the leading monomial
of the numerator is positive.

An optional grading can be given, in which case the graded (or weighted) Hilbert series is given.

**EXAMPLES:**

```
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([x^3*y^2 + 3*x^2*y^2*z + y^3*z^2 + z^5])
sage: I.hilbert_series()
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: K = R.ideal([a^2*b^3, a*b^4 + a^3*b^2])
sage: K.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: K.hilbert_series(grading=[2,1])
(2*t^7 - t^6 - t^4 - t^2 - 1)/(t - 1)
```

**integral_closure** *(p=0, r=True, singular='singular_default')*

Let $I = \text{self}$.

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

3.1. Multivariate Polynomials and Polynomial Rings 333
the closure of $R[It]$ (so this is the last power whose closure is not just the sum/product of the smaller). If $r$ is given and $r$ is True, I.integral_closure() starts with a check whether I is already a radical ideal.

INPUT:

- $p$ - powers of I (default: 0)
- $r$ - check whether self is a radical ideal first (default: True)

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: I = ideal([x^2,x*y^4,y^5])
sage: I.integral_closure()
[x^2, x*y^4, y^5, x*y^3]
```

ALGORITHM:

Uses libSINGULAR.

`interreduced_basis()`

If this ideal is spanned by $(f_1,..., f_n)$ this method returns $(g_1,..., g_s)$ such that:

- $(f_1,..., f_n) = (g_1,..., g_s)$
- $LT(g_i) = LT(g_j)$ for all $i! = j$
- $LT(g_i)$ does not divide $m$ for all monomials $m$ of $\{g_1,..., g_i-1, g_i+1,..., g_s\}$
- $LC(g_i)$ == 1 for all $i$ if the coefficient ring is a field.

EXAMPLES:

```
sage: R.<x,y,z> = polynomialRing(QQ)
sage: 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 \texttt{sage.rings.polynomial.toy_buchberger.inter_reduction()} if conversion to Singular fails.

\texttt{intersection(*others)}

Return the intersection of the arguments with this ideal.

**EXAMPLES:**

\begin{verbatim}
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
\end{verbatim}

The following simple example illustrates that the product need not equal the intersection.

\begin{verbatim}
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: I*J == K
False
\end{verbatim}

Intersection of several ideals:

\begin{verbatim}
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 in x, y, z over Rational Field
\end{verbatim}

The ideals must share the same ring:

\begin{verbatim}
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.
\end{verbatim}

\texttt{is_prime(**kwds)}

Return True if this ideal is prime.

**INPUT:**

- keyword arguments are passed on to \texttt{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:

```python
sage: Q = R.quotient(I); Q
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the
→ ideal (x^2 - y^2 - 1)
sage: Q.fraction_field()
Fraction Field of Quotient of Multivariate Polynomial Ring in x, y over
→ Rational Field by the ideal (x^2 - y^2 - 1)
```

minimal_associated_primes()

OUTPUT:

- list - a list of prime ideals

EXAMPLES:

```python
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:

```python
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)
```

(continues on next page)
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 \(n \in \mathbb{Z}\).

If Q is a primary ideal of the ring R, then the radical ideal \(P\) of Q (i.e. the ideal consisting of all \(a \in R\) with \(a^n \in Q\) for some \(n \in \mathbb{Z}\)), is called the associated prime of Q.

If I is a proper ideal of a Noetherian ring R, then there exists a finite collection of primary ideals \(Q_i\) such that the following hold:

• the intersection of the \(Q_i\) is I;
• none of the \(Q_i\) contains the intersection of the others;
• the associated prime ideals of the \( Q_i \) are pairwise distinct.

INPUT:

• \texttt{algorithm} – string:
  
  – 'sy' – (default) use the Shimoyama-Yokoyama algorithm
  
  – 'gtz' – use the Gianni-Trager-Zacharias algorithm

OUTPUT:

• a list of primary ideals \( Q_i \) forming a primary decomposition of \( \text{self} \).

EXAMPLES:

```
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^2 + 1, y + 1) of Multivariate Polynomial Ring in x, y, z over Rational Field,
 Ideal (z^6 + 4*z^3 + 4, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field]
```

```
sage: from functools import reduce
sage: reduce(lambda Q_i, Q_j: Q_i.intersection(Q_j), pd) == I
True
```

ALGORITHM:

Uses Singular.

REFERENCES:

• Thomas Becker and Volker Weispfenning. Groebner Bases - A Computational Approach To Commu-

\texttt{primary\_decomposition\_complete} (\texttt{object})

A decorator that creates a cached version of an instance method of a class.

\textbf{Note:} For proper behavior, the method must be a pure function (no side effects). Arguments to the

method must be hashable or transformed into something hashable using \texttt{key} or they must define \texttt{sage}.

\texttt{structure.sage\_object.SageObject._cache\_key()}.

EXAMPLES:

```
sage: class Foo(object):
    ....:   @cached_method
    ....:   def f(self, t, x=2):
    ....:     print('computing')
    ....:     return t**x
sage: a = Foo()
```

The example shows that the actual computation takes place only once, and that the result is identical for

equivalent input:

```
sage: res = a.f(3, 2); res
computing
9
```
Note, however, that the CachedMethod is replaced by a CachedMethodCaller or CachedMethodCallerNoArgs as soon as it is bound to an instance or class:

```python
sage: P.<a,b,c,d> = QQ[]
sage: I = P*[a,b]
sage: type(I.__class__.gens)
<type 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
```

So, you would hardly ever see an instance of this class alive.

The parameter `key` can be used to pass a function which creates a custom cache key for inputs. In the following example, this parameter is used to ignore the `algorithm` keyword for caching:

```python
sage: class A(object):
    ....: def _f_normalize(self, x, algorithm):
    ....:     return x
    ....: @cached_method(key=_f_normalize)
    ....: def f(self, x, algorithm='default'):
    ....:     return x
sage: a = A()
sage: a.f(1, algorithm="default") is a.f(1) is a.f(1, algorithm="algorithm")
True
```

The parameter `do_pickle` can be used to enable pickling of the cache. Usually the cache is not stored when pickling:

```python
sage: class A(object):
    ....: @cached_method
    ....: def f(self, x):
    ....:     return None
sage: import __main__
sage: __main__.A = A
sage: a = A()
sage: a.f(1)
<...>
sage: len(a.f.cache)
1
sage: b = loads(dumps(a))
sage: len(b.f.cache)
0
```

When `do_pickle` is set, the pickle contains the contents of the cache:

```python
sage: class A(object):
    ....: @cached_method(do_pickle=True)
    ....: def f(self, x):
    ....:     return None
sage: __main__.A = A
sage: a = A()
sage: a.f(1)
<...>
sage: len(a.f.cache)
1
sage: b = loads(dumps(a))
sage: len(b.f.cache)
1
```

Cached methods can not be copied like usual methods, see trac ticket #12603. Copying them can lead to
very surprising results:

```
sage: class A:
....:     @cached_method
....:     def f(self):
....:         return 1
sage: class B:
....:     g=A.f
....:     def f(self):
....:         return 2
sage: b=B()
sage: b.f() 2
sage: b.g() 1
sage: b.f() 1
```

**quotient** *(J)*

Given ideals \( I = \text{self} \) and \( J \) in the same polynomial ring \( P \), return the ideal quotient of \( I \) by \( J \) consisting of the polynomials \( a \) of \( P \) such that \( \{ aJ \subset I \} \).

This is also referred to as the colon ideal \((I:J)\).

**INPUT:**

- \( J \) - multivariate polynomial ideal

**EXAMPLES:**

```
sage: R.<x,y,z> = PolynomialRing(GF(181),3)
sage: I = Ideal([x^2+x*y*z,y^2-z^3*y,z^3+y^5*x*z])
sage: J = Ideal([x])
sage: Q = I.quotient(J)
sage: y*z + x in I False
sage: x in J True
sage: x * (y*z + x) in I True
```

**radical()**

The radical of this ideal.

**EXAMPLES:**

This is an obviously not radical ideal:

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: I = (x^2, y^3, (x+z)^4 + y^3 + 10*x^2)*R
sage: I.radical()
Ideal (y, x) of Multivariate Polynomial Ring in x, y, z over Rational Field
```

That the radical is correct is clear from the Groebner basis.

```
sage: I.groebner_basis()
[y^3, x^2]
```

This is the example from the Singular manual:
sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y-z^2)*R
sage: I.radical()
Ideal (z^2 - y, y^2*z + y*z + 2*y + 2) of Multivariate Polynomial Ring in x, y, z over Rational Field

Note: From the Singular manual: A combination of the algorithms of Krick/Logar and Kemper is used. Works also in positive characteristic (Kemper’s algorithm).

sage: R.<x,y,z> = PolynomialRing(GF(37), 3)
sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y - z^2)*R
sage: I.radical()
Ideal (z^2 - y, y^2*z + y*z + 2*y + 2) of Multivariate Polynomial Ring in x, y, z over Finite Field of size 37

**saturation** (other)

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.

**EXAMPLES:**

sage: R.<x,y> = PolynomialRing(QQ)
sage: f = 2*x^2 + y
sage: g = y
sage: h = 2*f + g
sage: I = Ideal([f,g,h])
sage: M = I.syzygy_module(); M
[ -2 -1 1]
[ -y 2*x^2 + y 0]
sage: G = vector(I.gens())
sage: M*G
(0, 0)

**ALGORITHM:**

Uses Singular’s syz command.
**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
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
```

**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 - decom. 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.

EXAMPLES:

```python
sage: P.<e,d,c,b,a> = PolynomialRing(QQ,5,order='lex')
sage: I = sage.rings.ideal.Cyclic(P)
sage: GB = Ideal(I.groebner_basis('libsingular:stdfglm'))
sage: GB.triangular_decomposition('singular:triangLfak')
[Ideal (a - 1, b - 1, c - 1, d^2 + 3*d + 1, e + d + 3) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
 Ideal (a - 1, b - 1, c^2 + 3*c + 1, d + c + 3, e - 1) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
 Ideal (a - 1, b^2 + 3*b + 1, c + b + 3, d - 1, e - 1) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
 Ideal (a - 1, b^3 + b^2 + b + 1, -c + b^2, -d + b^3, e + b^3 + b^2 + b^1) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
 Ideal (a^2 + 3*a + 1, b - 1, c - 1, d - 1, e + a + 3) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
 Ideal (a^2 + 3*a + 1, b + a + 3, c - 1, d - 1, e - 1) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
 Ideal (a^4 - 4*a^3 + 6*a^2 + a + 1, -11*b^2 + 6*b*a^3 - 26*b*a^2 + 41*b*a - 41*b - 26*a^2 + 41*a - 4) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
 Ideal (a^4 + a^3 + a^2 + a + 1, b - 1, c + a^3 + a^2 + a + 1, -d + a^3, -e + a^2) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
 Ideal (a^4 + a^3 + a^2 + a + 1, b - a, c - a, d^2 + 3*d*a + a^2, e + d + 3*a) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
 Ideal (a^4 + a^3 + a^2 + a + 1, b - a, c^2 + 3*c*a + a^2, d + c + 3*a, e - a) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
 Ideal (a^4 + a^3 + a^2 + a + 1, b^2 + 3*b*a + a^2, c + b + 3*a, d - a, e - a) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
 Ideal (a^4 + a^3 + a^2 + a + 1, b^3 + b^2*a + b^2 + b*a + b + a^3 + a^1 + 2 + a + 1, c + b^2*a^3 + b^2*a^2 + b^2*a + b^2, -d + b^2*a^2 + b^2*a + b^2) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
 Ideal (a^4 + a^3 + 6*a^2 - 4*a + 1, -11*b^2 + 6*b*a^3 + 10*b*a^2 + 39*b*a + 2 - 2*b + 16*a^3 + 23*a^2 + 104*a - 24, 11*c + 3*a^3 + 5*a^2 + 25*a + 1, 11*d + 3*a^3 + 5*a^2 + 25*a + 1, -11*e + 11*b + 6*a^3 + 10*a^2 + 39*a + 2) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field]
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]
```

(continues on next page)
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 \( \mathbb{Q} \) (to allow fast symbolic computation of the triangular decomposition) and \( ring \) is \( \mathbb{R}, \mathbb{A}, \mathbb{C}, \) or \( \mathbb{Q}\bar{Q} \) (to compute the whole real or complex variety of the ideal).

Note that with \( ring=\mathbb{R} \) or \( \mathbb{C} \), computation is done numerically and potentially inaccurately; in particular, the number of points in the real variety may be miscomputed. With \( ring=\mathbb{A} \) or \( \mathbb{Q}\bar{Q} \), computation is done exactly (which may be much slower, of course).

INPUT:
- \( \text{ring} \) - return roots in the \( \text{ring} \) instead of the base ring of this ideal (default: None)
- \( \text{proof} \) - return a provably correct result (default: True)

EXAMPLES:

```
sage: K.<w> = GF(27) # this example is from the MAGMA handbook
sage: P.<x, y> = PolynomialRing(K, 2, order='lex')
sage: I = Ideal([ x^8 + y + 2, y^6 + x*y^5 + x^2 ])
sage: I = Ideal(I.groebner_basis()); I
Ideal (x - y^47 - y^45 + y^44 - y^43 + y^41 - y^39 - y^38
- y^37 - y^36 + y^35 + y^34 - y^33 + y^32 - y^31 + y^30 +
y^28 + y^27 + y^26 + y^25 - y^23 + y^22 + y^21 - y^19 -
y^18 - y^16 + y^15 + y^13 + y^12 - y^10 + y^9 + y^8 + y^7
- y^6 + y^4 + y^3 + y^2 + y - 1, y^48 + y^41 - y^40 + y^37
- y^36 - y^33 + y^32 - y^29 + y^28 - y^25 + y^24 + y^2 + y
+ 1) of Multivariate Polynomial Ring in x, y over Finite
Field in w of size 3^3
```

```
sage: V = I.variety();
sage: sorted(V, key=str)
[[y: w^2 + 2*w, x: 2*w + 2], [y: w^2 + 2, x: 2*w], [y: w^2 + w, x: 2*w + 1]]
sage: [f.subs(v) for f in I.gens() for v in V] # check that all polynomials vanish
[0, 0, 0, 0, 0, 0]
sage: [I.subs(v).is_zero() for v in V] # same test, but nicer syntax
[True, True, True]
```

However, we only account for solutions in the ground field and not in the algebraic closure:
Here we compute the points of intersection of a hyperbola and a circle, in several fields:

```python
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:

```python
sage: I.variety()
[{y: 1, x: 1}]
```

There are two real intersections:

```python
sage: sorted(I.variety(ring=RR), key=str)
[{y: 0.361103080528647, x: 2.76929235423863},
 {y: 1.00000000000000, x: 1.00000000000000}]
sage: I.variety(ring=AA) # py2
[{y: 1, x: 1},
 {x: 2.769292354238632?, y: 0.3611030805286474?}]
sage: I.variety(ring=AA) # py3
[{y: 1, x: 1},
 {y: 0.3611030805286474?, x: 2.769292354238632?}]
```

and a total of four intersections:

```python
sage: sorted(I.variety(ring=CC), key=str)
[{y: 0.3194484597356763? + 1.633170240915238?*I,
  x: 0.11535382288068... - 0.5897428052220...*I},
 {y: 0.3194484597356763? - 1.633170240915238?*I,
  x: 0.11535382288068... + 0.5897428052220...*I},
 {y: 0.3611030805286474?, x: 2.769292354238632?},
 {y: 1.00000000000000, x: 1.00000000000000}]
sage: sorted(I.variety(ring=QQbar), key=str)
[{y: 0.3194484597356763? + 1.633170240915238?*I,
  x: 0.11535382288068... - 0.5897428052220...*I},
 {y: 0.3194484597356763? - 1.633170240915238?*I,
  x: 0.11535382288068... + 0.5897428052220...*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:

```python
sage: R.<x,y> = CC[]
sage: I = ideal([x^2+y^2-1, x*y-1])
sage: sorted(I.variety(), key=str)
verbose 0 (...: multi_polynomial_ideal.py, variety) Warning: computations in
  the complex field are inexact; variety may be computed partially or
  incorrectly.
verbose 0 (...: multi_polynomial_ideal.py, variety) Warning: falling back to
  very slow toy implementation.
[{y: -0.86602540378443... + 0.500000000000000*I},
 (continues on next page)

3.1. Multivariate Polynomials and Polynomial Rings 345
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')
I=ideal([x^3-2*y^2,3*x+y^4])
I.variety()
```

```
verbose 0 (...: multi_polynomial_ideal.py, groebner_basis) Warning: falling back to very slow toy implementation.
verbose 0 (...: multi_polynomial_ideal.py, dimension) Warning: falling back to very slow toy implementation.
verbose 0 (...: multi_polynomial_ideal.py, variety) Warning: falling back to very slow toy implementation.
```

```
[{y: 0, x: 0}]
```

The dictionary expressing the variety will be indexed by generators of the polynomial ring after changing to the target field. But the mapping will also accept generators of the original ring, or even generator names as strings, when provided as keys:

```sage
K.<x,y> = QQ[]
I = ideal([x^2+2*y-5,x+y+3])
v = I.variety(AA)[0]; v[x], v[y]
```

```
(4.464101615137755?, -7.464101615137755?)
```

```
list(v)[0].parent()
```

```
Multivariate Polynomial Ring in x, y over Algebraic Real Field
```

```
v[x]
```

```
4.464101615137755?
```

```
v["y"]
```

```
-7.464101615137755?
```

**ALGORITHM:**

Uses triangular decomposition.

**vector_space_dimension**

Return the vector space dimension of the ring modulo this ideal. If the ideal is not zero-dimensional, a TypeError is raised.

**ALGORITHM:**

Uses Singular.

**EXAMPLES:**

```sage
R.<u,v> = PolynomialRing(QQ)
g = u^4 + v^4 + u^3 + v^3
I = ideal(g) + ideal(g.gradient())
I.dimension()
```

```
0
```

```
I.vector_space_dimension()
```

```
4
```

When the ideal is not zero-dimensional, we return 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:

```python
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: H.inject_variables()
Defining x, y, z
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False)  # indirect doctest
sage: I
Left Ideal (y^2, x^2, z^2 - 1) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z}
sage: sorted(I.gens(),key=str)
[x^2, y^2, z^2 - 1]
sage: H.ideal([y^2, x^2, z^2-H.one()], side="twosided")  # random
Twosided Ideal (y^2, x^2, z^2 - 1) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z}
sage: sorted(H.ideal([y^2, x^2, z^2-H.one()], side="twosided").gens(),key=str)
[x^2, y^2, z^2 - 1]
sage: H.ideal([y^2, x^2, z^2-H.one()], side="right")
Traceback (most recent call last):
... ValueError: Only left and two-sided ideals are allowed.
```

`elimination_ideal(variables)`

Return the elimination ideal of this ideal with respect to the variables given in “variables”.

EXAMPLES:

```python
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: H.inject_variables()
Defining x, y, z
```
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False)
sage: I.elimination_ideal([x, z])
Left Ideal (y^2) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {...}
sage: J = I.twostd()
sage: J
Twosided Ideal (z^2 - 1, y*z - y, x*z + x, y^2, 2*x*y - z - 1, x^2) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {...}
sage: J.elimination_ideal([x, z])
Twosided Ideal (y^2) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {...}

ALGORITHM: Uses Singular’s eliminate command

reduce(p)
Reduce an element modulo a Groebner basis for this ideal.
It returns 0 if and only if the element is in this ideal. In any case, this reduction is unique up to monomial orders.

NOTE:
There are left and two-sided ideals. Hence,

EXAMPLES:

sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False, side='twosided')
sage: Q = H.quotient(I); Q #random
Quotient of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z} by the ideal (y^2, x^2, z^2 - 1)
sage: Q.2^2 == Q.one() # indirect doctest
True

Here, we see that the relation that we just found in the quotient is actually a consequence of the given relations:

sage: H.2^2-H.one() in I.std().gens()
True

Here is the corresponding direct test:

sage: I.reduce(z^2)
1

res(length)
Compute the resolution up to a given length of the ideal.

NOTE:
Only left syzygies can be computed. So, even if the ideal is two-sided, then the resolution is only one-sided. In that case, a warning is printed.

EXAMPLES:
std()

Computes a GB of the ideal. It is two-sided if and only if the ideal is two-sided.

EXAMPLES:

```python
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
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>
```

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:

```python
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 over Rational Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z}
sage: sorted(JL.gens(),key=str)
[x^3, y^3, z^3 - 4*z]
sage: JL.std() #random
Left Ideal (z^3 - 4*z, y*z^2 - 2*y*z, x*z^2 + 2*x*z, y^2*z - 2*y^2, x*y*z - z^2 - 2*z, x^3) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z}
sage: sorted(JL.std().gens(),key=str)
[2*x*y*z - z^2 - 2*z, x*y^2 - y*z, x*z^2 + 2*x*z, x^2*y - x*z - 2*x, x^3, y*z^2 - 2*y*z, y^2*z - 2*y^2, y^3, z^3 - 4*z]
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, z over Rational Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z}
sage: sorted(JT.gens(),key=str)
[x^3, y^3, z^3 - 4*z]
sage: JT.std() #random
Twosided Ideal (z^3 - 4*z, y*z^2 - 2*y*z, x*z^2 + 2*x*z, y^2*z - 2*y^2, x*y*z - z^2 - 2*z, x^3) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z}
sage: sorted(JT.std().gens(),key=str)
[2*x*y*z - z^2 - 2*z, x*y^2 - y*z, x*z^2 + 2*x*z, x^2*y - x*z - 2*x, x^2*z + t, x^3, y*z^2 - 2*y*z, y^2*z - 2*y^2, y^3, z^3 - 4*z]
sage: JT.std() == JL.twostd()
True
```
ALGORITHM: Uses Singular’s std command

syzygy_module()  
Computes the first syzygy (i.e., the module of relations of the given generators) of the ideal.

NOTE:
Only left syzygies can be computed. So, even if the ideal is two-sided, then the syzygies are only one-sided.
In that case, a warning is printed.

EXAMPLES:

```sage
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 y^2]
[ -2 + 8*z - 15 0 x^2]
[ 2*z + 15*x^2 -y^2*z^2 + 8*y^2*z - 15*y^2 -4*x*y*z + 2*z^2 + 2*z]
[ -282*x*z - 360*x -y^3*z^2 + 6*y^3*z - 12*y^3 -z^2 + 8*z - 15 0]
[ 2*x*y*z - 224*y*z + 320*y -6*x*z^2]
[ 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 -y^4*z + 4*y^4 +y^2*z^2 + 8*x^2*z^2 + 8*x^2]
[ 2*x^3*y*z + 8*x^3*y + 9*x^2*z + 2*x^2*y^2 + 99*y^2*z - 195*y^2 -36*x*y*z + 24*z^2 + 18*z]
[ 64*x*y + 66*z^2 - 240*z + 288 0]
```

(continues on next page)
\[x^3y^2z + 4x^3y^2 + 18x^2yz - 36xz^3 + 66x^2y - 432xz^2 - \cdots\]
\[\rightarrow 1656x^2z - 2052x\]
\[\rightarrow 8y^3z^2 + 62y^3z - 114y^3\]
\[\rightarrow 48yz^2 - 36yz\]

\texttt{sage: M*G}
\begin{verbatim}
(0, 0, 0, 0, 0, 0, 0, 0, 0)
\end{verbatim}

ALGORITHM: Uses Singular’s \texttt{syz} command

twostd()

Computes a two-sided GB of the ideal (even if it is a left ideal).

EXAMPLES:

\begin{verbatim}
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: H.inject_variables()
Defining x, y, z
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False)
sage: I.twostd() #random
Twosided Ideal (z^2 - 1, y*z - y, x*z + x, y^2, 2*x*y - z - 1, x^2) of
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field..
sage: sorted(I.twostd().gens(),key=str)
[2*x*y - z - 1, x*z + x, x^2, y*z - y, y^2, z^2 - 1]
\end{verbatim}

ALGORITHM: Uses Singular’s \texttt{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.

\textbf{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:

\begin{verbatim}
sage: from sage.rings.polynomial.multi_polynomial_ideal import is_MPolynomialIdeal
c sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = [x + 2*y^2 + 2*z^2 - 1, x^2 + 2*y^2 + 2*z^2 - x, 2*x*y + 2*y*z - y]
sage: is_MPolynomialIdeal(I)
False
\end{verbatim}

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.

3.1. Multivariate Polynomials and Polynomial Rings
We call a finite list of polynomials a Polynomial Sequence.

Polynomial sequences in Sage can optionally be viewed as consisting of various parts or sub-sequences. These kind of polynomial sequences which naturally split into parts arise naturally for example in algebraic cryptanalysis of symmetric cryptographic primitives. The most prominent examples of these systems are: the small scale variants of the AES [CMR2005] (cf. sage.crypto.mq.sr.SR()) and Flurry/Curry [BPW2006]. By default, a polynomial sequence has exactly one part.

AUTHORS:
- Martin Albrecht (2007ff): initial version
- Martin Albrecht (2009): refactoring, clean-up, new functions
- Martin Albrecht (2011): refactoring, moved to sage.rings.polynomial
- Alex Raichev (2011-06): added algebraic_dependence()
- Charles Bouillaguet (2013-1): added solve()

EXAMPLES:
As an example consider a small scale variant of the AES:

```python
sage: sr = mq.SR(2,1,2,4,gf2=True,polybori=True)
sage: sr
SR(2,1,2,4)
```

We can construct a polynomial sequence for a random plaintext-ciphertext pair and study it:

```python
sage: set_random_seed(1)
sage: F,s = sr.polynomial_system()
sage: F
Polynomial Sequence with 112 Polynomials in 64 Variables
```

```python
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*
```

(continues on next page)
We separate the system in independent subsystems:

\[
\begin{align*}
&w_{213} + k_{113} + x_{111} + x_{112} + x_{113}, \\
&w_{212} + k_{112} + x_{110} + x_{111} + x_{112} + 1, \\
&w_{211} + k_{111} + x_{110} + x_{111} + x_{113} + 1, \\
&w_{210} + k_{110} + x_{110} + x_{112} + x_{113}, \\
&w_{110} + x_{111} + x_{111} + x_{112} + x_{113} + 1, \\
&w_{110} + x_{110} + x_{111} + x_{112} + x_{113} + 1, \\
&w_{110} + x_{110} + x_{111} + x_{112} + x_{113} + 1, \\
&w_{110} + x_{110} + x_{111} + x_{112} + x_{113} + 1, \\
&w_{110} + x_{110} + x_{111} + x_{112} + x_{113} + 1, \\
&w_{110} + x_{110} + x_{111} + x_{112} + x_{113} + 1.
\end{align*}
\]

We separate the system in independent subsystems:

\[
\begin{align*}
&w_{203} + k_{103} + x_{101} + x_{102} + x_{103}, \\
&w_{202} + k_{102} + x_{100} + x_{101} + x_{102} + 1,
\end{align*}
\]

(continues on previous page)
w201 + k101 + x110 + x101 + x103 + 1,
w200 + k100 + x100 + x102 + x101,
x100*w102 + x101*w101 + x102*w100 + x103*w103 + 1,
x100*w102 + x101*w100 + x101*w101 + x102*w101 + x102*w100 + x103*w102 + w101,
x100*w101 + x101*w100 + x101*w102 + x102*w100 + x103*w103 + w103,
x100*w101 + x101*w100 + x101*w101 + x102*w100 + x102*w103 + x103*w101 + x103*w102 + x101,
x100*w101 + x101*w100 + x101*w102 + x102*w100 + x102*w103 + x103*w101 + x103*w102 + x101,
x100*w101 + x101*w100 + x101*w102 + x102*w100 + x102*w103 + x103*w101 + x103*w102 + x101,
x100*w101 + x101*w100 + x101*w102 + x102*w100 + x102*w103 + x103*w101 + x103*w102 + x101,
x100*w100 + x100*w103 + x101*w102 + x102*w101 + x103*w100,
x100*w100 + x100*w102 + x100*w103 + x101*w100 + x101*w101 + x102*w102 + x102*w103 + x103*w102 + x100,
x100*w100 + x100*w101 + x100*w102 + x100*w103 + x101*w100 + x101*w102 + x102*w100 + x102*w102 + x103*w100 + x101,
x100*w100 + x100*w101 + x100*w102 + x100*w103 + x101*w101 + x101*w102 + x102*w100 + x102*w102 + x103*w100 + x103*w102 + x100,
x100*w100 + x100*w101 + x100*w102 + x100*w103 + x100*w101 + x101*w101 + x101*w102 + x102*w100 + x102*w102 + x103*w100 + x103*w102 + x100,
x100*w100 + x100*w101 + x100*w102 + x100*w103 + x101*w101 + x101*w102 + x102*w100 + x102*w102 + x103*w100 + x103*w102 + x100,
x100*w100 + x100*w101 + x100*w102 + x100*w103 + x100*w101 + x101*w101 + x101*w102 + x102*w100 + x102*w102 + x103*w100 + x103*w102 + x100,
x100*w100 + x100*w101 + x100*w102 + x100*w103 + x101*w101 + x101*w102 + x102*w100 + x102*w102 + x103*w100 + x103*w102 + x100,
x100*w100 + x100*w101 + x100*w102 + x100*w103 + x101*w101 + x101*w102 + x102*w100 + x102*w102 + x103*w100 + x103*w102 + x100,
x100*w100 + x100*w101 + x100*w102 + x100*w103 + x101*w101 + x101*w102 + x102*w100 + x102*w102 + x103*w100 + x103*w102 + x100,
x100*w100 + x100*w101 + x100*w102 + x100*w103 + x100*w101 + x101*w101 + x101*w102 + x102*w100 + x102*w102 + x103*w100 + x103*w102 + x100,

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)
```

Note: In many other computer algebra systems (cf. Singular) this class would be called Ideal but an ideal is a very distinct object from its generators and thus this is not an ideal in Sage.
Classes

`sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence(arg1, arg2=None, immutable=False, cr=False, cr_str=None)`

Construct a new polynomial sequence object.

**INPUT:**

- `arg1` - a multivariate polynomial ring, an ideal or a matrix
- `arg2` - an iterable object of parts or polynomials (default: None)
  - `immutable` - if True the sequence is immutable (default: False)
  - `cr` - print a line break after each element (default: False)
  - `cr_str` - print a line break after each element if `str` is called (default: None)

**EXAMPLES:**

```python
sage: P.<a,b,c,d> = PolynomialRing(GF(127),4)
sage: I = sage.rings.ideal.Katsura(P)
sage: Sequence([I.gens(),I.gens()], I.ring()); F
\[\begin{align*}
\text{[} & a + 2*b + 2*c + 2*d - 1, \\
& a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a, \\
& 2*a*b + 2*b*c + 2*c*d - b, \\
& b^2 + 2*a*c + 2*b*d - c, \\
& a + 2*b + 2*c + 2*d - 1, \\
& a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a, \\
& 2*a*b + 2*b*c + 2*c*d - b, \\
& b^2 + 2*a*c + 2*b*d - c\]\n\end{align*}
```

If a list of tuples is provided, those form the parts:

```python
sage: F = Sequence(([I.gens(),I.gens()], I.ring()); F # indirect doctest
\[\begin{align*}
\text{[} & a + 2*b + 2*c + 2*d - 1, \\
& a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a, \\
& 2*a*b + 2*b*c + 2*c*d - b, \\
& b^2 + 2*a*c + 2*b*d - c\]\n\end{align*}
```

If an ideal is provided, the generators are used:

```python
sage: Sequence(I)
\[\begin{align*}
\text{[} & a + 2*b + 2*c + 2*d - 1, \\
& a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a, \\
& 2*a*b + 2*b*c + 2*c*d - b, \\
& b^2 + 2*a*c + 2*b*d - c\]\n\end{align*}
```

If a list of polynomials is provided, the system has only one part:

```python
sage: F = Sequence(I.gens(), I.ring()); F
\[\begin{align*}
\text{[} & a + 2*b + 2*c + 2*d - 1, \\
& a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a, \\
& 2*a*b + 2*b*c + 2*c*d - b, \\
& b^2 + 2*a*c + 2*b*d - c\]\n\end{align*}
```

We test that the ring is inferred correctly:
Construct a new system of multivariate polynomials.

INPUT:

- `parts` - a list of lists with polynomials
- `ring` - a multivariate polynomial ring
- `immutable` - if True the sequence is immutable (default: False)
- `cr` - print a line break after each element (default: False)
- `cr_str` - print a line break after each element if 'str' is called (default: None)

EXAMPLES:

```python
sage: P.<a,b,c,d> = PolynomialRing(GF(127),4)
sage: I = sage.rings.ideal.Katsura(P)
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 - a, 2*b^2 + 2*a*c + 2*b*d - c]
```

If an ideal is provided, the generators are used:

```python
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 - a, 2*b^2 + 2*a*c + 2*b*d - c]
```

If a list of polynomials is provided, the system has only one part:

```python
sage: Sequence(I.gens(), I.ring())
[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 - a, 2*b^2 + 2*a*c + 2*b*d - c]
```

`algebraic_dependence()`

Returns the ideal of annihilating polynomials for the polynomials in `self`, if those polynomials are algebraically dependent. Otherwise, returns the zero ideal.

OUTPUT:

If the polynomials $f_1, \ldots, f_r$ in `self` are algebraically dependent, then the output is the ideal \( \{ F \in K[T_1, \ldots, T_r] : F(f_1, \ldots, f_r) = 0 \} \) of annihilating polynomials of $f_1, \ldots, f_r$. Here $K$ is the coefficient
ring of polynomial ring of \(f_1, \ldots, f_r\) and \(T_1, \ldots, T_r\) are new indeterminates. If \(f_1, \ldots, f_r\) are algebraically independent, then the output is the zero ideal in \(K[T_1, \ldots, T_r]\).

**EXAMPLES:**

```python
sage: R.<x,y> = PolynomialRing(QQ)
sage: S = Sequence([x, x*y])
sage: I = S.algebraic_dependence(); I
Ideal (0) of Multivariate Polynomial Ring in T0, T1 over Rational Field
```

```python
sage: R.<x,y> = PolynomialRing(QQ)
sage: S = Sequence([x, (x^2 + y^2 - 1)^2, x*y - 2])
sage: I = S.algebraic_dependence(); I
Ideal (16 + 32*T2 - 8*T0^2 + 24*T2^2 - 8*T0^2*T2 + 8*T2^3 + 9*T0^4 - 2*T0^2*T2^2 + T0^4*T1 + 8*T0^4*T2 - 2*T0^6 + 2*T0^4*T2^2 + T0^8) of
Multivariate Polynomial Ring in T0, T1, T2 over Rational Field
```

```python
sage: [F(S) for F in I.gens()]
[0]
```

```python
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 + T0^4*T2 - 2*T0^6 + 2*T0^4*T2^2 + T0^8) of Multivariate Polynomial Ring in T0, T1, T2 over Finite Field of size 7
```

```python
sage: [F(S) for F in I.gens()]
[0]
```

**Note:** This function’s code also works for sequences of polynomials from a univariate polynomial ring, but i don’t know where in the Sage codebase to put it to use it to that effect.

**AUTHORS:***

- Alex Raichev (2011-06-22)

**coefficient_matrix** *(sparse=True)*

Return tuple \((A, v)\) where \(A\) is the coefficient matrix of this system and \(v\) the matching monomial vector.

Thus value of \(A[i, j]\) corresponds the coefficient of the monomial \(v[j]\) in the \(i\)-th polynomial in this system.

Monomials are order w.r.t. the term ordering of \(self\.\ring()\) in reverse order, i.e. such that the smallest entry comes last.

**INPUT:**

- **sparse** - construct a sparse matrix (default: True)

**EXAMPLES:**

```python
sage: P.<a,b,c,d> = PolynomialRing(GF(127),4)
sage: I = sage.rings.ideal.Katsura(P)
sage: I.gens()
[a + 2*b + 2*c + 2*d - 1,
a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a,
2*a*b + 2*b*c + 2*c*d - b,
b^2 + 2*a*c + 2*b*d - c]
```

(continues on next page)
sage: F = Sequence(I)
sage: A, v = F.coefficient_matrix()
sage: A

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 2 & 2 & 126 \\
1 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 2 & 126 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 126 & 0 & 0 \\
0 & 0 & 1 & 2 & 0 & 0 & 2 & 0 & 0 & 0 & 126 & 0 & 0
\end{bmatrix}
\]

sage: v

\[
\begin{bmatrix}
a^2 \\
a*b \\
b^2 \\
a*c \\
b*c \\
c^2 \\
b*d \\
c*d \\
d^2 \\
a \\
b \\
c \\
d \\
1
\end{bmatrix}
\]

sage: A*v

\[
\begin{bmatrix}
a + 2*b + 2*c + 2*d - 1 \\
(a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a) \\
2*a*b + 2*b*c + 2*c*d - b \\
b^2 + 2*a*c + 2*b*d - c
\end{bmatrix}
\]

connected_components()

Split the polynomial system in systems which do not share any variables.

EXAMPLES:

As an example consider one part of AES, which naturally splits into four subsystems which are independent:

sage: 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.

EXAMPLES:

sage: B.<x,y,z> = BooleanPolynomialRing()
sage: F = Sequence([x*y + y + 1, z + 1])
sage: F = F.connection_graph()
Graph on 3 vertices
**groebner_basis** (*args, **kwargs*)

Compute and return a Groebner basis for the ideal spanned by the polynomials in this system.

**INPUT:**
- *args* - list of arguments passed to `MPolynomialIdeal.groebner_basis` call
- **kwargs** - dictionary of arguments passed to `MPolynomialIdeal.groebner_basis` call

**EXAMPLES:**
```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F, s = sr.polynomial_system()
sage: gb = F.groebner_basis()
sage: Ideal(gb).basis_is_groebner()
True
```

**ideal**()

Return ideal spanned by the elements of this system.

**EXAMPLES:**
```
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 over Finite Field in a of size 2^4
```

**is_groebner** (*singular=Singular*)

Returns `True` if the generators of this ideal (`self.gens()`) form a Groebner basis.

Let $I$ be the set of generators of this ideal. The check is performed by trying to lift $Syz(LM(I))$ to $Syz(I)$ as $I$ forms a Groebner basis if and only if for every element $S$ in $Syz(LM(I))$:

$$ S * G = \sum_{i=0}^{m} h_i g_i > G 0. $$

**EXAMPLES:**
```
sage: R.<a,b,c,d,e,f,g,h,i,j> = PolynomialRing(GF(127),10)
sage: I = sage.rings.ideal.Cyclic(R,4)
sage: I.basis.is_groebner()
False
sage: I2 = Ideal(I.groebner_basis())
sage: I2.basis.is_groebner()
True
```

**maximal_degree**()

Return the maximal degree of any polynomial in this sequence.

**EXAMPLES:**
monomials()  
Return an unordered tuple of monomials in this polynomial system.

EXAMPLES:

```python
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F, s = sr.polynomial_system()
sage: len(F.monomials())
49
```

nmonomials()  
Return the number of monomials present in this system.

EXAMPLES:

```python
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F, s = sr.polynomial_system()
sage: F.nmonomials()
49
```

nparts()  
Return number of parts of this system.

EXAMPLES:

```python
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F, s = sr.polynomial_system()
sage: F.nparts()
4
```

nvariables()  
Return number of variables present in this system.

EXAMPLES:

```python
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.

EXAMPLES:

```python
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F, s = sr.polynomial_system()
sage: R0 = F.part(1)
sage: R0
(k000^2 + k001, k001^2 + k002, k002^2 + k003, k003^2 + k000)
```
parts()

Return a tuple of parts of this system.

EXAMPLES:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F, s = sr.polynomial_system()
sage: l = F.parts()
sage: len(l)
4
```

reduced()

If this sequence is \((f_1, \ldots, f_n)\) then this method returns \((g_1, \ldots, g_s)\) such that:

- \((f_1, \ldots, f_n) = (g_1, \ldots, g_s)\)
- \(LT(g_i) = LT(g_j)\) for all \(i \neq j\)
- \(LT(g_i)\) does not divide \(m\) for all monomials \(m\) of \([g_i-1, g_i+1, \ldots, g_s]\)
- \(LC(g_i) = 1\) for all \(i\) if the coefficient ring is a field.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: F = Sequence([z*x+y^3,z+y^3,z+x*y])
sage: F.reduced()
[y^3 + z, x*y + z, x*z - z]

Note that tail reduction for local orderings is not well-defined:

```
sage: R.<x,y,z> = PolynomialRing(QQ,order='negdegrevlex')
sage: F = Sequence([z*x+y^3,z+y^3,z+x*y])
sage: F.reduced()
[z + x*y, x*y - y^3, x^2*y - y^3]
```

A fixed error with nonstandard base fields:

```
sage: R.<t>=QQ['t']
sage: K.<x,y>=R.fraction_field()['x,y']
sage: I=t*x*K
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 \texttt{sage.rings.polynomial.toy_buchberger.inter_reduction()} if conversion to Singular fails.

\textbf{ring()}

Return the polynomial ring all elements live in.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: sr = mq.SR(allow_zero_inversions=True,gf2=True,order='block')
sage: F,s = sr.polynomial_system()
sage: print(F.ring().repr_long()) Polynomial Ring Base Ring : Finite Field of size 2 Size : 20 Variables Block 0 : Ordering : deglex Names : k100, k101, k102, k103, x100, x101, x102, x103, w100, ...
               w101, w102, w103, s000, s001, s002, s003 Block 1 : Ordering : deglex Names : k000, k001, k002, k003
\end{verbatim}

\textbf{subs(*args, **kwargs)}

Substitute variables for every polynomial in this system and return a new system. See \texttt{MPolynomial.subs} for calling convention.

\textbf{INPUT:}

- \texttt{args} - arguments to be passed to \texttt{MPolynomial.subs}
- \texttt{kwargs} - keyword arguments to be passed to \texttt{MPolynomial.subs}

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

\textbf{universe()}

Return the polynomial ring all elements live in.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: sr = mq.SR(allow_zero_inversions=True,gf2=True,order='block')
sage: F,s = sr.polynomial_system()
sage: print(F.ring().repr_long()) Polynomial Ring
  Base Ring : Finite Field of size 2
  Size : 20 Variables
  Block 0 : Ordering : deglex
      Names : k100, k101, k102, k103, x100, x101, x102, x103, w100, ...
               w101, w102, w103, s000, s001, s002, s003
  Block 1 : Ordering : deglex
      Names : k000, k001, k002, k003
\end{verbatim}

\textbf{variables()}

Return all variables present in this system. This tuple may or may not be equal to the generators of the ring of this system.
EXAMPLES:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: F.variables()[:10]
(k003, k002, k001, k000, s003, s002, s001, s000, w103, w102)
```

```python
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 $F_2$.

**eliminate_linear_variables**

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: `use_polybori=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.

**EXAMPLES:**

```
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)
```

(continues on next page)
The list of reductors can be requested by setting `return_reductors` to True:

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing()
(sage: F = Sequence([a + b + d, a + b + c])
(sage: F,R = F.eliminate_linear_variables(return_reductors=True)
(sage: F
[]
(sage: R
[a + b + d, c + d]
```

If the input system is detected to be inconsistent then [1] is returned and the list of reductors is empty:

```python
sage: R.<x,y,z> = BooleanPolynomialRing()
(sage: S = Sequence([x*y*z+x*y+z*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], [])
```

Note: This is called “massaging” in [BCJ2007].

### reduced()

If this sequence is \((f_1, \ldots, f_n)\) this method returns \((g_1, \ldots, g_s)\) such that:

- \(<f_1, \ldots, f_n> = <g_1, \ldots, g_s>
- LT(g_i)! = LT(g_j)\) for all i! = j
- LT(g_i) does not divide m for all monomials m of g_1, \ldots, g_{i-1}, g_{i+1}, \ldots, g_s

**EXAMPLES:**

```python
sage: sr = mq.SR(1, 1, 1, 4, gf2=True, polybori=True)
(sage: F,s = sr.polynomial_system()
(sage: F.reduced()
[k100 + 1, k101 + k001 + 1, k102, k103 + 1, ..., s002, s003 + k001 + 1, k000
˓→ 1, k002 + 1, k003 + 1]
```

### solve (algorithm='polybori', n=1, eliminate_linear_variables=True, verbose=False, **kwds)

Find solutions of this boolean polynomial system.

This function 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 \( n == +\infty \) then all solutions are returned. If \( n < \infty \) then \( n \) solutions are returned if the equations have at least \( n \) solutions. Otherwise, all the solutions are returned. (default: 1)

• \texttt{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).

• \texttt{verbose} - whether to display progress and (potentially) useful information while the computation runs. (default: False)

EXAMPLES:

Without argument, a single arbitrary solution is returned:

```python
sage: from sage.doctest.fixtures import reproducible_repr
sage: R.<x,y,z> = BooleanPolynomialRing()
sage: S = Sequence([x*y+z, y*z+x, x+y+z+1])
sage: sol = S.solve()
sage: print(reproducible_repr(sol))
[{x: 0, y: 1, z: 0}]
```

We check that it is actually a solution:

```python
sage: S.subs( sol[0] )
[0, 0, 0]
```

We obtain all solutions:

```python
sage: sols = S.solve(n=Infinity)
sage: print(reproducible_repr(sols))
[{x: 0, y: 1, z: 0}, {x: 1, y: 1, z: 1}]
sage: [S.subs(x) for x in sols]
[[0, 0, 0], [0, 0, 0]]
```

We can force the use of exhaustive search if the optional package FES is present:

```python
sage: sol = S.solve(algorithm='exhaustive_search')  # optional - FES
sage: print(reproducible_repr(sol))  # optional - FES
[{x: 1, y: 1, z: 1}]
sage: S.subs( sol[0] )
[0, 0, 0]
```

And we may use SAT-solvers if they are available:

```python
sage: sol = S.solve(algorithm='sat')  # optional - cryptominisat
sage: print(reproducible_repr(sol))  # optional - cryptominisat
[{x: 0, y: 1, z: 0}]
sage: S.subs( sol[0] )
[0, 0, 0]
```

```python
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 $F_2$, i.e extensions over $F(2)$.

**weil_restriction()**

Project this polynomial system to $F_2$.

That is, compute the Weil restriction of scalars for the variety corresponding to this polynomial system and express it as a polynomial system over $F_2$.

**EXAMPLES:**

```python
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:

```python
sage: sr = mq.SR(1,1,1,4,gf2=False)
sage: F,s = sr.polynomial_system(); F
Polynomial Sequence with 40 Polynomials in 20 Variables
sage: F2 = F.weil_restriction(); F2
Polynomial Sequence with 240 Polynomials in 80 Variables
```

```
sage.rings.polynomial.multi_polynomial_sequence.is_PolynomialSequence(F)
Return True if F is a PolynomialSequence.

**INPUT:**

• F - anything

**EXAMPLES:**

```python
sage: P.<x,y> = PolynomialRing(QQ)
sage: I = [[x^2 + y^2], [x^2 - y^2]]
sage: F = Sequence(I, P); F
[x^2 + y^2, x^2 - y^2]
sage: from sage.rings.polynomial.multi_polynomial_sequence import is_PolynomialSequence
sage: is_PolynomialSequence(F)
True
```

### 3.1.8 Multivariate Polynomials via libSINGULAR

This module implements specialized and optimized implementations for multivariate polynomials over many coefficient rings, via a shared library interface to SINGULAR. In particular, the following coefficient rings are supported by this implementation:

• the rational numbers $\mathbb{Q}$,
• the ring of integers $\mathbb{Z}$,
• $\mathbb{Z}/n\mathbb{Z}$ for any integer $n$,
• finite fields $\mathbb{F}_{p^n}$ for $p$ prime and $n > 0$,
• and absolute number fields $\mathbb{Q}(a)$. 

366 Chapter 3. Multivariate Polynomials
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 \( \mathbb{Q}(a) \) and \( \mathbb{Z} \) support
- Simon King (2009-04): improved coercion
- Martin Albrecht (2009-05): added \( \mathbb{Z}/n\mathbb{Z} \) support, refactoring
- Martin Albrecht (2009-06): refactored the code to allow better re-use
- Simon King (2011-03): Use a faster way of conversion from the base ring.

Todo: Implement Real, Complex coefficient rings via libSINGULAR

EXAMPLES:
We show how to construct various multivariate polynomial rings:

```python
sage: P.<x,y,z> = QQ[]
sage: P
Multivariate Polynomial Ring in x, y, z over Rational Field

sage: f = 27/113 * x^2 + y*z + 1/2; f
27/113*x^2 + y*z + 1/2

sage: P.term_order()
Degree reverse lexicographic term order

sage: P = PolynomialRing(GF(127),3,names='abc', order='lex')
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
```

(continues on next page)

(continued from previous page)

```
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 12676506002282901496703205376

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 \( \mathbb{F}_5[x, y, z] \) over \( \mathbb{F}_5 \):

```
sage: R.<x,y,z> = PolynomialRing(GF(5), 3)
sage: frob = R.hom([x^5, y^5, z^5])
sage: frob(x^2 + 2*y - z^4)
-x^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_libsingular.MPolynomialRing_libsingular

Bases: sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base

Construct a multivariate polynomial ring subject to the following conditions:

INPUT:

- **base_ring** - base ring (must be either 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
```

(continues on next page)
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: P.<x,y,z> = Zmod(25213521351515232)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 25213521351515232

sage: P.<x,y,z> = PolynomialRing(Integers(2^32),order='lex')
sage: P(2^32-1)
4294967295

Element

- alias of MPolynomial_libsingular

gen(n=0)

- Returns the n-th generator of this multivariate polynomial ring.

INPUT:
n – an integer \( \geq 0 \)

EXAMPLES:

```python
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
```

```python
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:**

```python
sage: P.<x,y,z> = QQ[]
sage: sage.rings.ideal.Katsura(P)
Ideal (x + 2*y + 2*z - 1, x^2 + 2*y^2 + 2*z^2 - x, 2*x*y + 2*y*z - y) of
Multivariate Polynomial Ring in x, y, z over Rational Field

sage: P.ideal([x + 2*y + 2*z - 1, 2*x*y + 2*y*z - y, x^2 + 2*y^2 + 2*z^2 - x])
Ideal (x + 2*y + 2*z - 1, 2*x*y + 2*y*z - y, x^2 + 2*y^2 + 2*z^2 - x) of
Multivariate Polynomial Ring in x, y, z over Rational Field
```

```python
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:**

```python
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

```python
monomial_divides(a, b)
```

Return False if \( a \) does not divide \( b \) and True otherwise.

**Coefficients are ignored.**

**INPUT:**
• $a$ – monomial
• $b$ – monomial

EXAMPLES:
```python
sage: P.<x,y,z> = QQ[

sage: P.monomial_divides(x*y*z, x^3*y^2*z^4)
True
sage: P.monomial_divides(x^3*y^2*z^4, x*y*z)
False
```

**monomial_lcm** $(f, g)$

LCM for monomials. Coefficients are ignored.

INPUT:
• $f$ - monomial
• $g$ - monomial

EXAMPLES:
```python
sage: P.<x,y,z> = QQ[

sage: P.monomial_lcm(3/2*x*y,x)
x*y
```

**monomial_pairwise_prime** $(g, h)$

Return **True** if $h$ and $g$ are pairwise prime. Both are treated as monomials.

Coefficients are ignored.

INPUT:
• $h$ - monomial
• $g$ - monomial

EXAMPLES:
```python
sage: P.<x,y,z> = QQ[

sage: P.monomial_pairwise_prime(x^2*z^3, y^4)
True
sage: P.monomial_pairwise_prime(1/2*x^3*y^2, 3/4*y^3)
False
```

**monomial_quotient** $(f, g, coeff=False)$

Return $f/g$, 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 \( \mathbb{Z} \) 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.

**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.

\[ \text{monomial\_reduce}(f, G) \]

Try to find a \( g \) in \( G \) where \( g.\text{lm}() \) divides \( f \). If found \((\text{flt}, g)\) is returned, \((0, 0)\) otherwise, where \( \text{flt} = f / g.\text{lm}() \).

It is assumed that \( G \) is iterable and contains only elements in this 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)

\[ \text{ngens()} \]

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
```

coefficient(degrees)
Return the coefficient of the variables with the degrees specified in the python dictionary degrees. Mathematically, this is the coefficient in the base ring adjoined by the variables of this ring not listed in degrees. However, the result has the same parent as this polynomial.

This function contrasts with the function monomial_coefficient which returns the coefficient in the base ring of a monomial.

INPUT:
• degrees - Can be any of:
  – a dictionary of degree restrictions
  – a list of degree restrictions (with None in the unrestricted variables)
  – a monomial (very fast, but not as flexible)

OUTPUT: element of the parent of this element.

Note: For coefficients of specific monomials, look at monomial_coefficient().

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: f=x*y+y+5
sage: f.coefficient({x:0,y:1})
1
sage: f.coefficient({x:0})
y + 5
sage: f=(1+y+y^2)*(1+x+x^2)
sage: f.coefficient({x:0})
y^2 + y + 1
sage: f.coefficient([0,None])
y^2 + y + 1
sage: f.coefficient(x)
y^2 + y + 1
```

Note that exponents have all variables specified:
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,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 \( \text{exponents()} \) for the exponents themselves, or the optional argument \( \text{std_grading=False} \).

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
```

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*y2^2 + q
3*y0*y1*y2^2
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}
```

divides(other)

Return True if this polynomial divides other.

EXAMPLES:

```
sage: R.<x, y, z> = QQ[]
sage: p = 3*x*y + 2*y*z + x*z
sage: q = x + y + z + 1
sage: r = p * q
sage: p.divides(r)
```

(continues on next page)
True

sage: q.divides(p)
False

sage: r.divides(0)
True

sage: R.zero().divides(r)
False

sage: R.zero().divides(0)
True

**exponents** *(as_ETuples=True)*

Return the exponents of the monomials appearing in this polynomial.

INPUT:

- **as_ETuples** *(default: True)* if true returns the result as a 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=None)*

Return the factorization of this polynomial.

INPUT:

- **proof** - ignored.

EXAMPLES:

```
sage: R.<x, y> = QQ[

sage: f = (x^3 + 2*y^2*x) * (x^2 + x + 1); f
x^5 + 2*x^3*y^2 + x^4 + 2*x^2*y^2 + x^3 + 2*x*y^2

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

376 Chapter 3. Multivariate Polynomials
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 trac 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

  sage: P.<x,y> = GF(2)[]

  sage: P(1).factor()
  1
```

Factorization for finite prime fields with characteristic $> 2^{29}$ is not supported.
sage: q = 1073741789
sage: T.<aa, bb> = PolynomialRing(GF(q))
sage: f = aa^2 + 12124343*bb*aa + 32434598*bb^2
sage: f.factor()
Traceback (most recent call last):
  ... 
NotImplementedError: Factorization of multivariate polynomials over prime
  ->fields with characteristic > 2^29 is not implemented.

Factorization over the integers is now supported, see trac ticket #17840:

sage: P.<x,y> = PolynomialRing(ZZ)
sage: f = 12 * (3*x*y + 4) * (5*x - 2) * (2*y + 7)^2
sage: f.factor()
2^2 * 3 * (2*y + 7)^2 * (5*x - 2) * (3*x*y + 4)
sage: g = -12 * (x^2 - y^2)
sage: g.factor()
(-1) * 2^2 * 3 * (x - y) * (x + y)
sage: factor(-4*x*y - 2*x + 2*y + 1)
(-1) * (2*y + 1) * (2*x - 1)

Factorization over non-integral domains is not supported

sage: R.<x,y> = PolynomialRing(Zmod(4))
sage: f = (2*x + 1) * (x^2 + x + 1)
sage: f.factor()
Traceback (most recent call last):
  ... 
NotImplementedError: Factorization of multivariate polynomials over Ring of
  ->integers modulo 4 is not implemented.

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:
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
```

Gradient

Return a list of partial derivatives of this polynomial, ordered by the variables of the parent.

**EXAMPLES:**

```
sage: P.<x,y,z> = PolynomialRing(QQ,3)
sage: f= x*y + 1
sage: f.gradient()
[y, x, 0]
```

Hamming weight

Return the number of non-zero coefficients of this polynomial.

This is also called weight, **hamming_weight()** or sparsity.

**EXAMPLES:**

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

The method **hamming_weight()** is an alias:

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

Integral (var)

Integrates this polynomial with respect to the provided variable.

One requires that Q is contained in the ring.

**INPUT:**

- **variable** - the integral is taken with respect to variable
EXAMPLES:

```
sage: R.<x, y> = PolynomialRing(QQ, 2)
sage: f = 3*x^3*y^2 + 5*y^2 + 3*x + 2
sage: f.integral(x)
3/4*x^4*y^2 + 5*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
```

**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:**

```python
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: x.is_monomial()
True
sage: (2*x).is_monomial()
False
sage: (x*y).is_monomial()
True
sage: (x*y + x).is_monomial()
False
```

**is_squarefree()**
Return True if this polynomial is square free.

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: f= x^2 + 2*x*y + 1/2*z
sage: f.is_squarefree()
True
sage: h = f^2
sage: h.is_squarefree()
False
```

**is_univariate()**
Return True if self is a univariate polynomial, that is if self contains only one variable.

**EXAMPLES:**

```python
sage: P.<x,y,z> = GF(2)[]
sage: f = x^2 + 1
sage: f.is_univariate()
True
sage: f = y*x^2 + 1
sage: f.is_univariate()
False
sage: f = P(0)
sage: f.is_univariate()
True
```

**is_zero()**
Return True if this polynomial is zero.

**EXAMPLES:**

```python
sage: P.<x,y> = PolynomialRing(QQ)
sage: x.is_zero()
False
sage: (x-x).is_zero()
True
```
lc() Leading coefficient of this polynomial with respect to the term order of self.parent().

EXAMPLES:

```python
sage: R.<x,y,z>=PolynomialRing(GF(7),3,order='lex')
sage: f = 5*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:

```python
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:

```python
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]
```

(continues on next page)
sage: sum( map( mul, zip( M, I.gens() ) ) ) == f
True

Check that trac ticket #13671 is fixed:

```python
sage: R.<x1,x2> = QQ[]
sage: I = R.ideal(x2*x2 + x1 - 2, x1*x2 - 1)
sage: f = I.gen(0) + x2*I.gen(1)
sage: f.lift(I)
[1, x2]
sage: (f+1).lift(I)
Traceback (most recent call last):
...
ValueError: polynomial is not in the ideal
sage: f.lift(I)
[1, x2]
```

`lm()`

Returns the lead monomial of self with respect to the term order of `self.parent()`. In Sage a monomial is a product of variables in some power without a coefficient.

EXAMPLES:

```python
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:

```python
sage: R.<x,y,z>=PolynomialRing(GF(7),3,order='lex')
sage: f = 3*x^1*y^2 + 2*y^3*z^4
sage: f.lt()
3*x*y^2
```

(continues on next page)
monomial_coefficient (mon)  
Return the coefficient in the base ring of the monomial mon in self, where mon must have the same parent as self.

This function contrasts with the function coefficient which returns the coefficient of a monomial viewing this polynomial in a polynomial ring over a base ring having fewer variables.

INPUT:

• mon - a monomial

OUTPUT:

coefficient in base ring

See also:

For coefficients in a base ring of fewer variables, look at coefficient.

EXAMPLES:

```
sage: P.<x,y> = QQ[]
The parent of the return is a member of the base ring.
sage: f = 2 * x * y
sage: c = f.monomial_coefficient(x*y); c
2
sage: c.parent()
Rational Field
sage: f = y^2 + y^2*x - x^9 - 7*x + 5*x*y
sage: f.monomial_coefficient(y^2)
1
sage: f.monomial_coefficient(x*y)
5
sage: f.monomial_coefficient(x^9)
-1
sage: f.monomial_coefficient(x^10)
0
```

monomials ()
Return the list of monomials in self. The returned list is decreasingly ordered by the term ordering of self.parent().

EXAMPLES:

```
sage: P.<x,y,z> = QQ[]
sage: f = x + 3/2*y*z^2 + 2/3
sage: f.monomials()
[y*z^2, x, 1]
sage: f = P(3/2)
sage: f.monomials()
[1]
```

number_of_terms ()
Return the number of non-zero coefficients of this polynomial.
This is also called weight, \texttt{hamming\_weight()} or sparsity.

**EXAMPLES:**

```python
sage: R.<x, y> = ZZ[]
sage: f = x^3 - y
sage: f.number_of_terms()
sage: R(0).number_of_terms()
sage: f = (x+y)^100
sage: f.number_of_terms()
```

The method \texttt{hamming\_weight()} is an alias:

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

\texttt{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.

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

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

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

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

```python
sage: K=QQ['x,y']
sage: f=K.random_element()
sage: f.numerator() / f.denominator() == f
```
The following tests against a bug fixed in trac ticket #11780:

```python
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:**

```python
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:**

```python
sage: R.<x,y> = QQ[]
sage: f = y*x^2 + x + 1
sage: f.quo_rem(x)
(x*y + 1, 1)
sage: f.quo_rem(y)
(x^2, x + 1)

sage: R.<x,y> = ZZ[]
sage: f = 2*y*x^2 + x + 1
sage: f.quo_rem(x)
(2*x*y + 1, 1)
sage: f.quo_rem(y)
(2*x^2, x + 1)
sage: f.quo_rem(3*x)
(0, 2*x^2*y + x + 1)
```

`reduce(I)`

Return a remainder of this polynomial modulo the polynomials in `I`.

**INPUT:**

- `I` - an ideal or a list/set/iterable of polynomials.

**OUTPUT:**

A polynomial `r` such that:

- `self - r` is in the ideal generated by `I`.
- No term in `r` is divisible by any of the leading monomials of `I`.

The result `r` is canonical if:

- `I` is an ideal, and Sage can compute a Groebner basis of it.
• $I$ is a list/set/iterable that is a (strong) Groebner basis for the term order of $\text{self}$. (A strong Groebner basis is such that for every leading term $t$ of the ideal generated by $I$, there exists an element $g$ of $I$ such that the leading term of $g$ divides $t$.)

The result $r$ is implementation-dependent (and possibly order-dependent) otherwise. If $I$ is an ideal and no Groebner basis can be computed, its list of generators $I.gens()$ is used for the reduction.

EXAMPLES:

```
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
```

The reduction is not canonical when $I$ is not a Groebner basis:

```
sage: A.<x,y> = QQ[]
sage: (x+y).reduce([x+y, x-y])
2*y
sage: (x+y).reduce([x-y, x+y])
0
```

**resultant** (*other*, *variable=None*)

Compute the resultant of this polynomial and the first argument with respect to the variable given as the second argument.

If a second argument is not provide the first variable of the parent is chosen.

INPUT:

• *other* - polynomial
  
• *variable* - optional variable (default: None)

EXAMPLES:
```python
sage: P.<x,y> = PolynomialRing(QQ,2)
sage: a = x+y
case: b = x^3-y^3
sage: c = a.resultant(b); c
-2*y^3
case: d = a.resultant(b,y); d
2*x^3
```

The SINGULAR example:

```python
sage: R.<x,y,z> = PolynomialRing(GF(32003),3)
sage: f = 3 * (x+2)^3 + y
case: 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:

```python
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 + 2*y^4*a^3*b^2 - 2*y^4*a^2*b + 6*y^4*a^2*u^2 - 2*y^4*a*u + 4*y^4*b^2 - 4*y^3*a^4*b^2 - 12*y^3*a^3*b^2 - 2*y^3*a^2*b^2 + 6*y^3*a^2*u^2 - 2*y^3*a*b^2 + 4*y^3*a*b*u - 4*y^3*b^2 + 12*y^3*a^2*b - 12*y^3*a*b*u^2 + 6*y^3*a*b + 6*y^3*a*u - 2*y^3*a + 6*y^3*b^2 - 12*y^3*b + y^3*a^3*b^3 + y^3*a^2*b^2 + 6*y^3*a^2*b^2 - 6*y^3*a*b^2 + 6*y^3*a*b*u^2 + 6*y^3*a*b + 6*y^3*a + y^3*b^2 - 6*y^3*b + y^3*u^2 + 6*y^3*u - 6*y^3 + 4*y^2*a^4*b^4 + 4*y^2*a^3*b^4 - 4*y^2*a^3*b^3 - 4*y^2*a^2*b^4 - 4*y^2*a^2*b^3 - 4*y^2*a*b^4 - 4*y^2*a*b^3 - 4*y^2*b^3 - 2*y^2 - 2*b^2*u^2 + y^2*u^2 + y*a^2*b^4 + 2*y*a*b^4 + 2*y*a*b^3*u - y*b^4 + y*b^2*u^2
```

**sub_m_mul_q** (*m*, *q*)

Return self - m*q, where m must be a monomial and q a polynomial.

**INPUT:**

- m - a monomial
- q - a polynomial

**EXAMPLES:**

```python
sage: P.<x,y,z>=PolynomialRing(QQ,3)
sage: x.sub_m_mul_q(y,z)
-y*z + x
```

**subs** *(fixed=None, **kw)*

Fixes some given variables in a given multivariate polynomial and returns the changed multivariate polynomials. The polynomial itself is not affected. The variable,value pairs for fixing are to be provided as dictionary of the form `{variable:value}`.

This is a special case of evaluating the polynomial with some of the variables constants and the others the original variables, but should be much faster if only few variables are to be fixed.

**INPUT:**

- fixed - (optional) dict with variable:value pairs
- **kw** - names parameters

**OUTPUT:** a new multivariate polynomial
EXEMPLARY:

```python
sage: R.<x,y> = QQ[]
sage: f = x^2 + y + x^2*y^2 + 5
sage: f(5,y)
25*y^2 + y + 30
sage: f.subs((x:5))
25*y^2 + y + 30
sage: f.subs(x=5)
25*y^2 + y + 30
sage: P.<x,y,z> = PolynomialRing(GF(2),3)
sage: f = x + y + 1
sage: f.subs((x:y+1))
0
sage: f.subs(x=y)
1
sage: f.subs(x=x)
x + y + 1
sage: f.subs((x:z))
y + z + 1
sage: f.subs(x=z+1)
y + z
sage: f.subs(x=1/y)
(y^2 + y + 1)/y
sage: f.subs((x:1/y))
(y^2 + y + 1)/y
```

The parameters are substituted in order and without side effects:

```python
sage: R.<x,y>=QQ[]
sage: g=x+y
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
```

```python
sage: R.<x,y> = QQ[]
sage: f = x + 2*y
sage: f.subs(x=y,y=x)
2*x + y
```

**total_degree** *(std_grading=False)*

Return the total degree of *self*, which is the maximum degree of all monomials in *self*.

EXAMPLES:

```python
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
```

**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`

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
```

3.1.9 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
```

3.1. Multivariate Polynomials and Polynomial Rings
Two examples from the Mathematica documentation (done in Sage):

- We compute a Groebner basis:

```python
sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: ideal(x^2 - 2*y^2, x*y - 3).groebner_basis('libsingular:slimgb')
[x - 2/3*y^3, y^4 - 9/2]
```

- We show that three polynomials have no common root:

```python
sage: R.<x,y> = QQ[]
sage: ideal(x+y, x^2 - 1, y^2 - 2*x).groebner_basis('libsingular:slimgb')
[1]
```

SINGULAR's `interred()` command.

**INPUT:**

- `I` – a Sage ideal

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(ZZ)
sage: I = ideal( x^2 - 3*y, y^3 - x*y, z^3 - x, x^4 - y*z + 1 )
sage: I.interreduced_basis()
[y*z^2 - 81*x*y - 9*y - z, z^3 - x, x^2 - 3*y, 9*y^2 - y*z + 1]
```

```python
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]
```

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:**

```python
sage: P.<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]
```

SINGULAR’s `slimgb()` algorithm.

**INPUT:**

- `I` – a Sage ideal

SINGULAR’s `std()` algorithm.
INPUT:

- \( I \) – a Sage ideal

### 3.1.10 PolyDict engine for generic multivariate polynomial rings

This module provides an implementation of the underlying arithmetic for multi-variate polynomial rings using Python dicts.

This class is not meant for end users, but instead for implementing multivariate polynomial rings over a completely general base. It does not do strong type checking or have parents, etc. For speed, it has been implemented in Cython.

The functions in this file use the ‘dictionary representation’ of multivariate polynomials

\[
\{(e_1, \ldots, e_r): c_1, \ldots\} \leftrightarrow c_1 x_1^{e_1} \cdots x_r^{e_r} + \cdots,
\]

which we call a polydict. The exponent tuple \((e_1, \ldots, e_r)\) in this representation is an instance of the class `ETuple`. This class behaves like a normal Python tuple but also offers advanced access methods for sparse monomials like positions of non-zero exponents etc.

AUTHORS:

- William Stein
- David Joyner
- Martin Albrecht (ETuple)
- Joel B. Mohler (2008-03-17) – ETuple rewrite as sparse C array

```python
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:

        ```python
        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:
```
dotprod\( (other) \)
Return the dot product of this tuple by \( other \).

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
sage: f = ETuple([0,1,1])
sage: e.dotprod(f)
2
sage: e = ETuple([1,1,-1])
sage: f = ETuple([0,-2,1])
sage: e.dotprod(f)
-3
```

eadd\( (other) \)
Vector addition of \( self \) with \( other \).

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
sage: f = ETuple([0,1,1])
sage: e.eadd(f)
(1, 1, 3)
```

Verify that trac ticket \#6428 has been addressed:

```python
sage: R.<y, z> = Frac(QQ['x'])[]
sage: type(y)
:class 'sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict'>
sage: y^(2^32)
Traceback (most recent call last):
...
OverflowError: exponent overflow (2147483648)
```

eadd_p\( (other, pos) \)
Add \( other \) to \( self \) at position \( pos \).

**EXAMPLES:**

```python
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:

```python
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: e.emax(f) (4, 2, 3, 4)
sage: e = ETuple((1,-2,-2,4))
sage: f = ETuple((4,0,0,0))
sage: e.emax(f) (4, 0, 0, 4)
sage: e.emax(f).nonzero_positions() [0, 3]
```

emin(other)
Vector of minimum of components of self and other.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
sage: f = ETuple([0,1,1])
sage: e.emin(f) (0, 0, 1)
sage: e = ETuple([1,0,-1])
sage: f = ETuple([0,-2,1])
sage: e.emin(f) (0, -2, -1)
```

emul(factor)
Scalar Vector multiplication of self.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
sage: e.emul(2) (2, 0, 4)
```

escalar_div(n)
Divide each exponent by n.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: ETuple([1,0,2]).escalar_div(2) (0, 0, 1)
```

(continues on next page)
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_values(sort=True)
Return the non-zero values of the tuple.

INPUT:

• sort – (default: True) if True the values are sorted by their indices; otherwise the values are returned unsorted

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([2,0,1])
sage: e.nonzero_values()
[2, 1]
sage: f = ETuple([0,-1,1])
sage: f.nonzero_values(sort=True)
[-1, 1]
```

reversed()
Return the reversed ETuple of self.

EXAMPLES:

```
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)]
```

unweighted_degree()
Return the sum of entries.

ASSUMPTION:
All entries are non-negative.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2,0])
sage: e.unweighted_degree()
4
```

class sage.rings.polynomial.polydict.PolyDict
Bases: object

INPUT:

• pdict – dict or list, which represents a multi-variable polynomial with the distribute representation (a copy is not made)

• zero – (optional) zero in the base ring
• `force_int_exponents` — bool (optional) arithmetic with int exponents is much faster than some of the alternatives, so this is True by default.

• `force_etuples` — bool (optional) enforce that the exponent tuples are instances of ETuple class

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polydict import PolyDict
sage: PolyDict({(2,3):2, (1,2):3, (2,1):4})
PolyDict with representation {(1, 2): 3, (2, 1): 4, (2, 3): 2}

# I've removed fractional exponent support in ETuple when moving to a sparse C_ integer array
# PolyDict with representation {(2, 1): 4, (1, 2, 1): 3, (2/3, 3, 5): 2}

sage: PolyDict({(2,3):0, (1,2):3, (2,1):4}, remove_zero=True)
PolyDict with representation {(1, 2): 3, (2, 1): 4}

# sage: PolyDict({(0,0):RIF(-1,1)}, remove_zero=True)
# PolyDict with representation {(0, 0): 0.?}
```

**coefficient** *(mon)*

Return a polydict that defines a polynomial in 1 less number of variables that gives the coefficient of mon in this polynomial.

The coefficient is defined as follows. If \( f \) is this polynomial, then the coefficient is the sum \( T/\text{mon} \) where the sum is over terms \( T \) in \( f \) that are exactly divisible by mon.

**coefficients** ()

Return the coefficients of self.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: sorted(f.coefficients())
[2, 3, 4]
```

**degree** *(x=\text{None})*

**dict** ()

Return a copy of the dict that defines self. It is safe to change this. For a reference, use dictref.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: sorted(f.dict())
[(1, 2): 3, (2, 1): 4, (2, 3): 2]
```

**exponents** ()

Return the exponents of self.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: sorted(f.exponents())
[(1, 2), (2, 1), (2, 3)]
```
**homogenize**(var)

**is_homogeneous**()

**latex**(vars, atomic_exponents=True, atomic_coefficients=True, sortkey=None)

Return a nice polynomial latex representation of this PolyDict, where the vars are substituted in.

**INPUT:**

- **vars** – list
- **atomic_exponents** – bool (default: True)
- **atomic_coefficients** – bool (default: True)

**EXAMPLES:**

```sage
def from sage.rings.polynomial.polydict import PolyDict
def f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
def 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:

```sage
# I've removed fractional exponent support in ETuple when moving to a sparse
# C integer array
#          ->exponents=False)
sage: f.latex(['a', 'b', 'c'], atomic_exponents=False)
'4 a^{2}bc + 3 ab^{2}c + 2 a^{2/3}b^{3}c^{5}'
```

**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
def from sage.rings.polynomial.polydict import PolyDict
def f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
def sorted(f.list())
[[2, [2, 3]], [3, [1, 2]], [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
def from sage.rings.polynomial.polydict import PolyDict
def f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
def f.max_exp()
```

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

```python
(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 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:**

```python
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)**

**INPUT:**

a PolyDict with a single key

**EXAMPLES:**

```python
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, sortkey=None)**

Return a nice polynomial string representation of this PolyDict, where the vars are substituted in.

**INPUT:**

- `vars` – list
- `atomic_exponents` – bool (default: True)
- `atomic_coefficients` – bool (default: True)

**EXAMPLES:**

```python
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'], atomic_exponents=False)
'2*a^2*WW^3 + 4*a^2*WW + 3*a*WW^2'
```

We check to make sure that when we are in characteristic two, we don’t put negative signs on the generators.
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, 1): 4, (0, 3): 2}
sage: f = PolyDict({(0,3):2, (0,2):3, (2,1):4})
sage: f.polynomial_coefficient([0,None])
PolyDict with representation {(0, 2): 3, (0, 3): 2}
```

rich_compare (other, op=None, sortkey=None)

Compare two \( \mathcal{P} \text{olyDict} \) argument is given it should be a sort key used to specify a term order.

If not sort key is provided than only comparison by equality (== or !=) is supported.

**EXAMPLES:**

```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: from sage.structure.richcmp import op_EQ, op_NE, op_LT
sage: p1 = PolyDict({(0,): 1})
sage: p2 = PolyDict({(0,): 2})
sage: p1.rich_compare(PolyDict({(0,): 1}), op_EQ)
True
sage: p1.rich_compare(p2, op_EQ)
False
sage: p1.rich_compare(p2, op_NE)
True
sage: p1.rich_compare(p2, op_LT)
Traceback (most recent call last):
... TypeError: ordering of PolyDicts requires a sortkey
```

```
sage: O = TermOrder()
```

```
sage: p1.rich_compare(p2, op_LT, O.sortkey)
True
```

```
sage: p3 = PolyDict({(3, 2, 4): 1, (3, 2, 5): 2})
sage: p4 = PolyDict({(3, 2, 4): 1, (3, 2, 3): 2})
sage: p3.rich_compare(p4, op_LT, O.sortkey)
False
```

3.1. Multivariate Polynomials and Polynomial Rings 401
scalar_lmult \( (s) \)
Left Scalar Multiplication

EXAMPLES:

```python
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-commutative!
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, 1): -8, (2, 3): -4}
sage: f.scalar_lmult(RIF(-1,1))
PolyDict with representation {(1, 2): 0.?e1, (2, 1): 0.?e1, (2, 3): 0.?e1}
```

scalar_rmult \( (s) \)
Right Scalar Multiplication

EXAMPLES:

```python
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-commutative!
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, 1): -8, (2, 3): -4}
sage: f.scalar_rmult(RIF(-1,1))
PolyDict with representation {(1, 2): 0.?e1, (2, 1): 0.?e1, (2, 3): 0.?e1}
```

term_lmult \( (exponent, s) \)
Return this element multiplied by \( s \) on the left and with exponents shifted by \( exponent \).

INPUT:

- \( exponent \) – a ETuple
- \( s \) – a scalar

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import ETuple, PolyDict
sage: x, y = FreeMonoid(2, 'x, y').gens()  # a strange object to live in a
→polydict, but non-commutative!
sage: f = PolyDict({(2,3):x})
sage: f.term_lmult(ETuple((1, 2)), y)
PolyDict with representation {(3, 5): y*x}
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: f.term_lmult(ETuple((1, 2)), -2)
PolyDict with representation {(3, 5): y*x}
```

term_rmult \( (exponent, s) \)
Return this element multiplied by \( s \) on the right and with exponents shifted by \( exponent \).

INPUT:
• exponent – a \texttt{ETuple}  
• \( s \) – a scalar

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import ETuple, PolyDict
sage: x, y = FreeMonoid(2, 'x, y').gens()  # a strange object to live in a polydict, but non-commutative!
  sage: f = PolyDict({(2, 3): x})
  sage: f.term_rmult(ETuple((1, 2)), y)
PolyDict with representation {3, 5: x*y}
  sage: f = PolyDict({(2,3): 2, (1,2): 3, (2,1): 4})
  sage: f.term_rmult(ETuple((1, 2)), -2)
PolyDict with representation {2, 4: -6, 3, 3: -8, 3, 5: -4}
```

```python
total_degree()
valuation(x=None)
sage.rings.polynomial.polydict.make_ETuple(data, length)
sage.rings.polynomial.polydict.make_PolyDict(data)
```

### 3.1.11 Compute Hilbert series of monomial ideals

This implementation was provided at trac ticket \#26243 and is supposed to be a way out when Singular fails with an int overflow, which will regularly be the case in any example with more than 34 variables.

```python
class sage.rings.polynomial.hilbert.Node
  Bases: object
  A node of a binary tree
  It has slots for data that allow to recursively compute the first Hilbert series of a monomial ideal.

def first_hilbert_series(I, grading=None, return_grading=False)
  Return the first Hilbert series of the given monomial ideal.

  INPUT:
  • I – a monomial ideal (possibly defined in singular)
  • grading – (optional) a list or tuple of integers used as degree weights
  • return_grading – (default: False) whether to return the grading

  OUTPUT:
  A univariate polynomial, namely the first Hilbert function of I, and if \texttt{return\_grading==True} also the grading used to compute the series.

EXAMPLES:

```python
sage: from sage.rings.polynomial.hilbert import first_hilbert_series
sage: R = singular.ring(0,'(x,y,z)','dp')
  sage: I = singular.ideal(['x^2','y^2','z^2'])
  sage: first_hilbert_series(I)
-t^6 + 3*t^4 - 3*t^2 + 1
  sage: first_hilbert_series(I,return_grading=True)
```
```
sage: first_hilbert_series(I, grading=(1, 2, 3))
-t^12 + t^10 + t^8 - t^4 - t^2 + 1

sage.rings.polynomial.hilbert.hilbert_poincare_series(I, grading=None)

Return the Hilbert Poincaré series of the given monomial ideal.

**INPUT:**

- I - a monomial ideal (possibly defined in Singular)
- grading – (optional) a tuple of degree weights

**EXAMPLES:**

```sage
sage: from sage.rings.polynomial.hilbert import hilbert_poincare_series
sage: R = PolynomialRing(QQ, 'x', 9)
sage: I = [m.lm() for m in ((matrix(R, 3, R.gens())^2).list()*R).groebner_basis()]*R
sage: hilbert_poincare_series(I)
(t^7 - 3*t^6 + 2*t^5 + 2*t^4 - 2*t^3 + 6*t^2 + 5*t + 1)/(t^4 - 4*t^3 + 6*t^2 - 4*t + 1)

sage: hilbert_poincare_series((R*R.gens())^2, grading=range(1,10))
t^9 + t^8 + t^7 + t^6 + t^5 + t^4 + t^3 + t^2 + t + 1
```

The following example is taken from [trac ticket #20145](https://trac.sagemath.org/ticket/20145):

```sage
sage: n=4;m=11;P = PolynomialRing(QQ,n+m,"x"); x = P.gens(); M = Matrix(n,x)
```

```sage
doctest:...: warning: Could not import _gp (missing from Sage?)
```

```sage
sage: from sage.rings.polynomial.hilbert import first_hilbert_series
sage: I = P.ideal(M.minors(2))
sage: J = P*[m.lm() for m in I.groebner_basis()]
sage: hilbert_poincare_series(J).numerator()
120*t^3 + 135*t^2 + 30*t + 1

sage: hilbert_poincare_series(J).denominator().factor()
(t - 1)^14
```

This example exceeds the current capabilities of Singular:

```sage
sage: J.hilbert_numerator(algorithm='singular')
Traceback (most recent call last):
  ...
RuntimeError: error in Singular function call 'hilb'
  int overflow in hilb 1
```

### 3.1.12 Class to flatten polynomial rings over polynomial ring

For example \(\mathbb{Q}[a,b], [x,y]\) flattens to \(\mathbb{Q}[a',b',x',y']\).

**EXAMPLES:**

```sage
sage: R = QQ['x']['y']['s','t']['X']
sage: from sage.rings.polynomial.flatten import FlatteningMorphism
sage: phi = FlatteningMorphism(R); phi
Flattening morphism:
  From: Univariate Polynomial Ring in X over Multivariate Polynomial Ring in s, t over Univariate Polynomial Ring in y over Univariate Polynomial Ring in x over Rational Field
  To:   Multivariate Polynomial Ring in x, y, s, t, X over Rational Field
```
sage: phi('x*y*s + t*X').parent()
Multivariate Polynomial Ring in x, y, s, t, X over Rational Field

Authors:
Vincent Delecroix, Ben Hutz (July 2016): initial implementation

class sage.rings.polynomial.flatten.FlatteningMorphism(domain)
    Bases: sage.categories.morphism.Morphism

EXAMPLES:

Also works when univariate polynomial ring are involved:

section()
    Inverse of this flattening morphism.

EXAMPLES:

sage: R = ZZ['a','b','c']
sage: from sage.rings.polynomial.flatten import FlatteningMorphism
sage: FlatteningMorphism(R).section()
Unflattening morphism:
From: Multivariate Polynomial Ring in a, b, c over Integer Ring
To: Univariate Polynomial Ring in c over Univariate Polynomial Ring in b, over Univariate Polynomial Ring in a over Integer Ring

```
class sage.rings.polynomial.flatten.FractionSpecializationMorphism(domain, D)
Bases: sage.categories.morphism.Morphism
A specialization morphism for fraction fields over (stacked) polynomial rings
```

```
class sage.rings.polynomial.flatten.SpecializationMorphism(domain, D)
Bases: sage.categories.morphism.Morphism
Morphisms to specialize parameters in (stacked) polynomial rings
EXAMPLES:
```
sage: R.<c> = PolynomialRing(QQ)
sage: S.<x,y,z> = PolynomialRing(R)
sage: D = dict({c:1})
sage: from sage.rings.polynomial.flatten import SpecializationMorphism
sage: f = SpecializationMorphism(S, D)
sage: g = f(x^2 + c*y^2 - z^2); g
x^2 + y^2 - z^2
sage: g.parent()
Multivariate Polynomial Ring in x, y, z over Rational Field
```
```
sage: R.<c> = PolynomialRing(QQ)
sage: S.<z> = PolynomialRing(R)
sage: D = dict({c:0})
sage: from sage.rings.polynomial.flatten import SpecializationMorphism
sage: xi = SpecializationMorphism(S, D); xi
Specialization morphism:
From: Univariate Polynomial Ring in z over Univariate Polynomial Ring in c, over Rational Field
To: Univariate Polynomial Ring in z over Rational Field
sage: xi(z^2+c)
z^2
```
```
sage: R1.<u,v> = PolynomialRing(QQ)
sage: R2.<a,b,c> = PolynomialRing(R1)
sage: S.<x,y,z> = PolynomialRing(R2)
sage: D = dict({a:1, b:2, x:0, u:1})
sage: from sage.rings.polynomial.flatten import SpecializationMorphism
sage: xi = SpecializationMorphism(S, D); xi
Specialization morphism:
From: Multivariate Polynomial Ring in x, y, z over Multivariate Polynomial Ring in a, b, c over Multivariate Polynomial Ring in u, v over Rational Field
To: Multivariate Polynomial Ring in y, z over Univariate Polynomial Ring in c, over Univariate Polynomial Ring in v over Rational Field
sage: xi(a*(x*z+y^2)*u+b*v*u*(x*z+y^2)*y^2+c+y^2*z^2)
2*v*c+y^4 + c*y^2*z^2 + y^2
```
```
class sage.rings.polynomial.flatten.UnflatteningMorphism(domain, codomain)
Bases: sage.categories.morphism.Morphism
Inverses for FlatteningMorphism
EXAMPLES:
```
sage: R = QQ['c','x','y','z']
sage: S = QQ['c']['x','y','z']
sage: from sage.rings.polynomial.flatten import UnflatteningMorphism
sage: f = UnflatteningMorphism(R, S)
sage: g = f(R('x^2 + c*y^2 - z^2')); g
x^2 + c*y^2 - z^2
sage: g.parent()
Multivariate Polynomial Ring in x, y, z over Univariate Polynomial Ring in c over Rational Field

sage: R = QQ['a','b', 'x','y']
sage: S = QQ['a','b']['x','y']
sage: UnflatteningMorphism(R, S)
Unflattening morphism:
  From: Multivariate Polynomial Ring in a, b, x, y over Rational Field
  To:  Multivariate Polynomial Ring in x, y over Multivariate Polynomial Ring in a, b over Rational Field

3.1.13 Monomials

sage.rings.monomials.monomials(v, n)

Given two lists \(v\) and \(n\), of exactly the same length, return all monomials in the elements of \(v\), where variable \(i\) (i.e., \(v[i]\)) in the monomial appears to degree strictly less than \(n[i]\).

INPUT:

- \(v\) – list of ring elements
- \(n\) – list of integers

EXAMPLES:

sage: monomials([x], [3])
[1, x, x^2]
sage: R.<x,y,z> = QQ[]
sage: monomials([x,y], [5,5])
[1, x*y, x*y^2, x*y^3, x*y^4, x^2, x^2*y, x^2*y^2, x^2*y^3, x^2*y^4, x^3, x^3*y, x^3*y^2, x^3*y^3, x^3*y^4, x^4, x^4*y, x^4*y^2, x^4*y^3, x^4*y^4]
sage: monomials([x,y,z], [2,3,2])
[1, x*y, x*y*z, x*y^2, x*y^2*z, x*y*z, x*y^2*z]

3.2 Classical Invariant Theory

3.2.1 Classical Invariant Theory

This module lists classical invariants and covariants of homogeneous polynomials (also called algebraic forms) under the action of the special linear group. That is, we are dealing with polynomials of degree \(d\) in \(n\) variables. The special linear group \(SL(n, C)\) acts on the variables \((x_1, \ldots, x_n)\) linearly.

MATH:
The linear action on the variables transforms a polynomial $p$ generally into a different polynomial $gp$. We can think of it as an action on the space of coefficients in $p$. An invariant is a polynomial in the coefficients that is invariant under this action. A covariant is a polynomial in the coefficients and the variables $(x_1, \ldots, x_n)$ that is invariant under the combined action.

For example, the binary quadratic $p(x, y) = ax^2 + bxy + cy^2$ has as its invariant the discriminant $\text{disc}(p) = b^2 - 4ac$. This means that for any $SL(2, \mathbb{C})$ coordinate change

$$
\begin{pmatrix}
  x' \\
  y'
\end{pmatrix}
= 
\begin{pmatrix}
  \alpha & \beta \\
  \gamma & \delta
\end{pmatrix}
\begin{pmatrix}
  x \\
  y
\end{pmatrix}
\alpha\delta - \beta\gamma = 1
$$

the discriminant is invariant, $\text{disc}(p(x', y')) = \text{disc}(p(x, y))$.

To use this module, you should use the factory object `invariant_theory`. For example, take the quartic:

```
sage: R.<x,y> = QQ[]
sage: q = x^4 + y^4
sage: quartic = invariant_theory.binary_quartic(q); quartic
Binary quartic with coefficients (1, 0, 0, 0, 1)
```

One invariant of a quartic is known as the Eisenstein D-invariant. Since it is an invariant, it is a polynomial in the coefficients (which are integers in this example):

```
sage: quartic.EisensteinD()
1
```

One example of a covariant of a quartic is the so-called $g$-covariant (actually, the Hessian). As with all covariants, it is a polynomial in $x, y$ and the coefficients:

```
sage: quartic.g_covariant()
-x^2*y^2
```

As usual, use tab completion and the online help to discover the implemented invariants and covariants.

In general, the variables of the defining polynomial cannot be guessed. For example, the zero polynomial can be thought of as a homogeneous polynomial of any degree. Also, since we also want to allow polynomial coefficients we cannot just take all variables of the polynomial ring as the variables of the form. This is why you will have to specify the variables explicitly if there is any potential ambiguity. For example:

```
sage: invariant_theory.binary_quartic(R.zero(), [x,y])
Binary quartic with coefficients (0, 0, 0, 0, 0)
sage: invariant_theory.binary_quartic(x^4, [x,y])
Binary quartic with coefficients (0, 0, 0, 1)
sage: R.<x,y,t> = QQ[]
sage: invariant_theory.binary_quartic(x^4 + y^4 + t*x^2*y^2, [x,y])
Binary quartic with coefficients (1, 0, t, 0, 1)
```

Finally, it is often convenient to use inhomogeneous polynomials where it is understood that one wants to homogenize them. This is also supported, just define the form with an inhomogeneous polynomial and specify one less variable:
```python
sage: R.<x,t> = QQ[]
sage: invariant_theory.binary_quartic(x^4 + 1 + t*x^2, [x])
Binary quartic with coefficients (1, 0, t, 0, 1)
```

REFERENCES:
- Wikipedia article Glossary_of_invariant_theory

AUTHORS:
- Volker Braun (2013-01-24): initial version
- Jesper Noordsij (2018-05-18): support for binary quintics added

```python
class sage.rings.invariants.invariant_theory.AlgebraicForm(n, d, polynomial, *args, **kwds)
```

The base class of algebraic forms (i.e. homogeneous polynomials). You should only instantiate the derived classes of this base class. Derived classes must implement `coeffs()` and `scaled_coeffs()`

INPUT:
- `n` – The number of variables.
- `d` – The degree of the polynomial.
- `polynomial` – The polynomial.
- `*args` – The variables, as a single list/tuple, multiple arguments, or `None` to use all variables of the polynomial.

Called classes must implement the same arguments for the constructor.

EXAMPLES:
```python
sage: from sage.rings.invariants.invariant_theory import AlgebraicForm
sage: R.<x,y> = QQ[]
sage: p = x^2 + y^2
sage: AlgebraicForm(2, 2, p).variables()
(x, y)
sage: AlgebraicForm(2, 2, p, None).variables()
(x, y)
sage: AlgebraicForm(3, 2, p).variables()
(x, y, None)
sage: AlgebraicForm(3, 2, p, None).variables()
(x, y, None)
```

(continues on next page)
sage: AlgebraicForm(2, 1, p, [x, y]).variables()
Traceback (most recent call last):
  ...
ValueError: polynomial is of the wrong degree

sage: AlgebraicForm(2, 2, x^2+y, [x, y]).variables()
Traceback (most recent call last):
  ...
ValueError: polynomial is not homogeneous

coefficients()

Alias for coeffs().

See the documentation for coeffs() for details.

EXAMPLES:

```python
sage: R.<a,b,c,d,e,f,g, x,y,z> = QQ[
```

```python
sage: p = a*x^2 + b*y^2 + c*z^2 + d*x*y + e*x*z + f*y*z
```

```python
sage: q = invariant_theory.quadratic_form(p, x,y,z)
sage: q.coefficients()
(a, b, c, d, e, f)
sage: q.coeffs()
(a, b, c, d, e, f)
```

form()

Return the defining polynomial.

OUTPUT:

The polynomial used to define the algebraic form.

EXAMPLES:

```python
sage: R.<x,y> = QQ[
```

```python
sage: quartic = invariant_theory.binary_quartic(x^4+y^4)
sage: quartic
Binary quadratic with coefficients (1, 3, 2)
sage: quartic.form()
x^4 + y^4
```

homogenized(var='h')

Return form as defined by a homogeneous polynomial.

INPUT:

- var – either a variable name, variable index or a variable (default: 'h').

OUTPUT:

The same algebraic form, but defined by a homogeneous polynomial.

EXAMPLES:

```python
sage: T.<t> = QQ[
```

```python
sage: quadratic = invariant_theory.binary_quadratic(t^2 + 2*t + 3)
sage: quadratic
```

```python
Binary quadratic with coefficients (1, 3, 2)
sage: quadratic.homogenized()
```

(continues on next page)
Binary quadratic with coefficients (1, 3, 2)
sage: quadratic == quadratic.homogenized()
True
sage: quadratic.form()
t^2 + 2*t + 3
sage: quadratic.homogenized().form()
t^2 + 2*t*h + 3*h^2
sage: R.<x,y,z> = QQ[]
sage: quadratic = invariant_theory.ternary_quadratic(x^2 + 1, [x,y])
sage: quadratic.homogenized().form()
x^2 + h^2
sage: R.<x> = QQ[]
sage: quintic = invariant_theory.binary_quintic(x^4 + 1, x)
sage: quintic.homogenized().form()
x^4*h + h^5

polynomial()
Return the defining polynomial.

OUTPUT:
The polynomial used to define the algebraic form.

EXAMPLES:

sage: R.<x,y> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+y^4)
sage: quartic.form()
x^4 + y^4
sage: quartic.polynomial()
x^4 + y^4

transformed(g)
Return the image under a linear transformation of the variables.

INPUT:

* g – a GL(n, C) matrix or a dictionary with the variables as keys. A matrix is used to define the linear transformation of homogeneous variables, a dictionary acts by substitution of the variables.

OUTPUT:

A new instance of a subclass of AlgebraicForm obtained by replacing the variables of the homogeneous polynomial by their image under g.

EXAMPLES:

sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3 + 2*y^3 + 3*z^3 + 4*x*y*z)
sage: cubic.transformed({x:y, y:z, z:x}).form()
3*x^3 + y^3 + 4*x*y*z + 2*z^3
sage: cyc = matrix([[2,0,0],[0,1,0],[0,0,1]])
sage: cubic.transformed(cyc) == cubic.transformed({x:y, y:z, z:x})
True
sage: g = matrix(QQ, [[1, 0, 0], [-1, 1, -3], [-5, -5, 16]])
sage: cubic.transformed(g)
Ternary cubic with coefficients (-356, -373, 12234, -1119, 3578, -1151,

(continues on next page)
sage: cubic.transformed(g).transformed(g.inverse()) == cubic
True

class sage.rings.invariants.invariant_theory.BinaryQuartic(n, d, polynomial, *args)

Bases: sage.rings.invariants.invariant_theory.AlgebraicForm

Invariant theory of a binary quartic.

You should use the invariant_theory factory object to construct instances of this class. See binary_quartic() for details.

EisensteinD()

One of the Eisenstein invariants of a binary quartic.

OUTPUT:

The Eisenstein D-invariant of the quartic.

\[ f(x) = a_0 x_1^4 + 4a_1 x_0 x_1^3 + 6a_2 x_0^2 x_1^2 + 4a_3 x_0^3 x_1 + a_4 x_0^4 \]
\[ \Rightarrow D(f) = a_0 a_4 + 3a_2^2 - 4a_1 a_3 \]

EXAMPLES:

```
sage: R.<a0, a1, a2, a3, a4, x0, x1> = QQ[]
sage: f = a0*x1^4+4*a1*x0*x1^3+6*a2*x0^2*x1^2+4*a3*x0^3*x1+a4*x0^4
sage: inv = invariant_theory.binary_quartic(f, x0, x1)
sage: inv.EisensteinD()
3*a2^2 - 4*a1*a3 + a0*a4
```

EisensteinE()

One of the Eisenstein invariants of a binary quartic.

OUTPUT:

The Eisenstein E-invariant of the quartic.

\[ f(x) = a_0 x_1^4 + 4a_1 x_0 x_1^3 + 6a_2 x_0^2 x_1^2 + 4a_3 x_0^3 x_1 + a_4 x_0^4 \]
\[ \Rightarrow E(f) = a_0 a_3^2 + a_1^2 a_4 - a_0 a_2 a_4 - 2a_1 a_2 a_3 + a_2^3 \]

EXAMPLES:

```
sage: R.<a0, a1, a2, a3, a4, x0, x1> = QQ[]
sage: f = a0*x1^4+4*a1*x0*x1^3+6*a2*x0^2*x1^2+4*a3*x0^3*x1+a4*x0^4
sage: inv = invariant_theory.binary_quartic(f, x0, x1)
sage: inv.EisensteinE()
a2^3 - 2*a1*a2*a3 + a0*a3^2 + a1^2*a4 - a0*a2*a4
```

coeffs()

The coefficients of a binary quartic.

Given

\[ f(x) = a_0 x_1^4 + a_1 x_0 x_1^3 + a_2 x_0^2 x_1^2 + a_3 x_0^3 x_1 + a_4 x_0^4 \]

this function returns \( a = (a_0, a_1, a_2, a_3, a_4) \)

EXAMPLES:
sage: R.<a0, a1, a2, a3, a4, x0, x1> = QQ[]
sage: p = a0*x1^4 + a1*x1^3*x0 + a2*x1^2*x0^2 + a3*x1*x0^3 + a4*x0^4
sage: quartic = invariant_theory.binary_quartic(p, x0, x1)
sage: quartic.coeffs()
(a0, a1, a2, a3, a4)
sage: R.<a0, a1, a2, a3, a4, x> = QQ[]
sage: p = a0 + a1*x + a2*x^2 + a3*x^3 + a4*x^4
sage: quartic = invariant_theory.binary_quartic(p, x)
sage: quartic.coeffs()
(a0, a1, a2, a3, a4)

g_covariant()  
The g-covariant of a binary quartic.  

OUTPUT:  
The g-covariant of the quartic.  

\[ f(x) = a_0x_1^4 + 4a_1x_0x_1^3 + 6a_2x_0^2x_1^2 + 4a_3x_0^3x_1 + a_4x_0^4 \]  
\Rightarrow D(f) = \frac{1}{144} \left( \frac{\partial^2 f}{\partial x \partial x} \right)  

EXAMPLES:  
sage: R.<a0, a1, a2, a3, a4, x, y> = QQ[]
sage: p = a0*x^4+4*a1*x^3*y+6*a2*x^2*y^2+4*a3*x*y^3+a4*y^4
sage: inv = invariant_theory.binary_quartic(p, x, y)
sage: g = inv.g_covariant(); g
-2*a1^3*x^6 + 3*a0*a1*a2*x^6 - a0^2*a3*x^6 - 6*a1^2*a2*x^5*y + 9*a0*a2^2*x^5*y
            + 2*a0*a3^2*x^5*y - 2*a0*a1*a3*x^5*y + 3*a2^2*x^4*y^2 - 2*a1*a2*a3*x^4*y^2
            + 2*a0*a2^2*x^4*y^2 - 2*a1*a2*a3*x^3*y^2 + 2*a0*a3^2*x^3*y^2
            - 2*a1*a4*x^3*y^3 + 3*a2^2*x^2*y^4 - 3*a0*a2*a3*x^2*y^4 - 2*a1*a4*x*y^4

sage: inv_inhomogeneous = invariant_theory.binary_quartic(p.subs(y=1), x)
sage: h = inv_inhomogeneous.g_covariant()
-2*a1^3*x^6 - a0*a2^2*x^6 - 2*a1*a2*a3*x^5*y + 3*a2^2*x^4*y^2 - 2*a1*a2*a3*x^3*y^2
            + 2*a0*a3^2*x^3*y^2 - 2*a1*a4*x^3*y^3 + 3*a2^2*x^2*y^4 - 3*a0*a2*a3*x^2*y^4
            - 2*a1*a4*x*y^4 - 2*a0*a4*x^4*y^2 - 2*a1*a4*x^3*y^2 + 2*a0*a2*a3*x^2*y^2
            + 2*a1*a3^2*x^2*y^2 - 2*a0*a4*x^2*y^2 - 2*a1*a2*a3*x*y^3 + 2*a0*a3^2*x*y^3
            + 3*a2^2*x*y^4 + 2*a1*a4*x*y^4 + 2*a0*a4*x^2*y + 3*a2^2*x*y^2 + 2*a1*a4*x*y + a3^2 - a2*a4

sage: g == 1/144 * (p.derivative(x,y)^2 - p.derivative(x,x)*p.derivative(y,y))
True

h_covariant()  
The h-covariant of a binary quartic.  

OUTPUT:  
The h-covariant of the quartic.  

\[ f(x) = a_0x_1^4 + 4a_1x_0x_1^3 + 6a_2x_0^2x_1^2 + 4a_3x_0^3x_1 + a_4x_0^4 \]  
\Rightarrow D(f) = \frac{1}{144} \left( \frac{\partial^2 f}{\partial x \partial x} \right)  

EXAMPLES:  
sage: R.<a0, a1, a2, a3, a4, x, y> = QQ[]
sage: p = a0*x^4+4*a1*x^3*y+6*a2*x^2*y^2+4*a3*x*y^3+a4*y^4
sage: inv = invariant_theory.binary_quartic(p, x, y)
sage: h = inv.h_covariant(); h
-2*a1^3*x^6 + 3*a0*a1*a2*x^6 - a0^2*a3*x^6 - 6*a1^2*a2*x^5*y + 9*a0*a2^2*x^5*y
            + 2*a0*a3^2*x^5*y - 2*a0*a1*a3*x^5*y + 3*a2^2*x^4*y^2 - 2*a1*a2*a3*x^4*y^2
            + 2*a0*a2^2*x^4*y^2 - 2*a1*a2*a3*x^3*y^2 + 2*a0*a3^2*x^3*y^2
            - 2*a1*a4*x^3*y^3 + 3*a2^2*x^2*y^4 - 3*a0*a2*a3*x^2*y^4 - 2*a1*a4*x*y^4
            - 2*a0*a4*x^4*y^2 - 2*a1*a4*x^3*y^2 + 2*a0*a3^2*x^3*y^2
            + 2*a1*a4*x^2*y^3 + 2*a0*a3^2*x^2*y^3 - 2*a1*a4*x*y^3 + 2*a0*a3^2*x*y^3
            + 3*a2^2*x*y^4 + 2*a1*a4*x*y^4 + 2*a0*a4*x^2*y + 3*a2^2*x*y^2 + 2*a1*a4*x*y + a3^2 - a2*a4

sage: inv_inhomogeneous = invariant_theory.binary_quartic(p.subs(y=1), x)
sage: h = inv_inhomogeneous.h_covariant()
-2*a1^3*x^6 - a0*a2^2*x^6 - 2*a1*a2*a3*x^5*y + 3*a2^2*x^4*y^2 - 2*a1*a2*a3*x^3*y^2
            + 2*a0*a3^2*x^3*y^2 - 2*a1*a4*x^3*y^3 + 3*a2^2*x^2*y^4 - 3*a0*a2*a3*x^2*y^4
            - 2*a1*a4*x*y^4 - 2*a0*a4*x^4*y^2 - 2*a1*a4*x^3*y^2 + 2*a0*a3^2*x^3*y^2
            + 2*a1*a4*x^2*y^3 + 2*a0*a3^2*x^2*y^3 - 2*a1*a4*x*y^3 + 2*a0*a3^2*x*y^3
            + 3*a2^2*x*y^4 + 2*a1*a4*x*y^4 + 2*a0*a4*x^2*y + 3*a2^2*x*y^2 + 2*a1*a4*x*y + a3^2 - a2*a4

sage: g == 1/144 * (p.derivative(x,y)^2 - p.derivative(x,x)*p.derivative(y,y))
True
- 2*a0*a1*a3*x^5*y - a0^2*a4*x^5*y - 10*a1^2*a3*x^4*y^2 + 15*a0*a2*a3*x^4*y^2 + 10*a1*a3^2*x^2*y^4 + 5*a0*a3*a4*x^2*y^4 + 6*a2*a3^2*x*y^5 - 9*a2^2*a4*x*y^5 + 2*a1*a3*a4*x*y^5 + a0*a4^2*x*y^5 + 2*a3^3*y^6 - 3*a2*a3*a4*y^6 + a1*a4^2*y^6

```
sage: inv_inhomogeneous = invariant_theory.binary_quartic(p.subs(y=1), x)
sage: inv_inhomogeneous.h_covariant()
-2*a1^3*x^6 + 3*a0*a1*a2*x^6 - a0^2*a3*x^6 - 6*a1^2*a2*x^5 + 9*a0*a2^2*x^5
- 2*a0*a1*a3*x^5 - a0^2*a4*x^5 - 10*a1^2*a3*x^4 + 15*a0*a2*a3*x^4
- 5*a0*a1*a4*x^4 + 10*a0*a3^2*x^3 - 10*a1^2*a4*x^3 + 10*a1*a3^2*x^2
- 15*a1*a2*a4*x^2 + 5*a0*a3*a4*x^2 - 6*a2*a3^2*x - 9*a2^2*a4*x
+ 2*a1*a3*a4*x + a0*a4^2*x + 2*a3^3 - 3*a2*a3*a4 + a1*a4^2
```

```
sage: g = inv.g_covariant()
sage: h == 1/8 * (p.derivative(x)*g.derivative(y)-p.derivative(y)*g.derivative(x))
True
```

### monomials()

List the basis monomials in the form.

**OUTPUT:**

A tuple of monomials. They are in the same order as `coeffs()`.

**EXAMPLES:**

```
sage: R.<x,y> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+y^4)
sage: quartic.monomials()
(y^4, x*y^3, x^2*y^2, x^3*y, x^4)
```

### scaled_coeffs()

The coefficients of a binary quartic.

Given

\[ f(x) = a_0 x_0^4 + 4a_1 x_0 x_1^3 + 6a_2 x_0^2 x_1^2 + 4a_3 x_0^3 x_1 + a_4 x_0^4 \]

this function returns \(\mathbf{a} = (a_0, a_1, a_2, a_3, a_4)\).

**EXAMPLES:**

```
sage: R.<a0, a1, a2, a3, a4, x0, x1> = QQ[]
sage: quartic = a0*x1^4 + 4*a1*x0*x1^3 + 6*a2*x0^2*x1^2 + 4*a3*x0^3*x1 + a4*x0^4
sage: inv = invariant_theory.binary_quartic(quartic, x0, x1)
sage: inv.scaled_coeffs()
(a0, a1, a2, a3, a4)
sage: R.<a0, a1, a2, a3, a4, x> = QQ[]
sage: quartic = a0 + 4*a1*x + 6*a2*x^2 + 4*a3*x^3 + a4*x^4
sage: inv = invariant_theory.binary_quartic(quartic, x)
sage: inv.scaled_coeffs()
(a0, a1, a2, a3, a4)
```

### class sage.rings.invariants.invariant_theory.BinaryQuintic(n, d, polynomial, *args)

**Bases:** `sage.rings.invariants.invariant_theory.AlgebraicForm`
Invariant theory of a binary quintic form.

You should use the \texttt{invariant\_theory} factory object to construct instances of this class. See \texttt{binary\_quintic()} for details.

REFERENCES:

For a description of all invariants and covariants of a binary quintic, see section 73 of [Cle1872].

\texttt{A\_invariant()} 

Return the invariant $A$ of a binary quintic.

OUTPUT:

The $A$-invariant of the binary quintic.

EXAMPLES:

```sage
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant\_theory.binary\_quintic(p, x0, x1)
sage: quintic.A\_invariant()
4/625*a2^2*a3^2 - 12/625*a1*a3^3 - 12/625*a2^3*a4 + 38/625*a1*a2*a3*a4 + 6/125*a0*a3^2*a4 - 18/625*a1^2*a4^2 - 16/125*a0*a2*a4^2 + 6/125*a1*a2^2*a5 - 16/125*a1^2*a3*a5 - 2/25*a0*a2*a3*a5 + 4/5*a0*a1*a4*a5 - 2*a0^2*a5^2
```

\texttt{B\_invariant()} 

Return the invariant $B$ of a binary quintic.

OUTPUT:

The $B$-invariant of the binary quintic.

EXAMPLES:

```sage
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant\_theory.binary\_quintic(p, x0, x1)
sage: quintic.B\_invariant()
1/1562500*a2^4*a3^4 - 3/781250*a1*a2^2*a3^5 + 9/1562500*a1^2*a3^6 - 3/781250*a2^5*a3^2*a4 + 37/1562500*a1*a2^3*a3^3*a4 - 57/1562500*a1^2*a2*a3^4*a4 + 3/312500*a0*a2^2*a3^4*a4 + ...
```

\texttt{C\_invariant()} 

Return the invariant $C$ of a binary quintic.

OUTPUT:

The $C$-invariant of the binary quintic.

EXAMPLES:

```sage
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant\_theory.binary\_quintic(p, x0, x1)
sage: quintic.C\_invariant()
```

(continues on next page)
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.C_invariant()
-3/1953125000*a2^6*a3^6 + 27/1953125000*a1*a2^4*a3^7
- 249/7812500000*a1^2*a2^2*a3^8 - 3/781250000*a0*a2^3*a3^8
+ 3/9765625000*a1^3*a3^9 + 27/1562500000*a0*a1*a2^3*a3^9
...
+ 192/15625*a0^2*a1^3*a2^2*a3^5 + 36/3125*a0^3*a1^2*a2^3*a3^5 + 24/3125*a0^3*a1^2*a2^3*a3^5
+ 6/625*a0^2*a1^2*a2^2*a3^5

\(H\)_covariant \((\text{as_form}=\text{False})\)

Return the covariant \(H\) of a binary quintic.

**INPUT:**

- \(\text{as_form}\) – if \(\text{as_form}\) is \text{False}, the result will be returned as polynomial (default). If it is \text{True} the result is returned as an object of the class \text{AlgebraicForm}.

**OUTPUT:**

The \(H\)-covariant of the binary quintic as polynomial or as binary form.

**EXAMPLES:**

```
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + ...
   + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.H_covariant()
-2/25*a4^2*x0^6 + 1/5*a3*a5*x0^6 - 3/25*a3*a4*x0^5*x1
+ 3/5*a2*a5*x0^5*x1 - 3/25*a3^2*x0^4*x1^2 + 3/25*a2*a4*x0^4*x1^2
+ 6/5*a1*a5*x0^4*x1^2 - 4/25*a2*a3*x0^3*x1^3 + 14/25*a1*a4*x0^3*x1^3
+ 2*a0*a5*x0^3*x1^3 - 3/25*a2*a3*x0^2*x1^4 + 3/25*a1*a4*x0^2*x1^4
+ 6/5*a0*a4*x0^2*x1^4 - 3/25*a1*a2*x0^2*x1^5 + 3/5*a0*a3*x0*x1^5
- 2/25*a1^2*x1^6 + 1/5*a0*a2*x1^6

sage: quintic.H_covariant(as_form=True)
Binary sextic given by ...
```

\(R\)_invariant ()

Return the invariant \(R\) of a binary quintic.

**OUTPUT:**

The \(R\)-invariant of the binary quintic.

**EXAMPLES:**

```
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + ...
   + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.R_invariant()
3/3906250000000*a1^2*a2^5*a3^10 - 3/976562500000*a0*a2^6*a3^11
- 51/781250000000*a1^3*a2^3*a3^12 + 27/976562500000*a0*a1*a2^4*a3^12
+ 27/1953125000000*a1^4*a2^2*a3^13 - 81/1562500000000*a0*a1*a2^2*a2^2*a3^13
...
+ 384/9765625*a0*a1^10*a5^7 - 192/390625*a0^2*a1^8*a2*a5^7
+ 192/78125*a0^3*a1^6*a2^2*a5^7 - 96/15625*a0^4*a1^4*a2^3*a5^7
+ 24/3125*a0^5*a1^2*a2^4*a5^7 - 12/3125*a0^6*a2^5*a5^7
```

416 Chapter 3. Multivariate Polynomials
\textbf{T\_covariant (as\_form=False)}

Return the covariant $T$ of a binary quintic.

\textbf{INPUT:}

- \textit{as\_form} – if \textit{as\_form} is False, the result will be returned as polynomial (default). If it is True the result is returned as an object of the class \texttt{AlgebraicForm}.

\textbf{OUTPUT:}

The $T$-covariant of the binary quintic as polynomial or as binary form.

\textbf{EXAMPLES:}

```python
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.T_covariant()
2/125*a4^3*x0^9 - 3/50*a3*a4*a5*x0^9 + 1/10*a2*a5^2*x0^9
+ 9/250*a3^2*a5*x0^8*x1 + 1/50*a2*a4*a5*x0^8*x1
+ 2/25*a1*a5^2*x0^7*x1 + 1/50*a2*a4^2*x0^7*x1 + 8/125*a2*a4^2*x0^7*x1^2
... + 11/25*a0*a1*a4*x0^2*x1^7 - a0^2*a5*x0^2*x1^7 - 9/250*a1^2*a2*x0*x1^8
+ 3/25*a0*a2^2*x0*x1^8 - 1/50*a0*a1*a3*x0*x1^8 - 2/5*a0*a2*a4*x0*x1^8
- 2/125*a1^3*x1^9 + 3/50*a0*a1*a2*x1^9 - 1/10*a0^2*a3*x1^9
```

```python
sage: quintic.T_covariant(as\_form=True)
Binary nonic given by ...
```

\textbf{alpha\_covariant (as\_form=False)}

Return the covariant $\alpha$ of a binary quintic.

\textbf{INPUT:}

- \textit{as\_form} – if \textit{as\_form} is False, the result will be returned as polynomial (default). If it is True the result is returned as an object of the class \texttt{AlgebraicForm}.

\textbf{OUTPUT:}

The $\alpha$-covariant of the binary quintic as polynomial or as binary form.

\textbf{EXAMPLES:}

```python
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.alpha_covariant()
1/2500*a2^2*a3^3*x0 - 3/2500*a1*a3^4*x0 + 1/625*a2^3*a3*a4*x0
+ 3/625*a1*a2*a3^2*a4*x0 + 3/625*a0*a3^3*a4*x0 + 2/625*a1*a2^2*a4^2*x0
- 6/625*a1^2*a3*a4^2*x0 - 12/625*a0*a2*a3*a4^2*x0 + 24/625*a0*a1*a4^3*x0
... - 12/625*a1^2*a2*a3*a5*x1 - 1/125*a0*a3^2*a5*x1
+ 8/125*a0*a1*a3^2*a5*x1 + 24/625*a1^3*a4*a5*x1 - 8/125*a0*a1*a2*a4*a5*x1
- 4/25*a0^2*a3*a4*a5*x1 - 4/25*a0*a1^2*a5^2*x1 + 2/5*a0^2*a2*a5^2*x1
```

```python
sage: quintic.alpha_covariant(as\_form=True)
Binary monic given by ...
```

\textbf{arithmetic\_invariants()}

Return a set of generating arithmetic invariants of a binary quintic.

3.2. Classical Invariant Theory 417
An arithmetic invariant is an invariant whose coefficients are integers for a general binary quintic. They are linear combinations of the Clebsch invariants, such that they still generate the ring of invariants.

**OUTPUT:**

The arithmetic invariants of the binary quintic. They are given by

\[
\begin{align*}
I_4 &= 2^{-1} \cdot 5^4 \cdot A \\
I_8 &= 5^5 \cdot (2^{-1} \cdot 47 \cdot A^2 - 2 \cdot B) \\
I_{12} &= 5^{10} \cdot (2^{-1} \cdot 3 \cdot A^3 - 2^5 \cdot 3^{-1} \cdot C) \\
I_{18} &= 2^8 \cdot 3^{-1} \cdot 5^{15} \cdot R
\end{align*}
\]

where \( A, B, C \) and \( R \) are the \texttt{BinaryQuintic.clebsch_invariants()}.

**EXAMPLES:**

```python
sage: R.<x0, x1> = QQ[]
sage: p = 2*x1^5 + 4*x1^4*x0 + 5*x1^3*x0^2 + 7*x1^2*x0^3 - 11*x1*x0^4 + x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.arithmetic_invariants()
{'I12': -1156502613073152,
 'I18': -12712872348048797642752,
 'I4': -138016,
 'I8': 14164936192}
```

We can check that the coefficients of the invariants have no common divisor for a general quintic form:

```python
sage: R.<a0,a1,a2,a3,a4,a5,x0,x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: invs = quintic.arithmetic_invariants()
sage: [invs[x].content() for x in invs]
[1, 1, 1, 1]
```

**beta_covariant** \((\text{as_form}=\text{False})\)

Return the covariant \( \beta \) of a binary quintic.

**INPUT:**

\* as_form – if as_form is \text{False}, the result will be returned as polynomial (default). If it is \text{True} the result is returned as an object of the class \texttt{AlgebraicForm}.

**OUTPUT:**

The \( \beta \)-covariant of the binary quintic as polynomial or as binary form.

**EXAMPLES:**

```python
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.beta_covariant()
-1/62500*a2^3*a3^4*x0 + 9/125000*a1*a2*a3^5*x0 - 27/125000*a0*a3^6*x0
+ 13/125000*a2^4*a3^2*a4*x0 - 31/62500*a1*a2^2*a3^3*a4*x0
- 3/62500*a1^2*a3^4*a4*x0 + 27/15625*a0*a2*a3^4*a4*x0
...
- 16/125*a0^2*a1*a3^2*a5^2*x1 - 28/625*a0*a1^3*a4*a5^2*x1
```

(continues on next page)
\[ + \frac{6}{125}a_0^2a_1a_2a_4a_5^2x_1 + \frac{8}{25}a_0^3a_3a_4a_5^2x_1 \\
+ \frac{4}{25}a_0^2a_1^2a_5^3x_1 - \frac{2}{5}a_0^3a_2a_5^3x_1 \]

```
sage: quintic.beta_covariant(as_form=True)
Binary monic given by ...
```

**canonical_form**(*reduce_gcd=False*)

Return a canonical representative of the quintic.

Given a binary quintic \( f \) with coefficients in a field \( K \), returns a canonical representative of the \( GL(2, \bar{K}) \)-orbit of the quintic, where \( \bar{K} \) is an algebraic closure of \( K \). This means that two binary quintics \( f \) and \( g \) are \( GL(2, \bar{K}) \)-equivalent if and only if their canonical forms are the same.

**INPUT:**

- \( \text{reduce}_\text{gcd} \) – If set to \( \text{True} \), then a variant of this canonical form is computed where the coefficients are coprime integers. The obtained form is then unique up to multiplication by a unit. See also `binary_quintic_from_invariants()`.

**OUTPUT:**

A canonical \( GL(2, \bar{K}) \)-equivalent binary quintic.

**EXAMPLES:**

```
sage: R.<x0, x1> = QQ[]
sage: p = 2*x1^5 + 4*x1^4*x0 + 5*x1^3*x0^2 + 7*x1^2*x0^3 - 11*x1*x0^4 + x0^5
sage: f = invariant_theory.binary_quintic(p, x0, x1)
sage: g = matrix(QQ, [[[11, 5], [7, 2]])
sage: gf = f.transformed(g)
sage: f.canonical_form() == gf.canonical_form()
True
sage: h = f.canonical_form(reduce_gcd=True)
sage: gcd(h.coeffs())
1
```

**clebsch_invariants**(*as_tuple=False*)

Return the invariants of a binary quintic as described by Clebsch.

The following invariants are returned: \( A, B, C \) and \( R \).

**OUTPUT:**

The Clebsch invariants of the binary quintic.

**EXAMPLES:**

```
sage: R.<x0, x1> = QQ[]
sage: p = 2*x1^5 + 4*x1^4*x0 + 5*x1^3*x0^2 + 7*x1^2*x0^3 - 11*x1*x0^4 + x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.clebsch_invariants()
\{'A': -276032/625, 
'B': 4983526016/390625, 
'C': -24705649584608/244140625, 
'R': -148978972828696847376/30517578125}
sage: quintic.clebsch_invariants(as_tuple=True)
(-276032/625, 
4983526016/390625, 
4983526016/390625, 
4983526016/390625)
```

(continues on next page)
coeffs()

The coefficients of a binary quintic.

Given

\[ f(x) = a_0 x^5 + a_1 x_0 x_1^4 + a_2 x_0^2 x_1^3 + a_3 x_0^3 x_1^2 + a_4 x_0^4 x_1 + a_5 x_1^5 \]

this function returns \( a = (a_0, a_1, a_2, a_3, a_4, a_5) \)

EXAMPLES:

```
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.coeffs()
(a0, a1, a2, a3, a4, a5)
sage: R.<a0, a1, a2, a3, a4, a5, x> = QQ[]
sage: p = a0 + a1*x + a2*x^2 + a3*x^3 + a4*x^4 + a5*x^5
sage: quintic = invariant_theory.binary_quintic(p, x)
sage: quintic.coeffs()
(a0, a1, a2, a3, a4, a5)
```

delta_covariant (as_form=False)

Return the covariant \( \delta \) of a binary quintic.

INPUT:

- as_form -- if as_form is False, the result will be returned as polynomial (default). If it is True the result is returned as an object of the class `AlgebraicForm`.

OUTPUT:

The \( \delta \)-covariant of the binary quintic as polynomial or as binary form.

EXAMPLES:

```
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.delta_covariant()
1/1562500000*a2^6*a3^7*x0 - 9/1562500000*a1*a2^4*a3^8*x0 + 9/6250000000*a1^2*a2^2*a3^9*x0 + 9/7812500000*a0*a2^3*a3^9*x0 - 9/1562500000*a1^3*a3^10*x0 - 81/1562500000*a0*a1*a2*a3^10*x0 + 64/3125*a0^3*a1^2*a2^2*a3^5*x0 - 12/625*a0^4*a1*a2^2*a3^5*x0 + 16/3125*a0^5*a1^4*a3*a5^5*x0 - 16/625*a0^4*a1^2*a2*a3*a5^5*x0 + 4/125*a0^5*a2^2*a3*a5^5*x0
sage: quintic.delta_covariant(as_form=True)
Binary monic given by ...
```

classmethod from_invariants (invariants, x, z, *args, **kwargs)

Construct a binary quintic from its invariants.
This function constructs a binary quintic whose invariants equal the ones provided as argument up to scaling.

**INPUT:**
- invariants – A list or tuple of invariants that are used to reconstruct the binary quintic.

**OUTPUT:**
A BinaryQuintic.

**EXAMPLES:**

```python
sage: R.<x,y> = QQ[]
sage: from sage.rings.invariants.invariant_theory import BinaryQuintic
sage: BinaryQuintic.from_invariants([3,6,12], x, y)
Binary quintic with coefficients (0, 1, 0, 0, 1, 0)
```

### gamma_covariant (as_form=False)

Return the covariant \( \gamma \) of a binary quintic.

**INPUT:**
- as_form – if as_form is False, the result will be returned as polynomial (default). If it is True the result is returned as an object of the class `AlgebraicForm`.

**OUTPUT:**
The \( \gamma \)-covariant of the binary quintic as polynomial or as binary form.

**EXAMPLES:**

```python
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.gamma_covariant()
1/156250000*a2^5*a3^6*x0 - 3/62500000*a1*a2^3*a3^7*x0 + 27/312500000*a1^2*a2*a3^8*x0 - 81/312500000*a0*a2^2*a3^9*x0 - 19/312500000*a2^6*a3^4*a4*x0 - ...
```

### i_covariant (as_form=False)

Return the covariant \( i \) of a binary quintic.

**INPUT:**
- as_form – if as_form is False, the result will be returned as polynomial (default). If it is True the result is returned as an object of the class `AlgebraicForm`.

**OUTPUT:**
The \( i \)-covariant of the binary quintic as polynomial or as binary form.

**EXAMPLES:**

```python
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.i_covariant(as_form=True)
Binary monic given by ...
```
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]  
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5  
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)  
sage: quintic.i_covariant()  
\frac{3}{50}a_3^2x_0^2 - \frac{4}{25}a_2a_4x_0^2 + \frac{2}{5}a_1a_5x_0^2 + \frac{1}{25}a_2a_3x_0x_1 - \frac{6}{25}a_1a_4x_0x_1 + 2a_0a_5x_0x_1 + \frac{3}{50}a_2^2x_1^2 - \frac{4}{25}a_1a_3x_1^2 + \frac{2}{5}a_0a_4x_1^2

sage: quintic.i_covariant(as_form=True)  
Binary quadratic given by ...

invariants (type='clebsch')
Return a tuple of invariants of a binary quintic.

INPUT:

- type – The type of invariants to return. The default choice is to return the Clebsch invariants.

OUTPUT:

The invariants of the binary quintic.

EXAMPLES:

sage: R.<x0, x1> = QQ[]  
sage: p = 2*x1^5 + 4*x1^4*x0 + 5*x1^3*x0^2 + 7*x1^2*x0^3 - 11*x1*x0^4 + x0^5  
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)  
sage: quintic.invariants()  
(-276032/625, 4983526016/390625, -247056495846408/244140625, -148978972828696847376/30517578125)  
sage: quintic.invariants('unknown')  
Traceback (most recent call last):  
  ...  
ValueError: unknown type of invariants unknown for a binary quintic

j_covariant (as_form=False)
Return the covariant $j$ of a binary quintic.

INPUT:

- as_form – if as_form is False, the result will be returned as polynomial (default). If it is True the result is returned as an object of the class AlgebraicForm.

OUTPUT:

The $j$-covariant of the binary quintic as polynomial or as binary form.

EXAMPLES:

sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]  
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5  
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)  
sage: quintic.j_covariant()  
-\frac{3}{500}a_3^3x_0^3 + \frac{3}{125}a_2a_3a_4x_0^3 - \frac{6}{125}a_1a_4^2x_0^3 - \frac{3}{50}a_2^2a_5x_0^2x_1  
+ \frac{3}{25}a_2a_4x_0^2x_1 + \frac{3}{125}a_1a_3a_4x_0^2x_1 - \frac{6}{25}a_0a_4^2x_0^2x_1  
- \frac{3}{25}a_1a_2a_5x_0^2x_1 + \frac{3}{5}a_0a_3a_5x_0^2x_1 - \frac{3}{50}a_2^2a_3x_0x_1^2
+ 3/250*a1*a3^2*x0*x1^2 + 3/125*a1*a2*a4*x0*x1^2 - 3/25*a0*a3*a4*x0*x1^2
- 6/25*a1^2*a5*x0*x1^2 + 3/5*a0*a2*a5*x0*x1^2 - 3/500*a2^3*x1^3
+ 3/125*a1*a2*a3*x1^3 - 3/50*a0*a3^2*x1^3 - 6/125*a1^2*a4*x1^3
+ 3/25*a0*a2*a4*x1^3

sage: quintic.j_covariant(as_form=True)
Binary cubic given by ...

monomials()
List the basis monomials of the form.
This function lists a basis of monomials of the space of binary quintics of which this form is an element.

OUTPUT:
A tuple of monomials. They are in the same order as coeffs().

EXAMPLES:

sage: R.<x,y> = QQ[]
sage: quintic = invariant_theory.binary_quintic(x^5+y^5)
sage: quintic.monomials()
(y^5, x*y^4, x^2*y^3, x^3*y^2, x^4*y, x^5)

scaled_coeffs()
The coefficients of a binary quintic.

Given
\[ f(x) = a_0 x_1^5 + 5 a_1 x_0 x_1^4 + 10 a_2 x_0^2 x_1^3 + 10 a_3 x_0^3 x_1^2 + 5 a_4 x_0^4 x_1 + a_5 x_0^5 \]
this function returns \( a = (a_0, a_1, a_2, a_3, a_4, a_5) \)

EXAMPLES:

sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + 5*a1*x0*x1^4 + 10*a2*x0^2*x1^3 + 10*a3*x0^3*x1^2 + 5*a4*x0^4*x1 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.scaled_coeffs()
(a0, a1, a2, a3, a4, a5)

tau_covariant (as_form=False)
Return the covariant \( \tau \) of a binary quintic.

INPUT:
- \( \text{as_form} \) – if \( \text{as_form} \) is False, the result will be returned as polynomial (default). If it is True the result is returned as an object of the class AlgebraicForm.

OUTPUT:
The \( \tau \)-covariant of the binary quintic as polynomial or as binary form.

EXAMPLES:

```
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 +
   a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.tau_covariant()
1/62500*a2^2*a3^4*x0^2 - 3/62500*a1*a3^5*x0^2
+ 3/6250*a0*a3^4*a4*x0^2 - 1/31250*a2^4*a4^2*x0^2

sage: quintic.tau_covariant(as_form=True)
Binary quadratic given by ...
```

**theta_covariant** *(as_form=False)*

Return the covariant \( \theta \) of a binary quintic.

**INPUT:**

- `as_form` -- if `as_form` is False, the result will be returned as polynomial (default). If it is True the result is returned as an object of the class `AlgebraicForm`.

**OUTPUT:**

The \( \theta \)-covariant of the binary quintic as polynomial or as binary form.

**EXAMPLES:**

```
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 +
   a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.theta_covariant()
-1/625000*a2^3*a3^5*x0^2 + 9/1250000*a1*a2*a3^6*x0^2
+ 27/1250000*a0*a3^7*x0^2 + 3/250000*a2^4*a3^3*a4*x0^2
- 7/1250000*a1*a2^2*a3^4*a4*x0^2 - 3/312500*a1^2*a3^5*a4*x0^2

sage: quintic.theta_covariant(as_form=True)
Binary quadratic given by ...
```

class sage.rings.invariants.invariant_theory.FormsBase(n, homogeneous, ring, variables)

Bases: sage.structure.sage_object.SageObject

The common base class of `AlgebraicForm` and `SeveralAlgebraicForms`.

This is an abstract base class to provide common methods. It does not make much sense to instantiate it.

**is_homogeneous()**

Return whether the forms were defined by homogeneous polynomials.

**OUTPUT:**

Boolean. Whether the user originally defined the form via homogeneous variables.

**EXAMPLES:**
sage: R.<x,y,t> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+y^4+t*x^2*y^2, [x,y])
sage: quartic.is_homogeneous()
True
sage: quartic.form()
x^2*y^2*t + x^4 + y^4
sage: R.<x,y,t> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+1+t*x^2, [x])
sage: quartic.is_homogeneous()
False
sage: quartic.form()
x^4 + x^2*t + 1

ring()
Return the polynomial ring.

OUTPUT:
A polynomial ring. This is where the defining polynomial(s) live. Note that the polynomials may be
homogeneous or inhomogeneous, depending on how the user constructed the object.

EXAMPLES:

sage: R.<x,y,t> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+y^4+t*x^2*y^2, [x,y])
sage: quartic.ring()
Multivariate Polynomial Ring in x, y, t over Rational Field
sage: R.<x,y,t> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+1+t*x^2, [x])
sage: quartic.ring()
Multivariate Polynomial Ring in x, y, t over Rational Field

variables()
Return the variables of the form.

OUTPUT:
A tuple of variables. If inhomogeneous notation is used for the defining polynomial then the last entry will
be None.

EXAMPLES:

sage: R.<x,y,t> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+y^4+t*x^2*y^2, [x,y])
sage: quartic.variables()
(x, y)
sage: R.<x,y,t> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+1+t*x^2, [x])
sage: quartic.variables()
(x, None)

class sage.rings.invariants.invariant_theory.InvariantTheoryFactory
Bases: object

Factory object for invariants of multilinear forms.
Use the invariant_theory object to construct algebraic forms. These can then be queried for invariant and covari-

EXAMPLES:

```python
sage: R.<x,y,z> = QQ[]
sage: invariant_theory.ternary_cubic(x^3+y^3+z^3)
Ternary cubic with coefficients (1, 1, 1, 0, 0, 0, 0, 0, 0, 0)
sage: invariant_theory.ternary_cubic(x^3+y^3+z^3).J_covariant()
x^6*y^3 - x^3*y^6 - x^6*z^3 + y^6*z^3 + x^3*z^6 - y^3*z^6
```

`binary_form_from_invariants(degree, invariants, variables=None, as_form=True, *args, **kwargs)`

Reconstruct a binary form from the values of its invariants.

**INPUT:**

- `degree` – The degree of the binary form.
- `invariants` – A list or tuple of values of the invariants of the binary form.
- `variables` – A list or tuple of two variables that are used for the resulting form (only if `as_form` is True). If no variables are provided, two abstract variables x and z will be used.
- `as_form` – boolean. If False, the function will return a tuple of coefficients of a binary form.

**OUTPUT:**

A binary form or a tuple of its coefficients, whose invariants are equal to the given invariants up to a scaling.

**EXAMPLES:**

In the case of binary quadratics and cubics, the form is reconstructed based on the value of the discriminant. See also `binary_quadratic_coefficients_from_invariants()` and `binary_cubic_coefficients_from_invariants()`. These methods will always return the same result if the discriminant is non-zero:

```python
sage: discriminant = 1
sage: invariant_theory.binary_form_from_invariants(2, [discriminant])
Binary quadratic with coefficients (1, -1/4, 0)
sage: invariant_theory.binary_form_from_invariants(3, [discriminant], as_form=False)
(0, 1, -1, 0)
```

For binary cubics, there is no class implemented yet, so `as_form=True` will yield an
`NotImplementedError`:

```python
sage: invariant_theory.binary_form_from_invariants(3, [discriminant])
Traceback (most recent call last):
  ...  
NotImplementedError: no class for binary cubics implemented
```

For binary quintics, the three Clebsch invariants of the form should be provided to reconstruct the form. For more details about these invariants, see `clebsch_invariants()`:

```python
sage: invariants = [1, 0, 0]
sage: invariant_theory.binary_form_from_invariants(5, invariants)
Binary quintic with coefficients (1, 0, 0, 0, 0, 1)
```
An optional scaling argument may be provided in order to scale the resulting quintic. For more details, see `binary_quintic_coefficients_from_invariants()`:

```
sage: invariants = [3, 4, 7]
sage: invariant_theory.binary_form_from_invariants(5, invariants)
Binary quintic with coefficients (-37725479487783/1048576, 565882192316745/8388608, 0, 1033866765362693115/67108864, 128494869409328715/268435456, -2312907649368591687/2147483648)
sage: invariant_theory.binary_form_from_invariants(5, invariants, scaling=˓→'normalized')
Binary quintic with coefficients (24389/892616806656, 4205/11019960576, 0, 1015/209952, -145/1296, -3/16)
sage: invariant_theory.binary_form_from_invariants(5, invariants, scaling=˓→'coprime')
Binary quintic with coefficients (-2048, 3840, 0, 876960, 2724840, -613089)
```

The invariants can also be computed using the invariants of a given binary quintic. The resulting form has the same invariants up to scaling, is $GL(2, \mathbb{Q})$-equivalent to the provided form and hence has the same canonical form (see `canonical_form()`):

```
sage: R.<x0, x1> = QQ[]
sage: p = 3*x1^5 + 6*x1^4*x0 + 3*x1^3*x0^2 + 4*x1^2*x0^3 - 5*x1*x0^4 + 4*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: invariants = quintic.clebsch_invariants(as_tuple=True)
sage: newquintic = invariant_theory.binary_form_from_invariants(5, invariants, ˓→variables=quintic.variables())
sage: newquintic
Binary quintic with coefficients (9592267437341790539005557/244140625000000, 214922692220762556323004064707/6103515625000000, 11149651890347700974453304786783/7629394531250000, 12265077571894638395648891202734239/47683715820312500000, 323996630945706528474286334593218447/119209289550781250000, 1504506503644608395841632538558481466127/14901161193847656250000)
sage: quintic.canonical_form() == newquintic.canonical_form()
True
```

For binary forms of other degrees, no reconstruction has been implemented yet. For forms of degree 6, see trac ticket #26462:

```
sage: invariant_theory.binary_form_from_invariants(6, invariants)
Traceback (most recent call last): ...
NotImplementedError: no reconstruction for binary forms of degree 6...
```

### `binary_quadratic`(*quadratic*, *args)

Invariant theory of a quadratic in two variables.

**INPUT:**
- `quadratic` – a quadratic form.
- `x, y` – the homogeneous variables. If `y` is `None`, the quadratic is assumed to be inhomogeneous.

**REFERENCES:**

**EXAMPLES:**

3.2. Classical Invariant Theory 427
binary_quartic(quartic, *args, **kwds)

Invariant theory of a quartic in two variables.

The algebra of invariants of a quartic form is generated by invariants \(i, j\) of degrees 2, 3. This ring is naturally isomorphic to the ring of modular forms of level 1, with the two generators corresponding to the Eisenstein series \(E_4\) (see \(\text{EisensteinD()}\)) and \(E_6\) (see \(\text{EisensteinE()}\)). The algebra of covariants is generated by these two invariants together with the form \(f\) of degree 1 and order 4, the Hessian \(g\) (see \(\text{g_covariant()}\)) of degree 2 and order 4, and a covariant \(h\) (see \(\text{h_covariant()}\)) of degree 3 and order 6. They are related by a syzygy

\[
jf^3 - gf^2i + 4g^3 + h^2 = 0
\]

of degree 6 and order 12.

INPUT:

- \(\text{quartic}\) – a quartic.
- \(x, y\) – the homogeneous variables. If \(y\) is None, the quartic is assumed to be inhomogeneous.

REFERENCES:

- Wikipedia article Invariant_of_a_binary_form

EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+y^4)
sage: quartic
Binary quartic with coefficients (1, 0, 0, 0, 1)
sage: type(quartic)
<class 'sage.rings.invariants.invariant_theory.BinaryQuartic'>
```

binary_quintic(quintic, *args, **kwds)

Create a binary quintic for computing invariants.

A binary quintic is a homogeneous polynomial of degree 5 in two variables. The algebra of invariants of a binary quintic is generated by the invariants \(A, B\) and \(C\) of respective degrees 4, 8 and 12 (see \(\text{A_invariant()}\), \(\text{B_invariant()}\) and \(\text{C_invariant()}\)).

INPUT:

- \(\text{quintic}\) – a homogeneous polynomial of degree five in two variables or a (possibly inhomogeneous) polynomial of degree at most five in one variable.
- \(*\text{args}\) – the two homogeneous variables. If only one variable is given, the polynomial \(\text{quintic}\) is assumed to be univariate. If no variables are given, they are guessed.

REFERENCES:

- Wikipedia article Invariant_of_a_binary_form
- [Cle1872]
EXAMPLES:

If no variables are provided, they will be guessed:

```python
sage: R.<x,y> = QQ[]
sage: quintic = invariant_theory.binary_quintic(x^5+y^5)
sage: quintic
Binary quintic with coefficients (1, 0, 0, 0, 0, 1)
```

If only one variable is given, the quintic is the homogenisation of the provided polynomial:

```python
sage: quintic = invariant_theory.binary_quintic(x^5+y^5, x)
sage: quintic
Binary quintic with coefficients (y^5, 0, 0, 0, 0, 1)
sage: quintic.is_homogeneous()
False
```

If the polynomial has three or more variables, the variables should be specified:

```python
sage: R.<x,y,z> = QQ[]
sage: quintic = invariant_theory.binary_quintic(x^5+z*y^5)
Traceback (most recent call last):
  ...  ValueError: need 2 or 1 variables, got (x, y, z)
sage: quintic = invariant_theory.binary_quintic(x^5+z*y^5, x, y)
sage: quintic
Binary quintic with coefficients (z, 0, 0, 0, 0, 1)
sage: type(quintic)
<class 'sage.rings.invariants.invariant_theory.BinaryQuintic'>
```

**inhomogeneous_quadratic_form**(*polynomial*, *args*)

Invariants of an inhomogeneous quadratic form.

**INPUT:**

- *polynomial* – an inhomogeneous quadratic form.
- *args* – the variables as multiple arguments, or as a single list/tuple.

**EXAMPLES:**

```python
sage: R.<x,y,z> = QQ[]
sage: quadratic = x^2+2*y^2+3*x*y+4*x+5*y+6
sage: inv3 = invariant_theory.inhomogeneous_quadratic_form(quadratic)
sage: type(inv3)
<class 'sage.rings.invariants.invariant_theory.TernaryQuadratic'>
sage: inv4 = invariant_theory.inhomogeneous_quadratic_form(x^2+y^2+z^2)
sage: type(inv4)
<class 'sage.rings.invariants.invariant_theory.QuadraticForm'>
```

**quadratic_form**(*polynomial*, *args*)

Invariants of a homogeneous quadratic form.

**INPUT:**

- *polynomial* – a homogeneous or inhomogeneous quadratic form.
- *args* – the variables as multiple arguments, or as a single list/tuple. If the last argument is None, the cubic is assumed to be inhomogeneous.

**EXAMPLES:**

```python
```
If some of the ring variables are to be treated as coefficients you need to specify the polynomial variables:

```python
sage: R.<x,y,z, a,b> = QQ[]
sage: quadratic = a*x^2+b*y^2+z^2
sage: invariant_theory.quadratic_form(quadratic, x,y,z)
Ternary quadratic with coefficients (a, b, 1, 0, 0, 2)
sage: invariant_theory.quadratic_form(quadratic, [x,y,z])  # alternate syntax
Ternary quadratic with coefficients (a, b, 1, 0, 0, 2)
```

Inhomogeneous quadratic forms (see also `inhomogeneous_quadratic_form()`) can be specified by passing `None` as the last variable:

```python
sage: inhom = quadratic.subs(z=1)
sage: invariant_theory.quadratic_form(inhom, x,y,None)
Ternary quadratic with coefficients (a, b, 1, 0, 0, 2)
```

`quaternary_biquadratic(quadratic1, quadratic2, *args, **kwds)`
Invariants of two quadratics in four variables.

**INPUT:**

- `quadratic1, quadratic2` – two polynomials. Either homogeneous quadratic in 4 homogeneous variables, or inhomogeneous quadratic in 3 variables.
- `w, x, y, z` – the variables. If `z` is `None`, the quadratics are assumed to be inhomogeneous.

**EXAMPLES:**

```python
sage: R.<w,x,y,z> = QQ[]
sage: q1 = w^2+x^2+y^2+z^2
sage: q2 = w*x + y*z
sage: inv = invariant_theory.quaternary_biquadratic(q1, q2)
sage: type(inv)
<class 'sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics'>
```

Distance between two spheres [Sal1958], [Sal1965]

```python
sage: R.<x,y,z, a,b,c, r1,r2> = QQ[]
sage: S1 = -r1^2 + x^2 + y^2 + z^2
sage: S2 = -r2^2 + (x-a)^2 + (y-b)^2 + (z-c)^2
sage: inv = invariant_theory.quaternary_biquadratic(S1, S2, [x, y, z])
sage: inv.Delta_invariant()  
-r1^2
sage: inv.Delta_prime_invariant()  
-r2^2
sage: inv.Theta_invariant()  
a^2 + b^2 + c^2 - 3*r1^2 - r2^2
sage: inv.Theta_prime_invariant()  
a^2 + b^2 + c^2 - r1^2 - 3*r2^2
sage: inv.Phi_invariant()  
2*a^2 + 2*b^2 + 2*c^2 - 3*r1^2 - 3*r2^2
sage: inv.J_covariant() 
0
```
quaternary_quadratic(quadratic, *args)
Invariant theory of a quadratic in four variables.

INPUT:

- quadratic – a quadratic form.
- w, x, y, z – the homogeneous variables. If z is None, the quadratic is assumed to be inhomogeneous.

REFERENCES:

- Wikipedia article Invariant_of_a_binary_form

EXAMPLES:

```sage
sage: R.<w,x,y,z> = QQ[]
sage: invariant_theory.quaternary_quadratic(w^2+x^2+y^2+z^2)
Quaternary quadratic with coefficients (1, 1, 1, 0, 0, 0, 0, 0, 0, 0)
sage: R.<x,y,z> = QQ[]
sage: invariant_theory.quaternary_quadratic(1+x^2+y^2+z^2)
Quaternary quadratic with coefficients (1, 1, 1, 0, 0, 0, 0, 0, 0, 0)
```

ternary_biquadratic(quadratic1, quadratic2, *args, **kwds)
Invariants of two quadratics in three variables.

INPUT:

- quadratic1, quadratic2 – two polynomials. Either homogeneous quadratic in 3 homogeneous variables, or inhomogeneous quadratic in 2 variables.
- x, y, z – the variables. If z is None, the quadratics are assumed to be inhomogeneous.

EXAMPLES:

```sage
sage: R.<x,y,z> = QQ[]
sage: q1 = x^2+y^2+z^2
sage: q2 = x*y + y*z + x*z
sage: inv = invariant_theory.ternary_biquadratic(q1, q2)
sage: type(inv)
<class 'sage.rings.invariants.invariant_theory.TwoTernaryQuadratics'>
```

Distance between two circles:

```sage
sage: R.<x,y, a,b, r1,r2> = QQ[]
sage: S1 = -r1^2 + x^2 + y^2
sage: S2 = -r2^2 + (x-a)^2 + (y-b)^2
sage: inv = invariant_theory.ternary_biquadratic(S1, S2, [x, y])
sage: inv.Delta_invariant()
-r1^2
sage: inv.Delta_prime_invariant()
-r2^2
sage: inv.Theta_invariant()
a^2 + b^2 - 2*r1^2 - r2^2
sage: inv.Theta_prime_invariant()
a^2 + b^2 - r1^2 - 2*r2^2
sage: inv.F_covariant()
2*x^2*a^2 + y^2*a^2 - 2*x*a*b^2 - 2*a^2*b^2 - 2*y*b^3 + b^4 - 2*x^2*r1^2 - 2*y^2*r1^2 + 2*x*a*r1^2 - 2*a^2*r1^2 + 2*y*b*r1^2 + r1^4 - 2*x^2*r2^2 + 2*y^2*r2^2 + 2*x*a*r2^2 - 2*a^2*r2^2 + 2*y*b*r2^2 + 2*b^2*r2^2 +
```
ternary_cubic(cubic, *args, **kwds)

Invariants of a cubic in three variables.

The algebra of invariants of a ternary cubic under $SL_3(\mathbb{C})$ is a polynomial algebra generated by two invariants $S$ (see `S_invariant()`) and $T$ (see `T_invariant()`) of degrees 4 and 6, called Aronhold invariants.

The ring of covariants is given as follows. The identity covariant $U$ of a ternary cubic has degree 1 and order 3. The Hessian $H$ (see `Hessian()`) is a covariant of ternary cubics of degree 3 and order 3. There is a covariant $\Theta$ (see `Theta_covariant()`) of ternary cubics of degree 8 and order 6 that vanishes on points $x$ lying on the Salmon conic of the polar of $x$ with respect to the curve and its Hessian curve. The Briowski covariant $J$ (see `J_covariant()`) is the Jacobian of $U$, $\Theta$, and $H$ of degree 12, order 9. The algebra of covariants of a ternary cubic is generated over the ring of invariants by $U$, $\Theta$, $H$, and $J$, with a relation

$$J^2 = 4\Theta^3 + TU^2\Theta^2 + \Theta(-4S^3U^4 + 2STU^3H - 72S^2U^2H^2$$
$$- 18TUH^3 + 108SH^4) - 16S^4U^5H - 11S^2TU^4H^2$$
$$- 4T^2U^3H^3 + 54STU^2H^4 - 432S^2U^2H^5 - 27TH^6$$

REFERENCES:

- Wikipedia article Ternary_cubic

INPUT:

- cubic – a homogeneous cubic in 3 homogeneous variables, or an inhomogeneous cubic in 2 variables.
- x, y, z – the variables. If z is None, the cubic is assumed to be inhomogeneous.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+z^3)
sage: type(cubic)
<class 'sage.rings.invariants.invariant_theory.TernaryCubic'>
```

ternary_quadratic(quadratic, *args, **kwds)

Invariants of a quadratic in three variables.

INPUT:

- quadratic – a homogeneous quadratic in 3 homogeneous variables, or an inhomogeneous quadratic in 2 variables.
- x, y, z – the variables. If z is None, the quadratic is assumed to be inhomogeneous.

REFERENCES:

- Wikipedia article Invariant_of_a_binary_form

EXAMPLES:
class sage.rings.invariants.invariant_theory.QuadraticForm(n, d, polynomial, *args)

Bases: sage.rings.invariants.invariant_theory.AlgebraicForm

Invariant theory of a multivariate quadratic form.

You should use the invariant_theory factory object to construct instances of this class. See quadratic_form() for details.

as_QuadraticForm()
Convert into a QuadraticForm.

OUTPUT:

Sage has a special quadratic forms subsystem. This method converts self into this QuadraticForm representation.

EXAMPLES:

sage: R.<x,y,z> = QQ[]
sage: p = x^2+y^2+z^2+2*x*y+3*x*z
sage: quadratic = invariant_theory.ternary_quadratic(p)
sage: matrix(quadratic)
[ 1 1 3/2]
[ 1 1 0 ]
[3/2 0 1]
sage: quadratic.as_QuadraticForm()
Quadratic form in 3 variables over Multivariate Polynomial Ring in x, y, z over Rational Field with coefficients:
[ 1 2 3 ]
[ * 1 0 ]
[ * * 1 ]
sage: _.polynomial('X,Y,Z')
X^2 + 2*X*Y + Y^2 + 3*X*Z + Z^2

coeffs()
The coefficients of a quadratic form.

Given

\[ f(x) = \sum_{0 \leq i < n} a_i x_i^2 + \sum_{0 \leq j < k < n} a_{jk} x_j x_k \]

this function returns \( a = (a_0, \ldots, a_n, a_{00}, a_{01}, \ldots, a_{n-1,n}) \)

EXAMPLES:
```python
sage: R.<a,b,c,d,e,f,g, x,y,z> = QQ[]
sage: p = a*x^2 + b*y^2 + c*z^2 + d*x*y + e*x*z + f*y*z
sage: inv = invariant_theory.quadratic_form(p, x,y,z); inv
Ternary quadratic with coefficients (a, b, c, d, e, f)
sage: inv.coeffs()
(a, b, c, d, e, f)
sage: inv.scaled_coeffs()
(a, b, c, 1/2*d, 1/2*e, 1/2*f)
```

**discriminant()**

Return the discriminant of the quadratic form.

Up to an overall constant factor, this is just the determinant of the defining matrix, see `matrix()`.

For a quadratic form in \( n \) variables, the overall constant is \( 2^{n-1} \) if \( n \) is odd and \( (-1)^{n/2}2^n \) if \( n \) is even.

**EXAMPLES:**

```python
sage: R.<a,b,c, x,y> = QQ[]
sage: p = a*x^2+b*x*y+c*y^2
sage: quadratic = invariant_theory.quadratic_form(p, x,y)
sage: quadratic.discriminant()
b^2 - 4*a*c
sage: R.<a,b,c,d,e,f,g, x,y,z> = QQ[]
sage: p = a*x^2 + b*y^2 + c*z^2 + d*x*y + e*x*z + f*y*z
sage: quadratic = invariant_theory.quadratic_form(p, x,y,z)
sage: quadratic.discriminant()
4*a*b*c - c*d^2 - b*e^2 + d*e*f - a*f^2
```

**dual()**

Return the dual quadratic form.

**OUTPUT:**

A new quadratic form (with the same number of variables) defined by the adjoint matrix.

**EXAMPLES:**

```python
sage: R.<a,b,c,x,y,z> = QQ[]
sage: cubic = x^2+y^2+z^2
sage: quadratic = invariant_theory.ternary_quadratic(a*x^2+b*y^2+c*z^2, [x,y, z])
sage: quadratic.form()
a*x^2 + b*x*y + c*x*z
sage: quadratic.dual().form()
b*c*x^2 + a*c*y^2 + a*b*z^2
sage: R.<x,y,z, t> = QQ[]
sage: cubic = x^2+y^2+z^2
sage: quadratic = invariant_theory.ternary_quadratic(x^2+y^2+z^2 + t*x*y, [x, y, z])
sage: quadratic.dual()
Ternary quadratic with coefficients (1, 1, -1/4*t^2 + 1, -t, 0, 0)
sage: R.<x,y, t> = QQ[]
sage: quadratic = invariant_theory.ternary_quadratic(x^2+y^2+1 + t*x*y, [x,y])
sage: quadratic.dual()
Ternary quadratic with coefficients (1, 1, -1/4*t^2 + 1, -t, 0, 0)
```
**classmethod from_invariants** (discriminant, x, z, *args, **kwargs)

Construct a binary quadratic from its discriminant.

This function constructs a binary quadratic whose discriminant equal the one provided as argument up to scaling.

**INPUT:**
- discriminant – Value of the discriminant used to reconstruct the binary quadratic.

**OUTPUT:**
A QuadraticForm with 2 variables.

**EXAMPLES:**

```python
sage: R.<x,y> = QQ[]
sage: from sage.rings.invariants.invariant_theory import QuadraticForm
sage: QuadraticForm.from_invariants(1, x, y)
Binary quadratic with coefficients (1, -1/4, 0)
```

**invariants** (type='discriminant')

Return a tuple of invariants of a binary quadratic.

**INPUT:**
- type – The type of invariants to return. The default choice is to return the discriminant.

**OUTPUT:**
The invariants of the binary quadratic.

**EXAMPLES:**

```python
sage: R.<x0, x1> = QQ[]
sage: p = 2*x1^2 + 5*x0*x1 + 3*x0^2
sage: quadratic = invariant_theory.binary_quadratic(p, x0, x1)
sage: quadratic.invariants()
(1,)
sage: quadratic.invariants('unknown')
Traceback (most recent call last):
  ... ValueError: unknown type of invariants unknown for a binary quadratic
```

**matrix()**

Return the quadratic form as a symmetric matrix

**OUTPUT:**
This method returns a symmetric matrix $A$ such that the quadratic $Q$ equals

$$Q(x, y, z, \ldots) = (x, y, \ldots)A(x, y, \ldots)^t$$

**EXAMPLES:**

```python
sage: R.<x,y,z> = QQ[]
sage: quadratic = invariant_theory.ternary_quadratic(x^2+y^2+z^2+x*y)
sage: matrix(quadratic)
[ 1 1/2 0]
[1/2 1 0]
[ 0 0 1]
sage: quadratic._matrix_() == matrix(quadratic)
True
```
monomials()

List the basis monomials in the form.

OUTPUT:

A tuple of monomials. They are in the same order as coeffs().

EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: quadratic = invariant_theory.quadratic_form(x^2+y^2)
sage: quadratic.monomials()
(x^2, y^2, x*y)
sage: quadratic = invariant_theory.inhomogeneous_quadratic_form(x^2+y^2)
sage: quadratic.monomials()
(x^2, y^2, 1, x*y, x, y)
```

scaled_coeffs()

The scaled coefficients of a quadratic form.

Given

\[ f(x) = \sum_{0 \leq i < n} a_i x_i^2 + \sum_{0 \leq j < k < n} 2a_{jk} x_j x_k \]

this function returns \(a = (a_0, \ldots, a_n, a_{00}, a_{01}, \ldots, a_{n-1,n})\)

EXAMPLES:

```python
sage: R.<a,b,c,d,e,f,g, x,y,z> = QQ[]
sage: p = a*x^2 + b*y^2 + c*z^2 + d*x*y + e*x*z + f*y*z
sage: inv = invariant_theory.quadratic_form(p, x,y,z); inv
Ternary quadratic with coefficients (a, b, c, d, e, f)
sage: inv.coeffs()
(a, b, c, d, e, f)
sage: inv.scaled_coeffs()
(a, b, c, 1/2*d, 1/2*e, 1/2*f)
```

class sage.rings.invariants.invariant_theory.SeveralAlgebraicForms (forms)

Bases: sage.rings.invariants.invariant_theory.FormsBase

The base class of multiple algebraic forms (i.e. homogeneous polynomials).

You should only instantiate the derived classes of this base class.

See AlgebraicForm for the base class of a single algebraic form.

INPUT:

- forms – a list/tuple/iterable of at least one AlgebraicForm object, all with the same number of variables. Interpreted as multiple homogeneous polynomials in a common polynomial ring.

EXAMPLES:

```python
sage: from sage.rings.invariants.invariant_theory import AlgebraicForm,
   \_SeveralAlgebraicForms
sage: R.<x,y> = QQ[]
sage: p = AlgebraicForm(2, 2, x^2, (x,y))
sage: q = AlgebraicForm(2, 2, y^2, (x,y))
sage: pq = SeveralAlgebraicForms([p, q])
```
get_form(i)
Return the i-th form.

EXAMPLES:

```sage
sage: R.<x,y> = QQ[]
sage: q1 = invariant_theory.quadratic_form(x^2 + y^2)
sage: q2 = invariant_theory.quadratic_form(x*y)
sage: from sage.rings.invariants.invariant_theory import SeveralAlgebraicForms
sage: q12 = SeveralAlgebraicForms([q1, q2])
sage: q12.get_form(0) is q1
True
sage: q12.get_form(1) is q2
True
sage: q12[0] is q12.get_form(0)  # syntactic sugar
True
sage: q12[1] is q12.get_form(1)  # syntactic sugar
True
```

homogenized(var='h')
Return form as defined by a homogeneous polynomial.

INPUT:

• var – either a variable name, variable index or a variable (default: 'h').

OUTPUT:
The same algebraic form, but defined by a homogeneous polynomial.

EXAMPLES:

```sage
sage: R.<x,y,z> = QQ[]
sage: q = invariant_theory.quaternary_biquadratic(x^2+1, y^2+1, [x,y,z])
sage: q
Joint quaternary quadratic with coefficients (1, 0, 0, 1, 0, 0, 0, 0, 0, 0)
and quaternary quadratic with coefficients (0, 1, 0, 1, 0, 0, 0, 0, 0, 0)
sage: q.homogenized()
Joint quaternary quadratic with coefficients (1, 0, 0, 1, 0, 0, 0, 0, 0, 0)
and quaternary quadratic with coefficients (0, 1, 0, 1, 0, 0, 0, 0, 0, 0)
sage: type(q) is type(q.homogenized())
True
```

n_forms()
Return the number of forms.

EXAMPLES:

```sage
sage: R.<x,y> = QQ[]
sage: q1 = invariant_theory.quadratic_form(x^2 + y^2)
sage: q2 = invariant_theory.quadratic_form(x*y)
sage: from sage.rings.invariants.invariant_theory import SeveralAlgebraicForms
sage: q12 = SeveralAlgebraicForms([q1, q2])
sage: q12.n_forms()
2
sage: len(q12) == q12.n_forms()  # syntactic sugar
True
```

class sage.rings.invariants.invariant_theory.TernaryCubic(n, d, polynomial, *args)
Bases: sage.rings.invariants.invariant_theory.AlgebraicForm
Invariant theory of a ternary cubic.

You should use the `invariant_theory` factory object to construct instances of this class. See `ternary_cubic()` for details.

**Hessian()**

Return the Hessian covariant.

**OUTPUT:**

The Hessian matrix multiplied with the conventional normalization factor $1/216$.

**EXAMPLES:**

```python
sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+z^3)
sage: cubic.Hessian()  
x*y*z
sage: R.<x,y> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+1)
sage: cubic.Hessian()  
x*y
```

**J_covariant()**

Return the J-covariant of the ternary cubic.

**EXAMPLES:**

```python
sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+z^3)
sage: cubic.J_covariant()  
x^6*y^3 - x^3*y^6 - x^6*z^3 + y^6*z^3 + x^3*z^6 - y^3*z^6
sage: R.<x,y> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+1)
sage: cubic.J_covariant()  
x^6*y^3 - x^3*y^6 - x^6 + y^6 + x^3 - y^3
```

**S_invariant()**

Return the S-invariant.

**EXAMPLES:**

```python
sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^2*y+y^3+z^3+x*y*z)
sage: cubic.S_invariant()  
-1/1296
```

**T_invariant()**

Return the T-invariant.

**EXAMPLES:**

```python
sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+z^3)
sage: cubic.T_invariant()  
1
sage: R.<x,y,z,t> = GF(7)[[]]
```

(continues on next page)
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+z^3+t*x*y*z, \ [x,y,z])
sage: cubic.T_invariant()
-t^6 - t^3 + 1

Theta_covariant()

Return the $\Theta$ covariant.

EXAMPLES:

sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+z^3)
sage: cubic.Theta_covariant()
-x^3*y^3 - x^3*z^3 - y^3*z^3

sage: R.<x,y> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+1)
sage: cubic.Theta_covariant()
-x^3 - y^3

sage: R.<x,y,z,a30,a21,a12,a03,a20,a11,a02,a10,a01,a00> = QQ[]

sage: p = ( a30*x^3 + a21*x^2*y + a12*x*y^2 + a03*y^3 + a20*x^2*z +
.....: a11*x*y*z + a02*y^2*z + a10*x*z^2 + a01*y*z^2 + a00*z^3 )
sage: cubic = invariant_theory.ternary_cubic(p, x,y,z)
sage: len(list(cubic.Theta_covariant()))
6952

coeffs()

Return the coefficients of a cubic.

Given

\[
p(x, y) = a_{30}x^3 + a_{21}x^2y + a_{12}xy^2 + a_{03}y^3 + a_{20}x^2+
    a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00}
\]

this function returns \(a = (a_{30}, a_{03}, a_{00}, a_{21}, a_{20}, a_{12}, a_{02}, a_{10}, a_{01}, a_{11})\)

EXAMPLES:

sage: R.<x,y,z,a30,a21,a12,a03,a20,a11,a02,a10,a01,a00> = QQ[]

sage: p = ( a30*x^3 + a21*x^2*y + a12*x*y^2 + a03*y^3 + a20*x^2*z +
.....: a11*x*y*z + a02*y^2*z + a10*x*z^2 + a01*y*z^2 + a00*z^3 )
sage: cubic = invariant_theory.ternary_cubic(p, x,y,z).coeffs()
(a30, a03, a00, a21, a20, a12, a02, a10, a01, a11)

sage: cubic = invariant_theory.ternary_cubic(p.subs(z=1), x, y).coeffs()
(a30, a03, a00, a21, a20, a12, a02, a10, a01, a11)

monomials()

List the basis monomials of the form.

OUTPUT:

A tuple of monomials. They are in the same order as \texttt{coeffs()}.

EXAMPLES:

sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+z^2)
sage: cubic.monomials()
(x^3, y^3, z^3, x^2*y, x^2*z, y^3*z, x*y^2, y^2*z, x*z^2, y*z^2, x*y*z)
polar_conic()
Return the polar conic of the cubic.

OUTPUT:
Given the ternary cubic \( f(X, Y, Z) \), this method returns the symmetric matrix \( A(x, y, z) \) defined by

\[
x f_X + y f_Y + z f_Z = (X, Y, Z) \cdot A(x, y, z) \cdot (X, Y, Z)^t
\]

EXAMPLES:
```
sage: R.<x,y,z,X,Y,Z,a30,a21,a12,a03,a20,a10,a02,a11,a01,a00> = QQ[]
sage: p = ( a30*x^3 + a21*x^2*y + a12*x*y^2 + a03*y^3 + a20*x^2*z +
        ....:     a11*x*y*z + a02*y^2*z + a10*x*z^2 + a01*y*z^2 + a00*z^3 )
sage: cubic = invariant_theory.ternary_cubic(p, x,y,z)
sage: cubic.polar_conic()
[ 3*x*a30 + y*a21 + z*a20 x*a21 + y*a12 + 1/2*z*a11 x*a20 + 1/2*y*a11 +
  z*a10]
[ x*a21 + y*a12 + 1/2*z*a11 x*a12 + 3*y*a03 + z*a02 1/2*x*a11 + y*a02 +
  z*a01]
[ x*a20 + 1/2*y*a11 + z*a10 1/2*x*a11 + y*a02 + z*a01 x*a10 + y*a01 +
  3*z*a00]
sage: polar_eqn = X*p.derivative(x) + Y*p.derivative(y) + Z*p.derivative(z)
sage: polar = invariant_theory.ternary_quadratic(polar_eqn, [x,y,z])
sage: polar.matrix().subs(X=x,Y=y,Z=z) == cubic.polar_conic()
True
```

scaled_coeffs()
Return the coefficients of a cubic.

Compared to coeffs(), this method returns rescaled coefficients that are often used in invariant theory.

Given

\[
p(x, y) = a_{30}x^3 + a_{21}x^2y + a_{12}xy^2 + a_{03}y^3 + a_{20}x^2 +
        a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00}
\]

this function returns \( a = (a_{30}, a_{03}, a_{00}, a_{21}/3, a_{20}/3, a_{12}/3, a_{02}/3, a_{10}/3, a_{01}/3, a_{11}/6) \)

EXAMPLES:
```
sage: R.<x,y,z,a30,a21,a12,a03,a20,a10,a02,a11,a01,a00> = QQ[]
sage: p = ( a30*x^3 + a21*x^2*y + a12*x*y^2 + a03*y^3 + a20*x^2*z +
        ....:     a11*x*y*z + a02*y^2*z + a10*x*z^2 + a01*y*z^2 + a00*z^3 )
sage: invariant_theory.ternary_cubic(p, x,y,z).scaled_coeffs()
(a30, a03, a00, 1/3*a21, 1/3*a20, 1/3*a12, 1/3*a02, 1/3*a10, 1/3*a01, 1/6*a11)
```

syzygy (U, S, T, H, Theta, J)
Return the syzygy of the cubic evaluated on the invariants and covariants.

INPUT:
• U, S, T, H, Theta, J – polynomials from the same polynomial ring.

OUTPUT:
0 if evaluated for the form, the S invariant, the T invariant, the Hessian, the \( \Theta \) covariant and the J-covariant of a ternary cubic.

EXAMPLES:
```python
sage: R.<x,y,z> = QQ[]
sage: monomials = [x^3, y^3, z^3, x^2*y, x^2*z, x*y^2,
....: y^2*z, x*z^2, y*z^2, x*y*z]
sage: random_poly = sum([ randint(0,10000) * m
for m in monomials ])
sage: cubic = invariant_theory.ternary_cubic(random_poly)
sage: U = cubic.form()
sage: S = cubic.S_invariant()
sage: T = cubic.T_invariant()
sage: H = cubic.Hessian()
sage: Theta = cubic.Theta_covariant()
sage: J = cubic.J_covariant()
sage: cubic.syzygy(U, S, T, H, Theta, J)
```

```python
class sage.rings.invariants.invariant_theory.TernaryQuadratic(n, d, polynomial, *args)

Bases: sage.rings.invariants.invariant_theory.QuadraticForm

Invariant theory of a ternary quadratic.

You should use the invariant_theory factory object to construct instances of this class. See ternary_quadratic() for details.

coeffs()

Return the coefficients of a quadratic.

Given

\[ p(x, y) = a_{20}x^2 + a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00} \]

this function returns \( a = (a_{20}, a_{02}, a_{00}, a_{11}, a_{10}, a_{01}) \)

EXAMPLES:

```python
sage: R.<x,y,z,a20,a11,a02,a10,a01,a00> = QQ[]
sage: p = ( a20*x^2 + a11*x*y + a02*y^2 +
....: a10*x + a01*y + a00 )
sage: invariant_theory.ternary_quadratic(p, x,y,z).coeffs()
(a20, a02, a00, a11, a10, a01)
```

covariant_conic(other)

Return the ternary quadratic covariant to self and other.

INPUT:

- other – Another ternary quadratic.

OUTPUT:

The so-called covariant conic, a ternary quadratic. It is symmetric under exchange of self and other.

EXAMPLES:

```python
sage: ring.<x,y,z> = QQ[]
sage: Q = invariant_theory.ternary_quadratic(x^2+y^2+z^2)
sage: R = invariant_theory.ternary_quadratic(x*y+x*z+y*z)
sage: Q.covariant_conic(R)
-x*y - x*z - y*z
sage: R.covariant_conic(Q)
-x*y - x*z - y*z
```

3.2. Classical Invariant Theory
monomials()
List the basis monomials of the form.

OUTPUT:
A tuple of monomials. They are in the same order as coeffs().

EXAMPLES:
```
sage: R.<x,y,z> = QQ[]
sage: quadratic = invariant_theory.ternary_quadratic(x^2+y*z)
sage: quadratic.monomials()
(x^2, y^2, z^2, x*y, x*z, y*z)
```

scaled_coeffs()
Return the scaled coefficients of a quadratic.

Given

\[ p(x, y) = a_{20}x^2 + a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00} \]

this function returns \( a = (a_{20}, a_{02}, a_{00}, a_{11}/2, a_{10}/2, a_{01}/2, ) \)

EXAMPLES:
```
sage: R.<x,y,z,a20,a11,a02,a10,a01,a00> = QQ[
 sage: p = ( a20*x^2 + a11*x*y + a02*y^2 +
 ....: a10*x + a01*y + a00*z^2 )
 sage: invariant_theory.ternary_quadratic(p, x,y,z).scaled_coeffs()
 (a20, a02, a00, 1/2*a11, 1/2*a10, 1/2*a01)
```

class sage.rings.invariants.invariant_theory.TwoAlgebraicForms/forms
Bases: sage.rings.invariants.invariant_theory.SeveralAlgebraicForms

first()
Return the first of the two forms.

OUTPUT:
The first algebraic form used in the definition.

EXAMPLES:
```
sage: R.<x,y> = QQ[]
 sage: q0 = invariant_theory.quadratic_form(x^2 + y^2)
 sage: q1 = invariant_theory.quadratic_form(x*y)
 sage: from sage.rings.invariants.invariant_theory import TwoAlgebraicForms
 sage: q = TwoAlgebraicForms([q0, q1])
 sage: q.first() is q0
 True
 sage: q.get_form(0) is q0
 True
 sage: q.first().polynomial()
 x^2 + y^2
```

second()
Return the second of the two forms.

OUTPUT:
The second form used in the definition.

**EXAMPLES:**

```python
sage: R.<x,y> = QQ[]
sage: q0 = invariant_theory.quadratic_form(x^2 + y^2)
sage: q1 = invariant_theory.quadratic_form(x*y)
sage: from sage.rings.invariants.invariant_theory import TwoAlgebraicForms
sage: q = TwoAlgebraicForms([q0, q1])
sage: q.second() is q1
True
sage: q.get_form(1) is q1
True
sage: q.second().polynomial()
x*y
```

**class** `sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics(forms)`

**Bases:** `sage.rings.invariants.invariant_theory.TwoAlgebraicForms`

Invariant theory of two quaternary quadratics.

You should use the `invariant_theory` factory object to construct instances of this class. See `quaternary_biquadratics()` for details.

**REFERENCES:**

- section on “Invariants and Covariants of Systems of Quadrics” in [Sal1958], [Sal1965]

**Delta_invariant()**

Return the $\Delta$ invariant.

**EXAMPLES:**

```python
sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[]
sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).
coefficients(sparse=False)
True
```

**Delta_prime_invariant()**

Return the $\Delta'$ invariant.

**EXAMPLES:**

```python
sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[]
sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).
coefficients(sparse=False)
sage: q.Delta_prime_invariant() == coeffs[0]
True
```
**J_covariant()**

The $J$-covariant.

This is the Jacobian determinant of the two biquadratics, the $T$-covariant, and the $T'$-covariant with respect to the four homogeneous variables.

**EXAMPLES:**

```
sage: R.<w,x,y,z,a0,a1,a2,a3,A0,A1,A2,A3> = QQ[]
sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3*w^2
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3*w^2
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [w, x, y, z])
sage: q.J_covariant().factor()  # long time
z * y * x * w * (a3*A2 - a2*A3) * (a3*A1 - a1*A3) * (-a2*A1 + a1*A2) *
   (a3*A0 - a0*A3) * (-a2*A0 + a0*A2) * (-a1*A0 + a0*A1)
```

**Phi_invariant()**

Return the $\Phi'$ invariant.

**EXAMPLES:**

```
sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[]
sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).
   coefficients(sparse=False)
sage: q.Phi_invariant() == coeffs[2]
True
```

**T_covariant()**

The $T$-covariant.

**EXAMPLES:**

```
sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[]
sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])
sage: T = invariant_theory.quaternary_quadratic(q.T_covariant(), [x,y,z]).
   matrix()
sage: M = q[0].matrix().adjugate() + t*q[1].matrix().adjugate()
sage: M = M.adjugate().apply_map(# long time (4s on my thinkpad)
   # long time
   lambda m: m.coefficient(t))
sage: M == q.Delta_invariant()*T
True
```

**T_prime_covariant()**

The $T'$-covariant.

**EXAMPLES:**

```
sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[]
sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])
sage: T = invariant_theory.quaternary_quadratic(q.T_prime_covariant(), [x,y,z]).
   matrix()
sage: M = q[0].matrix().adjugate() + t*q[1].matrix().adjugate()
sage: M = M.adjugate().apply_map(# long time (4s on my thinkpad)
   # long time
   lambda m: m.coefficient(t))
sage: M == q.Delta_invariant()*T
True
```
sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[]
sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])
sage: Tprime = invariant_theory.quaternary_quadratic(q.T_prime_covariant(), [x,y,z]).matrix()
sage: M = q[0].matrix().adjugate() + t*q[1].matrix().adjugate() # long time (4s on my thinkpad W530)
    sage: M = M.adjugate().apply_map(lambda m: m.coefficient(t^2))
    sage: M == q.Delta_prime_invariant() * Tprime # long time
True

Theta_invariant()

Return the $\Theta$ invariant.

EXAMPLES:

sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[]
sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])
    sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).
        coefficients(sparse=False)
    sage: q.Theta_invariant() == coeffs[3]
True

Theta_prime_invariant()

Return the $\Theta'$ invariant.

EXAMPLES:

sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[]
sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])
    sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).
        coefficients(sparse=False)
    sage: q.Theta_prime_invariant() == coeffs[1]
True

syzygy(Delta, Theta, Phi, Theta_prime, Delta_prime, U, V, T, T_prime, J)

Return the syzygy evaluated on the invariants and covariants.

INPUT:

- Delta, Theta, Phi, Theta_prime, Delta_prime, U, V, T, T_prime, J - polynomials from the same polynomial ring.

OUTPUT:
Zero if the $U$ is the first polynomial, $V$ the second polynomial, and the remaining input are the invariants and covariants of a quaternary biquadratic.

**EXAMPLES:**

```python
sage: R.<w,x,y,z> = QQ[]
sage: monomials = [x^2, x*y, y^2, x*z, y*z, z^2, x*w, y*w, z*w, w^2]
sage: def qRnd():
    return sum(randint(-1000,1000)*m for m in monomials)
sage: biquadratic = invariant_theory.quaternary_biquadratic(qRnd(), qRnd())
sage: Delta = biquadratic.Delta_invariant()
sage: Theta = biquadratic.Theta_invariant()
sage: Phi = biquadratic.Phi_invariant()
sage: Theta_prime = biquadratic.Theta_prime_invariant()
sage: Delta_prime = biquadratic.Delta_prime_invariant()
sage: U = biquadratic.first().polynomial()
sage: V = biquadratic.second().polynomial()
sage: T = biquadratic.T_covariant()
sage: T_prime = biquadratic.T_prime_covariant()
sage: J = biquadratic.J_covariant()
sage: biquadratic.syzygy(Delta, Theta, Phi, Theta_prime, Delta_prime, U, V, T, T_prime, J)
0
```

If the arguments are not the invariants and covariants then the output is some (generically non-zero) polynomial:

```python
sage: biquadratic.syzygy(1, 1, 1, 1, 1, 1, 1, 1, 1, x)
-x^2 + 1
```

**class** `sage.rings.invariants.invariant_theory.TwoTernaryQuadratics` *(forms)*

```
Bases: sage.rings.invariants.invariant_theory.TwoAlgebraicForms
```

Invariant theory of two ternary quadratics.

You should use the `invariant_theory` factory object to construct instances of this class. See `ternary_biquadratics()` for details.

**REFERENCES:**

- Section on “Invariants and Covariants of Systems of Conics”, Art. 388 (a) in [Sal1954]

**Delta_invariant()**

Return the $\Delta$ invariant.

**EXAMPLES:**

```python
sage: R.<a00, a01, a11, a02, a12, b00, b01, b11, b02, b12, y0, y1, y2, t> = QQ[]
sage: p1 = a00*y0^2 + 2*a01*y0*y1 + a11*y1^2 + 2*a02*y0*y2 + 2*a12*y1*y2 + a22*y2^2
sage: p2 = b00*y0^2 + 2*b01*y0*y1 + b11*y1^2 + 2*b02*y0*y2 + 2*b12*y1*y2 + b22*y2^2
sage: q = invariant_theory.ternary_biquadratic(p1, p2, [y0, y1, y2])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).
coefficients(sparse=False)
sage: q.Delta_invariant() == coeffs[3]
True
```

**Delta_prime_invariant()**

Return the $\Delta'$ invariant.

---

Chapter 3. Multivariate Polynomials
EXAMPLES:

```python
sage: R.<a00, a01, a11, a02, a12, b00, b01, b11, b02, b12, b22, y0, y1, y2, t> = QQ[]
sage: p1 = a00*y0^2 + 2*a01*y0*y1 + a11*y1^2 + 2*a02*y0*y2 + 2*a12*y1*y2 + a22*y2^2
sage: p2 = b00*y0^2 + 2*b01*y0*y1 + b11*y1^2 + 2*b02*y0*y2 + 2*b12*y1*y2 + b22*y2^2
sage: q = invariant_theory.ternary_biquadratic(p1, p2, [y0, y1, y2])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).coefficients(sparse=False)
sage: q.Delta_prime_invariant() == coeffs[0]
True
```

**F_covariant()**
Return the $F$ covariant.

EXAMPLES:

```python
sage: R.<a00, a01, a11, a02, a12, a22, b00, b01, b11, b02, b12, b22, x, y> = QQ[]
sage: p1 = 73*x^2 + 96*x*y - 11*y^2 + 4*x + 63*y + 57
sage: p2 = 61*x^2 - 100*x*y - 72*y^2 - 81*x + 39*y - 7
sage: q = invariant_theory.ternary_biquadratic(p1, p2, [x, y])
sage: q.F_covariant()
-32566577*x^2 + 29060637/2*x*y + 20153633/4*y^2 - 30250497/2*x - 241241273/4*y - 323820473/16
```

**J_covariant()**
Return the $J$ covariant.

EXAMPLES:

```python
sage: R.<a00, a01, a11, a02, a12, a22, b00, b01, b11, b02, b12, b22, x, y> = QQ[]
sage: p1 = 73*x^2 + 96*x*y - 11*y^2 + 4*x + 63*y + 57
sage: p2 = 61*x^2 - 100*x*y - 72*y^2 - 81*x + 39*y - 7
sage: q = invariant_theory.ternary_biquadratic(p1, p2, [x, y])
sage: q.J_covariant()
1057324024445*x^3 + 1209531088209*x^2*y + 942116599708*x*y^2 + 984553030871*y^3 + 543715345505/2*x^2 - 3065093506021/2*x*y + 755263948570*y^2 - 1118430692650*x - 509948695327/4*y + 3369951531745/8
```

**Theta_invariant()**
Return the $\Theta$ invariant.

EXAMPLES:

```python
sage: R.<a00, a01, a11, a02, a12, a22, b00, b01, b11, b02, b12, b22, y0, y1, y2, t> = QQ[]
sage: p1 = a00*y0^2 + 2*a01*y0*y1 + a11*y1^2 + 2*a02*y0*y2 + 2*a12*y1*y2 + a22*y2^2
sage: p2 = b00*y0^2 + 2*b01*y0*y1 + b11*y1^2 + 2*b02*y0*y2 + 2*b12*y1*y2 + b22*y2^2
sage: q = invariant_theory.ternary_biquadratic(p1, p2, [y0, y1, y2])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).coefficients(sparse=False)
sage: q.Theta_invariant() == coeffs[2]
True
```
Theta_prime_invariant()  
Return the $\Theta'$ invariant.

**EXAMPLES:**

```python
sage: R.<a00, a01, a11, a02, a12, a22, b00, b01, b11, b02, b12, b22, y0, y1, y2, t> = QQ[]
sage: p1 = a00*y0^2 + 2*a01*y0*y1 + a11*y1^2 + 2*a02*y0*y2 + 2*a12*y1*y2 + a22*y2^2
sage: p2 = b00*y0^2 + 2*b01*y0*y1 + b11*y1^2 + 2*b02*y0*y2 + 2*b12*y1*y2 + b22*y2^2
sage: q = invariant_theory.ternary_biquadratic(p1, p2, [y0, y1, y2])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).coefficients(sparse=False)
sage: q.Theta_prime_invariant() == coeffs[1]
True
```

syzygy(Delta, Theta, Theta_prime, Delta_prime, S, S_prime, F, J)  
Return the syzygy evaluated on the invariants and covariants.

**INPUT:**

- Delta, Theta, Theta_prime, Delta_prime, S, S_prime, F, J – polynomials from the same polynomial ring.

**OUTPUT:**

Zero if $S$ is the first polynomial, $S_{\text{prime}}$ the second polynomial, and the remaining input are the invariants and covariants of a ternary biquadratic.

**EXAMPLES:**

```python
sage: R.<x,y,z> = QQ[]
sage: monomials = [x^2, x*y, y^2, x*z, y*z, z^2]
sage: def q_rnd():
    return sum(randint(-1000,1000)*m for m in monomials)
sage: biquadratic = invariant_theory.ternary_biquadratic(q_rnd(), q_rnd(), [x, y, z])
sage: Delta = biquadratic.Delta_invariant()
sage: Theta = biquadratic.Theta_invariant()
sage: Theta_prime = biquadratic.Theta_prime_invariant()
sage: Delta_prime = biquadratic.Delta_prime_invariant()
sage: S = biquadratic.first().polynomial()
sage: S_prime = biquadratic.second().polynomial()
sage: F = biquadratic.F_covariant()
sage: J = biquadratic.J_covariant()
sage: biquadratic.syzygy(Delta, Theta, Theta_prime, Delta_prime, S, S_prime, F, J)
0
```

If the arguments are not the invariants and covariants then the output is some (generically non-zero) polynomial:

```python
sage: biquadratic.syzygy(1, 1, 1, 1, 1, 1, x)
1/64*x^2 + 1
```

sage.rings.invariants.invariant_theory.transvectant(f, g, h=1, scale='default')  
Return the h-th transvectant of f and g.

**INPUT:**

- $f, g$ – two homogeneous binary forms in the same polynomial ring.
• \( h \) – the order of the transvectant. If it is not specified, the first transvectant is returned.

• \( \text{scale} \) – the scaling factor applied to the result. Possible values are 'default' and 'none'. The 'default' scaling factor is the one that appears in the output statement below, if the scaling factor is 'none' the quotient of factorials is left out.

**OUTPUT:**

The \( h \)-th transvectant of the listed forms \( f \) and \( g \):

\[
(f, g)_h = \frac{(d_f - h)! \cdot (d_g - h)!}{d_f! \cdot d_g!} \left( \left( \frac{\partial}{\partial x} \frac{\partial}{\partial z} - \frac{\partial}{\partial x'} \frac{\partial}{\partial z'} \right)^h (f(x, z) \cdot g(x', z')) \right)_{(x', z')=(x, z)}
\]

**EXAMPLES:**

```python
sage: from sage.rings.invariants.invariant_theory import AlgebraicForm,
      transvectant
sage: R.<x,y> = QQ[]

sage: f = AlgebraicForm(2, 5, x^5 + 5*x^4*y + 5*x*y^4 + y^5)

sage: transvectant(f, f, 4)
Binary quadratic given by 2*x^2 - 4*x*y + 2*y^2

sage: transvectant(f, f, 8)
Binary form of degree -6 given by 0
```

The default scaling will yield an error for fields of positive characteristic below \( d_f! \) or \( d_g! \) as the denominator of the scaling factor will not be invertible in that case. The scale argument 'none' can be used to compute the transvectant in this case:

```python
sage: R.<a0,a1,a2,a3,a4,a5,x0,x1> = GF(5)[]

sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: f = AlgebraicForm(2, 5, p, x0, x1)

sage: transvectant(f, f, 4)
Traceback (most recent call last):
...
ZeroDivisionError

sage: transvectant(f, f, 4, scale='none')
Binary quadratic given by -a3^2*x0^2 + a2*a4*x0^2 + a2*a3*x0*x1 - a1*a4*x0*x1 - a2^2*x1^2 + a1*a3*x1^2
```

The additional factors that appear when \( \text{scale}='\text{none}' \) is used can be seen if we consider the same transvectant over the rationals and compare it to the scaled version:

```python
sage: R.<a0,a1,a2,a3,a4,a5,x0,x1> = QQ[]

sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: f = AlgebraicForm(2, 5, p, x0, x1)

sage: transvectant(f, f, 4)
Binary quadratic given by 3/50*a3^2*x0^2 - 4/25*a2*a4*x0^2 + 2/5*a1*a5*x0^2 + 3/50*a2^2*x1^2 - 4/25*a1*a3*x1^2 + 2/5*a0*a4*x1^2

sage: transvectant(f, f, 4, scale='none')
Binary quadratic given by 864*a3^2*x0^2 - 2304*a2*a4*x0^2 + 576*a1*a5*x0^2 + 576*a2*a3*x0*x1 - 3456*a1*a4*x0*x1 + 28800*a0*a5*x0*x1 + 864*a2^2*x1^2 - 2304*a1*a3*x1^2 + 5760*a0*a4*x1^2
```

If the forms are given as inhomogeneous polynomials, the homogenisation might fail if the polynomial ring has multiple variables. You can circumvent this by making sure the base ring of the polynomial has only one variable:
3.2.2 Reconstruction of Algebraic Forms

This module reconstructs algebraic forms from the values of their invariants. Given a set of (classical) invariants, it returns a form that attains this values as invariants (up to scaling).

AUTHORS:

• Jesper Noordsij (2018-06): initial version

sage.rings.invariants.reconstruction.binary_cubic_coefficients_from_invariants(discriminant, invariant_choice='default')

Reconstruct a binary cubic from the value of its discriminant.

INPUT:

• discriminant – The value of the discriminant of the binary cubic.

• invariant_choice – The type of invariants provided. The accepted options are 'discriminant' and 'default', which are the same. No other options are implemented.

OUTPUT:

A set of coefficients of a binary cubic, whose discriminant is equal to the given discriminant up to a scaling.

EXAMPLES:

sage: from sage.rings.invariants.reconstruction import binary_cubic_coefficients_from_invariants
sage: coeffs = binary_cubic_coefficients_from_invariants(1)
sage: coeffs
(0, 1, -1, 0)
sage: R.<x> = QQ[]
sage: R(coeffs).discriminant()
1

The two non-equivalent cubics $x^3$ and $x^2z$ with discriminant 0 can’t be distinguished based on their discriminant, hence an error is raised:

sage: binary_cubic_coefficients_from_invariants(0)
Traceback (most recent call last):
...  
ValueError: no unique reconstruction possible for binary cubics with a double root.
Reconstruct a binary quadratic from the value of its discriminant.

**INPUT:**
- `discriminant` – The value of the discriminant of the binary quadratic.
- `invariant_choice` – The type of invariants provided. The accepted options are 'discriminant' and 'default', which are the same. No other options are implemented.

**OUTPUT:**
A set of coefficients of a binary quadratic, whose discriminant is equal to the given discriminant up to a scaling.

**EXAMPLES:**

```python
sage: from sage.rings.invariants.reconstruction import binary_quadratic_coefficients_from_invariants
sage: quadratic = invariant_theory.binary_form_from_invariants(2, [24]) # indirect doctest
sage: quadratic
Binary quadratic with coefficients (1, -6, 0)
sage: quadratic.discriminant()
24
sage: binary_quadratic_coefficients_from_invariants(0)
(1, 0, 0)
```

Reconstruct a binary quintic from the values of its (Clebsch) invariants.

**INPUT:**
- `invariants` – A list or tuple of values of the three or four invariants. The default option requires the Clebsch invariants $A$, $B$, $C$ and $R$ of the binary quintic.
- `K` – The field over which the quintic is defined.
- `invariant_choice` – The type of invariants provided. The accepted options are 'clebsch' and 'default', which are the same. No other options are implemented.
- `scaling` – How the coefficients should be scaled. The accepted values are 'none' for no scaling, 'normalized' to scale in such a way that the resulting coefficients are independent of the scaling of the input invariants and 'coprime' which scales the input invariants by dividing them by their gcd.

**OUTPUT:**
A set of coefficients of a binary quintic, whose invariants are equal to the given invariants up to a scaling.

**EXAMPLES:**
First we check the general case, where the invariant $M$ is non-zero:
```python
sage: R.<x0, x1> = QQ[]
sage: p = 3*x1^5 + 6*x1^4*x0 + 3*x1^3*x0^2 + 4*x1^2*x0^3 - 5*x1*x0^4 + 4*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: invs = quintic.clebsch_invariants(as_tuple=True)
sage: reconstructed = invariant_theory.binary_form_from_invariants(5, invs, variables=quintic.variables()) # indirect doctest

We can see that the invariants of the reconstructed form match the ones of the original form by scaling the invariants $B$ and $C$:

```python
sage: scale = invs[0]/reconstructed.A_invariant()
True
True
```

If we compare the form obtained by this reconstruction to the one found by letting the covariants $\alpha$ and $\beta$ be the coordinates of the form, we find the forms are the same up to a power of the determinant of $\alpha$ and $\beta$:

```python
sage: alpha = quintic.alpha_covariant()
sage: beta = quintic.beta_covariant()
sage: g = matrix([[alpha(x0=1,x1=0),alpha(x0=0,x1=1)],[beta(x0=1,x1=0),beta(x0=0,...,x1=1)]]^-1
sage: transformed = tuple([g.determinant()^-5*x for x in quintic.transformed(g).coeffs()])
```

This can also be seen by computing the $\alpha$ covariant of the obtained form:

```python
sage: reconstructed.alpha_covariant().coefficient(x1)
0
sage: reconstructed.alpha_covariant().coefficient(x0) != 0
True
```

If the invariant $M$ vanishes, then the coefficients are computed in a different way:

```python
sage: [A,B,C] = [3,1,2]
sage: M = 2*A*B - 3*C
sage: M
0
sage: from sage.rings.invariants.reconstruction import binary_quintic_coefficients_from_invariants
sage: reconstructed = binary_quintic_coefficients_from_invariants([A,B,C])
```

(continues on next page)
Several special cases:

```
sage: newform = sum([reconstructed[i]*x0^i*x1^(5-i) for i in range(6)])
sage: newquintic = invariant_theory.binary_quintic(newform, x0, x1)
sage: scale = 3/newquintic.A_invariant()
sage: [3, newquintic.B_invariant()*scale^2, newquintic.C_invariant()*scale^3]
[3, 1, 2]
```

For fields of characteristic 2, 3 or 5, there is no reconstruction implemented. This is part of trac ticket #26786:

```
sage: binary_quintic_coefficients_from_invariants([3,1,2], K=GF(5))
Traceback (most recent call last):
...  
NotImplementedError: no reconstruction of binary quintics implemented for fields of characteristic 2, 3 or 5
```

### 3.3 Educational Versions of Groebner Basis Related Algorithms

#### 3.3.1 Educational versions of Groebner basis algorithms

Following [BW1993], the original Buchberger algorithm (algorithm GROEBNER in [BW1993]) and an improved version of Buchberger’s algorithm (algorithm GROEBNERNEW2 in [BW1993]) are implemented.

No attempt was made to optimize either algorithm as the emphasis of these implementations is a clean and easy presentation. To compute a Groebner basis most efficiently in Sage, use the `MPolynomialIdeal.groebner_basis()` method on multivariate polynomial objects instead.

**Note:** The notion of ‘term’ and ‘monomial’ in [BW1993] is swapped from the notion of those words in Sage (or the other way around, however you prefer it). In Sage a term is a monomial multiplied by a coefficient, while in [BW1993]
a monomial is a term multiplied by a coefficient. Also, what is called LM (the leading monomial) in Sage is called HT (the head term) in [BW1993].

EXAMPLES:

Consider Katsura-6 with respect to a degrevlex ordering.

```
sage: from sage.rings.polynomial.toy_buchberger import *
sage: P.<a,b,c,e,f,g,h,i,j,k> = PolynomialRing(GF(32003))
sage: I = sage.rings.ideal.Katsura(P, 6)
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))  
sage: I = sage.rings.ideal.Katsura(P)  
// sage... ideal
sage: I  
Ideal (a + 2*b + 2*c - 1, a^2 + 2*b^2 + 2*c^2 - a, 2*a*b + 2*b*c - b) of Multivariate Polynomial Ring in a, b, c over Finite Field of size 127
sage: buchberger(I)  
# random
(a + 2*b + 2*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([-5*b^2 - 6*b*c - 6*c^2 + b + 2*c])
(a^2 + 2*b^2 + 2*c^2 - a, a + 2*b + 2*c - 1) => 0
G: set([-5*b^2 - 6*b*c - 6*c^2 + b + 2*c])
(a + 2*b + 2*c - 1, 2*a*b + 2*b*c - b) => -5*b^2 - 6*b*c - 6*c^2 + b + 2*c
G: set([-5*b^2 - 6*b*c - 6*c^2 + b + 2*c])
(2*a*b + 2*b*c - b, a + 2*b + 2*c - 1) => 0
G: set([-5*b^2 - 6*b*c - 6*c^2 + b + 2*c])
```

(continues on next page)
(2*a*b + 2*b*c - b, -5*b*c - 6*c^2 - 63*b + 2*c) => -22*c^3 + 24*c^2 - 60*b - 62*c
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 +
-2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 +
-2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 +
-2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 +
-2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 +
-2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 +
-2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 +
-2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 +
-2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 +
-2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 +
-2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 +
-2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

15 reductions to zero.
The original Buchberger algorithm performs 15 useless reductions to zero for this example:

```sage
sage: gb = buchberger(I)
...
15 reductions to zero.
```

The ‘improved’ Buchberger algorithm in contrast only performs 1 reduction to zero:

```sage
sage: gb = buchberger_improved(I)
...
1 reductions to zero.
sage: sorted(gb)
[a + 2*b + 2*c - 1, b*c + 52*c^2 + 38*b + 25*c, b^2 - 26*c^2 - 51*b + 51*c, c^3 +
 → -22*c^2 - 55*b + 49*c]
```

AUTHORS:
- Marshall Hampton (2009-07-08): some doctest additions

```sage
sage.rings.polynomial.toy_buchberger.LCM(f, g)
sage.rings.polynomial.toy_buchberger.LM(f)
sage.rings.polynomial.toy_buchberger.LT(f)
sage.rings.polynomial.toy_buchberger.buchberger(F)
```

Compute a Groebner basis using the original version of Buchberger’s algorithm as presented in [BW1993], page 214.

**INPUT:**
- $F$ – an ideal in a multivariate polynomial ring

**OUTPUT:** a Groebner basis for $F$

**Note:** The verbosity of this function may be controlled with a `set_verbose()` call. Any value $\geq 1$ will result in this function printing intermediate bases.

**EXAMPLES:**

```sage
sage: from sage.rings.polynomial.toy_buchberger import buchberger
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: I = R.ideal([x^2 - z - 1, z^2 - y - 1, x*y^2 - x - 1])
sage: set_verbose(0)
sage: gb = buchberger(I)
sage: gb.is_groebner()
True
sage: gb.ideal() == I
True
```

```sage
sage.rings.polynomial.toy_buchberger.buchberger_improved(F)
```

Compute a Groebner basis using an improved version of Buchberger’s algorithm as presented in [BW1993], page 232.
This variant uses the Gebauer-Moeller Installation to apply Buchberger’s first and second criterion to avoid useless pairs.

**INPUT:**

- F – an ideal in a multivariate polynomial ring

**OUTPUT:** a Groebner basis for F

**Note:** The verbosity of this function may be controlled with a `set_verbose()` call. Any value \( \geq 1 \) will result in this function printing intermediate Groebner bases.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.toy_buchberger import buchberger_improved
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: set_verbose(0)
sage: sorted(buchberger_improved(R.ideal([x^4 - y - z, x*y*z - 1])))
[x*y*z - 1, x^3 - y^2*z - y*z^2, y^3*z^2 + y^2*z^3 - x^2]
```

`sage.rings.polynomial.toy_buchberger.inter_reduction(Q)`

Compute inter-reduced polynomials from a set of polynomials.

**INPUT:**

- Q – a set of polynomials

**OUTPUT:** if \( Q \) is the set \( (f_1, ..., f_n) \), this method returns \( (g_1, ..., g_s) \) such that:

- \( < f_1, ..., f_n > = < g_1, ..., g_s > \)
- \( LM(g_i) \neq LM(g_j) \) for all \( i \neq j \)
- \( LM(g_i) \) does not divide \( m \) for all monomials \( m \) of \( \{g_1, ..., g_{i-1}, g_{i+1}, ..., g_s\} \)
- \( LC(g_i) = 1 \) for all \( i \).

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.toy_buchberger import inter_reduction
sage: inter_reduction(set())
set()
```

```python
sage: P.<x,y> = QQ[]
sage: reduced = inter_reduction(set([x^2 - 5*y^2, x^3]))
sage: reduced == set([x*y^2, x^2 - 5*y^2])
True
sage: reduced == inter_reduction(set([2*(x^2 - 5*y^2), x^3]))
True
```

`sage.rings.polynomial.toy_buchberger.select(P)`

Select a polynomial using the normal selection strategy.

**INPUT:**

- P – a list of critical pairs

**OUTPUT:** an element of P

**EXAMPLES:**
sage: from sage.rings.polynomial.toy_buchberger import select
sage: R.<x,y,z> = PolynomialRing(QQ, order='lex')

sage: ps = [x^3 - z - 1, z^3 - y - 1, x^5 - y - 2]
sage: pairs = [[ps[i], ps[j]] for i in range(3) for j in range(i+1, 3)]
sage: select(pairs)
[x^3 - z - 1, -y + z^3 - 1]

sage.rings.polynomial.toy_buchberger.spol(f, g)

Compute the S-polynomial of \( f \) and \( g \).

INPUT:

- \( f, g \) – polynomials

OUTPUT: the S-polynomial of \( f \) and \( g \)

EXAMPLES:

sage: R.<x,y,z> = PolynomialRing(QQ)
sage: spol(x^2 - z - 1, z^2 - y - 1)
x^2*y - z^3 + x^2 - z^2

sage.rings.polynomial.toy_buchberger.update(G, B, h)

Update \( G \) using the set of critical pairs \( B \) and the polynomial \( h \) as presented in [BW1993], page 230. For this, Buchberger’s first and second criterion are tested.

This function implements the Gebauer-Moeller Installation.

INPUT:

- \( G \) – an intermediate Groebner basis
- \( B \) – a set of critical pairs
- \( h \) – a polynomial

OUTPUT: a tuple of

- an intermediate Groebner basis
- a set of critical pairs

EXAMPLES:

sage: G, B = update(set(), set(), x*y*z)

3.3.2 Educational versions of Groebner basis algorithms: triangular factorization

In this file is the implementation of two algorithms in [Laz1992].

The main algorithm is Triangular; a secondary algorithm, necessary for the first, is ElimPolMin. As per Lazard’s formulation, the implementation works with any term ordering, not only lexicographic.
Lazard does not specify a few of the subalgorithms implemented as the functions

- \texttt{is\_triangular},
- \texttt{is\_linearly\_dependent}, and
- \texttt{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.

\textbf{AUTHORS:}

- John Perry (2009-02-24): initial version, but some words of documentation were stolen shamelessly from Martin Albrecht’s \texttt{toy\_buchberger.py}.

\begin{verbatim}

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:

- \texttt{polys} - a list/tuple of polynomials

OUTPUT:

A matrix \(M\) of the coefficients of \(polys\).

EXAMPLES:

\begin{verbatim}
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]
\end{verbatim}

Note: This function may be merged with \texttt{sage.rings.polynomial.multi\_polynomial\_sequence.PolynomialSequence\_generic.coefficient\_matrix()} in the future.

\end{verbatim}

\begin{verbatim}

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 [Laz1992].

INPUT:

- \texttt{B} - a list/tuple of polynomials or a multivariate polynomial ideal
- \texttt{n} - the variable to check (see above) (default: \(-1\))

EXAMPLES:

\end{verbatim}

\end{verbatim}
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 \( \mathbb{R}^n \), where \( n \) is the number of distinct monomials in polys. There is a zero vector iff there is a linear dependence among polys.

The case where polys=[] is considered to be not linearly dependent.

INPUT:

• polys - a list/tuple of polynomials

OUTPUT:

True if the elements of polys are linearly dependent; False otherwise.

EXAMPLES:

sage: from sage.rings.polynomial.toy_variety import is_linearly_dependent
sage: R.<x,y> = PolynomialRing(QQ)
sage: B = [x^2 + 1, y^2 + 1, x*y + 1]
sage: is_linearly_dependent(B + [p])
True
sage: p = x*B[0]
sage: is_linearly_dependent(B + [p])
False
sage: is_linearly_dependent([])
False

sage.rings.polynomial.toy_variety.is_triangular(B)
Check whether the basis B of an ideal is triangular. That is: check whether the largest variable in B[i] with respect to the ordering of the base ring R is R.gens()[i].

The algorithm is based on the definition of a triangular basis, given by Lazard in 1992 in [Laz1992].

INPUT:

• B - a list/tuple of polynomials or a multivariate polynomial ideal

OUTPUT:

True if the basis is triangular; False otherwise.

EXAMPLES:
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 \( n == \text{len}(\text{polys}) \), returns \([a[0],a[1],...,a[n-1]]\) such that \( p == a[0]*\text{poly}[0] + ... + a[n-1]*\text{poly}[n-1] \).

EXAMPLES:

sage: from sage.rings.polynomial.toy_variety import linear_representation
sage: R.<x,y> = PolynomialRing(GF(32003))
sage: B = [x^2 + 1, y^2 + 1, x*y + 1]
sage: linear_representation(p, B)
[3, 32001, 1]

sage.rings.polynomial.toy_variety.triangular_factorization(B, n=-1)
Compute the triangular factorization of the Groebner basis \( B \) of an ideal.

This will not work properly if \( B \) is not a Groebner basis!

The algorithm used is that described in a 1992 paper by Daniel Lazard [Laz1992]. It is not necessary for the term ordering to be lexicographic.

INPUT:

- B - a list/tuple of polynomials or a multivariate polynomial ideal
- n - the recursion parameter (default: -1)

OUTPUT:

A list \( T \) of triangular sets \( T_0, T_1, \) etc.

EXAMPLES:
sage: set_verbose(0)
sage: from sage.rings.polynomial.toy_variety import triangular_factorization
sage: R.<x,y,z> = PolynomialRing(GF(32003))
sage: p1 = x^2*(x-1)^3*y^2*(z-3)^3
sage: p2 = z^2 - z
sage: p3 = (x-2)^2*(y-1)^3
sage: I = R.ideal(p1,p2,p3)
sage: triangular_factorization(I.groebner_basis())
[[x^2 - 4*x + 4, y, z],
 [x^5 - 3*x^4 + 3*x^3 - x^2, y - 1, z],
 [x^2 - 4*x + 4, y, z - 1],
 [x^5 - 3*x^4 + 3*x^3 - x^2, y - 1, z - 1]]

3.3.3 Educational version of the $d$-Groebner basis algorithm over PIDs

No attempt was made to optimize this algorithm as the emphasis of this implementation is a clean and easy presentation.

**Note:** The notion of ‘term’ and ‘monomial’ in [BW1993] is swapped from the notion of those words in Sage (or the other way around, however you prefer it). In Sage a term is a monomial multiplied by a coefficient, while in [BW1993] a monomial is a term multiplied by a coefficient. Also, what is called LM (the leading monomial) in Sage is called HT (the head term) in [BW1993].

**EXAMPLES:**

sage: from sage.rings.polynomial.toy_d_basis import d_basis

First, consider an example from arithmetic geometry:

sage: A.<x,y> = PolynomialRing(ZZ, 2)
sage: B.<X,Y> = PolynomialRing(Rationals(),2)
sage: f = -y^2 - y + x^3 + 7*x + 1
sage: fx = f.derivative(x)
sage: fy = f.derivative(y)
sage: I = B.ideal([B(f),B(fx),B(fy)])
sage: I.groebner_basis()
[1]

Since the output is 1, we know that there are no generic singularities.

To look at the singularities of the arithmetic surface, we need to do the corresponding computation over $\mathbb{Z}$:

sage: I = A.ideal([f,fx,fy])
sage: gb = d_basis(I); gb
[x - 2020, y - 11313, 22627]
sage: gb[-1].factor()
11^3 * 17

This Groebner Basis gives a lot of information. First, the only fibers (over $\mathbb{Z}$) that are not smooth are at 11 = 0, and 17 = 0. Examining the Groebner Basis, we see that we have a simple node in both the fiber at 11 and at 17. From the factorization, we see that the node at 17 is regular on the surface (an $I_1$ node), but the node at 11 is not. After blowing up this non-regular point, we find that it is an $I_3$ node.

Another example. This one is from the Magma Handbook:
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: gb = [R(f) for f in gb if R(f)]; gb
[x^2 - 1, x*z + 2*y, 2*x - y*z, y^2 - 1, z^2, 2*z]
```

A third example is also from the Magma Handbook. This example shows how one can use Groebner bases over the integers to find the primes modulo which a system of equations has a solution, when the system has no solutions over the rationals.

We first form a certain ideal $I$ in $\mathbb{Z}[x, y, z]$, and note that the Groebner basis of $I$ over $\mathbb{Q}$ contains 1, so there are no solutions over $\mathbb{Q}$ or an algebraic closure of it (this is not surprising as there are 4 equations in 3 unknowns).

```
sage: P.<x, y, z> = PolynomialRing(IntegerRing(), 3, order='degrevlex')
sage: I = ideal( x^2 - 3*y, y^3 - x*y, z^3 - x, x^4 - y*z + 1 )
sage: I.change_ring(P.change_ring(RationalField())).groebner_basis()
[1]
```

However, when we compute the Groebner basis of $I$ (defined over $\mathbb{Z}$), we note that there is a certain integer in the ideal which is not 1:

```
sage: gb = d_basis(I); gb
[z ..., y ..., x ..., 282687803443]
```

Now for each prime $p$ dividing this integer 282687803443, the Groebner basis of $I$ modulo $p$ will be non-trivial and will thus give a solution of the original system modulo $p$.

```
sage: factor(282687803443)
101 * 103 * 27173681
sage: I.change_ring( P.change_ring( GF(101) ) ).groebner_basis()
[z - 33, y + 48, x + 19]
sage: I.change_ring( P.change_ring( GF(103) ) ).groebner_basis()
[z - 18, y + 8, x + 39]
sage: I.change_ring( P.change_ring( GF(27173681) ) ).groebner_basis()
[z + 10380032, y + 3186055, x - 536027]
```

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)
sage.rings.polynomial.toy_d_basis.LM(f)
sage.rings.polynomial.toy_d_basis.d_basis(F, strat=True)

Return the \(d\)-basis for the Ideal \(F\) as defined in [BW1993].

INPUT:

- \(F\) – an ideal
- \(\text{strat}\) – use update strategy (default: True)

EXAMPLES:

```python
sage: from sage.rings.polynomial.toy_d_basis import d_basis
tsage: A.<x,y> = PolynomialRing(ZZ, 2)
tsage: f = -y^2 - y + x^3 + 7*x + 1
tsage: fx = f.derivative(x)
tsage: fy = f.derivative(y)
tsage: I = A.ideal([f,fx,fy])
tsage: gb = d_basis(I); gb
[x - 2020, y - 11313, 22627]
```

sage.rings.polynomial.toy_d_basis.gpol(g1, g2)

Return the G-Polynomial of \(g_1\) and \(g_2\).

Let \(a_i t_i\) be \(LT(g_i)\), \(a = a_i * c_i + a_j * c_j\) with \(a = GCD(a_i, a_j)\), and \(s_i = t_i t\) 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

EXAMPLES:

```python
sage: from sage.rings.polynomial.toy_d_basis import gpol
tsage: P.<x, y, z> = PolynomialRing(IntegerRing(), 3, order='lex')
tsage: f = x^2 - 1
tsage: g = 2*x*y - z
tsage: gpol(f,g)
x^2*y - y
```

sage.rings.polynomial.toy_d_basis.select(P)

The normal selection strategy.

INPUT:

- \(P\) – a list of critical pairs

OUTPUT:

an element of \(P\)

EXAMPLES:

```python
sage: from sage.rings.polynomial.toy_d_basis import select
tsage: A.<x,y> = PolynomialRing(ZZ, 2)
tsage: f = -y^2 - y + x^3 + 7*x + 1
tsage: fx = f.derivative(x)
tsage: fy = f.derivative(y)
```

(continues on next page)
sage: G = [f, fx, fy]
sage: B = set((f1, f2) for f1 in G for f2 in G if f1 != f2)
sage: select(B)
(-2*y - 1, 3*x^2 + 7)

sage.rings.polynomial.toy_d_basis.spol(g1, g2)
Return the S-Polynomial of g_1 and g_2.

Let $a_it_i$ be $LT(g_i)$, $b_i = a_i/a$, 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: $b_1s_1g_1 - b_2s_2g_2$.

INPUT:

- $g1$ – polynomial
- $g2$ – polynomial

EXAMPLES:

sage: from sage.rings.polynomial.toy_d_basis import spol
sage: P.<x, y, z> = PolynomialRing(IntegerRing(), 3, order='lex')
sage: f = x^2 - 1
sage: g = 2*x*y - z
sage: spol(f, g)
x*z - 2*y

sage.rings.polynomial.toy_d_basis.update(G, B, h)
Update G using the list of critical pairs B and the polynomial h as presented in [BW1993], page 230. For this, Buchberger’s first and second criterion are tested.

This function uses the Gebauer-Moeller Installation.

INPUT:

- G – an intermediate Groebner basis
- B – a list of critical pairs
- h – a polynomial

OUTPUT:

G, B where G and B are updated

EXAMPLES:

sage: from sage.rings.polynomial.toy_d_basis import update
sage: A.<x, y> = PolynomialRing(IntegerRing(), 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))
4.1 Univariate Skew Polynomials

This module provides the `SkewPolynomial`, which constructs a single univariate skew polynomial over commutative base rings and an automorphism over the base ring. Skew polynomials are non-commutative and so principal methods such as gcd, lcm, monic, multiplication, and division are given in left and right forms.

The generic implementation of dense skew polynomials is `SkewPolynomial_generic_dense`. The classes `ConstantSkewPolynomialSection` and `SkewPolynomialBaseringInjection` handle conversion from a skew polynomial ring to its base ring and vice versa respectively.

Warning: The current semantics of `__call__()` are experimental, so a warning is thrown when a skew polynomial is evaluated for the first time in a session. See the method documentation for details.

AUTHORS:
- Xavier Caruso (2012-06-29): initial version
- Arpit Merchant (2016-08-04): improved docstrings, fixed doctests and refactored classes and methods
- Johan Rosenkilde (2016-08-03): changes for bug fixes, docstring and doctest errors

class `sage.rings.polynomial.skew_polynomial_element.ConstantSkewPolynomialSection`

Bases: `sage.categories.map.Map`

Representation of the canonical homomorphism from the constants of a skew polynomial ring to the base ring.

This class is necessary for automatic coercion from zero-degree skew polynomial ring into the base ring.

EXAMPLES:

```python
sage: from sage.rings.polynomial.skew_polynomial_element import ConstantSkewPolynomialSection
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: m = ConstantSkewPolynomialSection(S, R); m
Generic map:
  From: Skew Polynomial Ring in x over Univariate Polynomial Ring in t over Rational Field twisted by t |--> t + 1
  To:   Univariate Polynomial Ring in t over Rational Field
```

class `sage.rings.polynomial.skew_polynomial_element.SkewPolynomial`

Bases: `sage.structure.element.AlgebraElement`
Abstract base class for skew polynomials.
This class must be inherited from and have key methods overridden.

**Definition**

Let $R$ be a commutative ring equipped with an automorphism $\sigma$.

Then, a skew polynomial is given by the equation:

\[ F(X) = a_n X^n + \cdots + a_0, \]

where the coefficients $a_i \in R$ and $X$ is a formal variable.

Addition between two skew polynomials is defined by the usual addition operation and the modified multiplication is defined by the rule $Xa = \sigma(a)X$ for all $a \in R$. Skew polynomials are thus non-commutative and the degree of a product is equal to the sum of the degrees of the factors.

Let $a$ and $b$ be two skew polynomials in the same ring $S$. The left (resp. right) euclidean division of $a$ by $b$ is a couple $(q, r)$ of elements in $S$ such that

- $a = qb + r$ (resp. $a = bq + r$)
- the degree of $r$ is less than the degree of $b$

$q$ (resp. $r$) is called the quotient (resp. the remainder) of this euclidean division.

**Properties**

Keeping the previous notation, if the leading coefficient of $b$ is a unit (e.g. if $b$ is monic) then the quotient and the remainder in the right euclidean division exist and are unique.

The same result holds for the left euclidean division if in addition the twist map defining the skew polynomial ring is invertible.

**Evaluation**

The value of a given a skew polynomial $p(x) = \sum_{i=0}^{d} a_i x^i$ at $r$ is calculated using the formula:

\[ p(r) = \sum_{i=0}^{d} a_i \sigma^i(r) \]

where $\sigma$ is the base ring automorphism. This is called the operator evaluation method.

**EXAMPLES:**

We illustrate some functionalities implemented in this class.

We create the skew polynomial ring:

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]; S
Skew Polynomial Ring in x over Univariate Polynomial Ring in t over Integer Ring twisted by t |--> t + 1
```

and some elements in it:
 Ring operations are supported:

```
sage: a + b
x^2 + (t + 2)*x + t^2 + t + 1
sage: a - b
-x^2 - t*x - t^2 + t + 1
sage: a * b
x^3 + (2*t + 3)*x^2 + (2*t^2 + 4*t + 2)*x + t^3 + t^2
sage: b * a
x^3 + (2*t + 4)*x^2 + (2*t^2 + 3*t + 2)*x + t^3 + t^2
sage: a * b == b * a
False
sage: b^2
x^4 + (2*t + 4)*x^3 + (3*t^2 + 7*t + 6)*x^2 + (2*t^3 + 4*t^2 + 3*t + 1)*x + t^4
sage: b^2 == b*b
True
```

Sage also implements arithmetic over skew polynomial rings. You will find below a short panorama:

```
sage: q, r = c.right_quo_rem(b)
sage: q
x - t - 2
sage: r
3*t*x + t^3 + 2*t^2
sage: c == q*b + r
True
```

The operators // and % give respectively the quotient and the remainder of the right euclidean division:

```
sage: q == c // b
True
sage: r == c % b
True
```

Left euclidean division won’t work over our current $S$ because Sage can’t invert the twist map:

```
sage: q, r = c.left_quo_rem(b)
Traceback (most recent call last):
... NotImplementedError: inversion of the twist map Ring endomorphism of Univariate Polynomial Ring in t over Integer Ring
   Defn: t |--> t + 1
```

Here we can see the effect of the operator evaluation compared to the usual polynomial evaluation:

```
sage: a = x^2
sage: a(t)
t + 2
```

4.1. Univariate Skew Polynomials
Here is a working example over a finite field:

```python
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = x^4 + (4*t + 1)*x^3 + (t^2 + 3*t + 3)*x^2 + (3*t^2 + 2*t + 2)*x + (3*t^2 + 3*t + 1)
sage: b = (2*t^2 + 3)*x^2 + (3*t^2 + 1)*x + 4*t + 2
sage: q, r = a.left_quo_rem(b)
sage: q
(4*t^2 + t + 1)*x^2 + (2*t^2 + 2*t + 2)*x + 2*t^2 + 4*t + 3
sage: r
(t + 2)*x + 3*t^2 + 2*t + 4
sage: a == b*q + r
True
```

Once we have euclidean divisions, we have for free \( \gcd \) and \( \lcm \) (at least if the base ring is a field):

```python
sage: a = (x + t) * (x + t^2)^2
sage: b = (x + t) * (t*x + t + 1) * (x + t^2)
```

The \( \lcm \) has the following meaning: given skew polynomials \( a \) and \( b \), their \( \lcm \) is the least degree polynomial \( c = ua = vb \) for some skew polynomials \( u, v \). Such a \( c \) always exist if the base ring is a field:

```python
sage: c = a.left_lcm(b); c
x^5 + (4*t^2 + t + 3)*x^4 + (3*t^2 + 4*t)*x^3 + 2*t^2*x^2 + (2*t^2 + t)*x + 4*t^2
```

The right \( \lcm \) is defined similarly as the least degree polynomial \( c = au = bv \) for some \( u, v \):

```python
sage: d = a.right_lcm(b); d
x^5 + (t^2 + 1)*x^4 + (3*t^2 + 3*t + 3)*x^3 + (3*t^2 + t + 2)*x^2 + (4*t^2 + t)*x + 4*t + 4
```

See also:

- `sage.rings.polynomial.skew_polynomial_ring`
- `sage.rings.polynomial.skew_polynomial_ring_constructor`

`base_ring()`

Return the base ring of `self`.

**EXAMPLES:**
```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x', sigma]
sage: a = S.random_element()
sage: a.base_ring()
Univariate Polynomial Ring in t over Integer Ring
sage: a.base_ring() is R
True
```

**change_variable_name**(var)

Change the name of the variable of self.

This will create the skew polynomial ring with the new name but same base ring and twist map. The returned skew polynomial will be an element of that skew polynomial ring.

**INPUT:**

- var – the name of the new variable

**EXAMPLES:**

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x', sigma]
sage: a = x^3 + (2*t + 1)*x + t^2 + 3*t + 5
sage: b = a.change_variable_name('y'); b
y^3 + (2*t + 1)*y + t^2 + 3*t + 5
```

Note that a new parent is created at the same time:

```python
sage: b.parent()
Skew Polynomial Ring in y over Univariate Polynomial Ring in t over Integer Ring
   twisted by t |--> t + 1
```

**coefficients**(sparse=True)

Return the coefficients of the monomials appearing in self.

If sparse=True (the default), return only the non-zero coefficients. Otherwise, return the same value as self.list().

**Note:** This should be overridden in subclasses.

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x', sigma]
sage: a = 1 + x^4 + (t+1)*x^2 + t^2
sage: a.coefficients() [t^2 + 1, t + 1, 1]
sage: a.coefficients(sparse=False) [t^2 + 1, 0, t + 1, 0, 1]
```

**conjugate**(n)

Return self conjugated by \(x^n\), where \(x\) is the variable of self.
The conjugate is obtained from self by applying the \( n \)-th iterate of the twist map to each of its coefficients.

INPUT:

- \( n \) – an integer, the power of conjugation

EXAMPLES:

```python
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = t*x^3 + (t^2 + 1)*x^2 + 2*t
sage: b = a.conjugate(2);
(b(t + 2)*x^3 + (t^2 + 4*t + 5)*x^2 + 2*t + 4)
sage: x^2*a == b*x^2
(True)
```

In principle, negative values for \( n \) are allowed, but Sage needs to be able to invert the twist map:

```python
sage: b = a.conjugate(-1)
Traceback (most recent call last):
...
NotImplementedError: inversion of the twist map
```

Here is a working example:

```python
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: T.<y> = k['y',Frob]
sage: u = T.random_element(); u
(2*t^2 + 3)*y^2 + (4*t^2 + t + 4)*y + 2*t^2 + 2
sage: v = u.conjugate(-1); v
(3*t^2 + t)*y^2 + (4*t^2 + 2*t + 4)*y + 3*t^2 + t + 4
sage: u*y == y*v
(True)
```

**constant_coefficient()**

Return the constant coefficient (i.e. the coefficient of term of degree 0) of self.

EXAMPLES:

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x + t^2 + 2
sage: a.constant_coefficient()
t^2 + 2
```

**degree()**

Return the degree of self.

By convention, the zero skew polynomial has degree \(-1\).

EXAMPLES:

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
```

(continues on next page)
sage: S.<x> = R['x',sigma]
sage: a = x^2 + t*x^3 + t^2*x + 1
sage: a.degree()
3
sage: S.zero().degree()
-1
sage: S(5).degree()
0

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

EXAMPLES:

sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = 1 + x^4 + (t+1)*x^2 + t^2
sage: a.exponents()
[0, 2, 4]

hamming_weight()
Return the number of non-zero coefficients of self.
This is also known as the weight, hamming weight or sparsity.

EXAMPLES:

sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = 1 + x^4 + (t+1)*x^2 + t^2
sage: a.number_of_terms()
3
sage: a.hamming_weight()
3

This is also an alias for hamming_weight:

sage: a.hamming_weight()
3

is_constant()
Return whether self is a constant polynomial.

EXAMPLES:

sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: R(2).is_constant()
True
sage: (x + 1).is_constant()
False

is_left_divisible_by(other)
Check if self is divisible by other on the left.

INPUT:

- other – a skew polynomial in the same ring as self
OUTPUT:

Return True or False.

EXAMPLES:

```python
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = x^2 + t*x + t^2 + 3
sage: b = x^3 + (t + 1)*x^2 + 1
sage: c = a*b
sage: c.is_left_divisible_by(a)
True
sage: c.is_left_divisible_by(b)
False
```

Divisibility by 0 does not make sense:

```python
sage: c.is_left_divisible_by(S(0))
Traceback (most recent call last):
  ... ZeroDivisionError: division by zero is not valid
```

**is_monic()**

Return True if this skew polynomial is monic.

The zero polynomial is by definition not monic.

EXAMPLES:

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x + t
sage: a.is_monic()
True
sage: a = 0*x
sage: a.is_monic()
False
sage: a = t*x^3 + x^4 + (t+1)*x^2
sage: a.is_monic()
True
sage: a = (t^2 + 2*t)*x^2 + x^3 + t^10*x^5
sage: a.is_monic()
False
```

**is_monomial()**

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

EXAMPLES:

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: x.is_monomial()
True
sage: (x+1).is_monomial()
False
```
\textbf{is\_monomial()} \\
Check if \texttt{self} is a monomial. \\
\textbf{is\_term()} \\
Test whether this polynomial is 1.

\section{4.1. Univariate Skew Polynomials}

\begin{itemize}
\item \texttt{sage: (x^2).is\_monomial()}
\text{True}
\item \texttt{sage: S(1).is\_monomial()}
\text{True}
\end{itemize}

The coefficient must be 1:

\begin{itemize}
\item \texttt{sage: (2*x^5).is\_monomial()}
\text{False}
\item \texttt{sage: S(t).is\_monomial()}
\text{False}
\end{itemize}

To allow a non-1 leading coefficient, use is\_term():

\begin{itemize}
\item \texttt{sage: (2*x^5).is\_term()}
\text{True}
\item \texttt{sage: S(t).is\_term()}
\text{True}
\end{itemize}

\textbf{is\_nilpotent()} \\
Check if \texttt{self} is nilpotent.

Given a commutative ring $R$ and a base ring automorphism $\sigma$ of order $n$, an element $f$ of $R[X,\sigma]$ is nilpotent if and only if all coefficients of $f^n$ are nilpotent in $R$.

\textbf{Note:} The paper “Nilpotents and units in skew polynomial rings over commutative rings” by M. Rimmer and K.R. Pearson describes the method to check whether a given skew polynomial is nilpotent. That method however, requires one to know the order of the automorphism which is not available in Sage. This method is thus not yet implemented.

\textbf{is\_one()} \\
Test whether this polynomial is 1.

\begin{itemize}
\item \texttt{sage: R.<t> = ZZ[]}
\item \texttt{sage: sigma = R.hom([t+1])}
\item \texttt{sage: S.<x> = R['x',sigma]}
\item \texttt{sage: x.is\_nilpotent()}
\text{Traceback (most recent call last):
... NotImplementedError}
\end{itemize}

\textbf{is\_right\_divisible\_by()} \texttt{(other)}
Check if \texttt{self} is divisible by \texttt{other} on the right.
INPUT:

- other – a skew polynomial in the same ring as self

OUTPUT:

Return True or False.

EXAMPLES:

```python
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = x^2 + t*x + t^2 + 3
sage: b = x^3 + (t + 1)*x^2 + 1
sage: c = a*b
sage: c.is_right_divisible_by(a)
False
sage: c.is_right_divisible_by(b)
True
```

Divisibility by 0 does not make sense:

```python
sage: c.is_right_divisible_by(S(0))
Traceback (most recent call last):
 ... ZeroDivisionError: division by zero is not valid
```

This function does not work if the leading coefficient of the divisor is not a unit:

```python
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x^2 + 2*x + t
sage: b = (t+1)*x + t^2
sage: c = a*b
sage: c.is_right_divisible_by(b)
Traceback (most recent call last):
 ... NotImplementError: the leading coefficient of the divisor is not invertible
```

**is_term()**

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

EXAMPLES:

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
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
```

If you want to test that self also has leading coefficient 1, use `is_monomial()` instead:
```python
sage: (3*x^5).is_monomial()
False
```

**is_unit()**

Return True if this skew polynomial is a unit.

When the base ring $R$ is an integral domain, then a skew polynomial $f$ is a unit if and only if degree of $f$ is 0 and $f$ is then a unit in $R$.

**Note:** The case when $R$ is not an integral domain is not yet implemented.

**EXAMPLES:**

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x + (t+1)*x^5 + t^2*x^3 - x^5
sage: a.is_unit()
False
```

**is_zero()**

Return True if self is the zero polynomial.

**EXAMPLES:**

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x + 1
sage: a.is_zero()
False
sage: b = S.zero()
sage: b.is_zero()
True
```

**leading_coefficient()**

Return the coefficient of the highest-degree monomial of self.

**EXAMPLES:**

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = (t+1)*x^5 + t^2*x^3 + x
sage: a.leading_coefficient()
t + 1
```

**left_divides (other)**

Check if self divides other on the left.

**INPUT:**

- `other` – a skew polynomial in the same ring as self

**OUTPUT:**

Return True or False.

**EXAMPLES:**

4.1. Univariate Skew Polynomials
Divisibility by 0 does not make sense:

```sage```
sage: S(0).left_divides(c)
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero is not valid```
```

**left_gcd**(other, monic=True)

Return the left gcd of self and other.

**INPUT:**

- other – a skew polynomial in the same ring as self
- monic – boolean (default: True). Return whether the left gcd should be normalized to be monic.

**OUTPUT:**

The left gcd of self and other, that is a skew polynomial \( g \) with the following property: any skew polynomial is divisible on the left by \( g \) iff it is divisible on the left by both self and other. If monic is True, \( g \) is in addition monic. (With this extra condition, it is uniquely determined.)

**Note:** Works only if two following conditions are fulfilled (otherwise left gcd do not exist in general): 1) the base ring is a field and 2) the twist map on this field is bijective.

**EXAMPLES:**

```sage```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = (x + t) * (x^2 + t*x + 1)
sage: b = 2 * (x + t) * (x^3 + (t+1)*x^2 + t^2)
sage: a.left_gcd(b)
x + t
```

Specifying monic=False, we can get a nonmonic gcd:

```sage```
sage: a.left_gcd(b,monic=False)
2*t*x + 4*t + 2
```
```

The base ring needs to be a field:

```sage```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = (x + t) * (x^2 + t*x + 1)
```
```
(continues on next page)
And the twist map needs to be bijective:

```python
sage: FR = R.fraction_field()
sage: f = FR.hom([FR(t)^2])
sage: S.<x> = FR['x',f]
sage: a = (x + t) * (x^2 + t*x + 1)
sage: b = 2 * (x + t) * (x^3 + (t+1)*x^2 + t^2)
sage: a.left_gcd(b)
Traceback (most recent call last):
  ...  
TypeError: inversion of the twist map
```

```python
left_lcm(other, monic=True)
```

Return the left lcm of self and other.

**INPUT:**
- `other` – a skew polynomial in the same ring as self
- `monic` – boolean (default: `True`). Return whether the left lcm should be normalized to be monic.

**OUTPUT:**

The left lcm of self and other, that is a skew polynomial $g$ with the following property: any skew polynomial divides $g$ on the right iff it divides both self and other on the right. If monic is True, $g$ is in addition monic. (With this extra condition, it is uniquely determined.)

**Note:** Works only if the base ring is a field (otherwise left lcm do not exist in general).

**EXAMPLES:**

```python
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = (x + t^2) * (x + t)
sage: b = 2 * (x^2 + t + 1) * (x * t)
sage: c = a.left_lcm(b); c
x^5 + (2*t^2 + t + 4)*x^4 + (3*t^2 + 4)*x^3 + (3*t^2 + 3*t + 2)*x^2 + (t^2 + t + 2)*x
sage: c.is_right_divisible_by(a)
True
sage: c.is_right_divisible_by(b)
True
sage: a.degree() + b.degree() == c.degree() + a.right_gcd(b).degree()
True
```

Specifying monic=False, we can get a nonmonic gcd:

```python
```
The base ring needs to be a field:

```python
sage: R.<t> = QQ[
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = (x + t^2) * (x + t)
sage: b = 2 * (x^2 + t + 1) * (x * t)
sage: a.left_lcm(b)
Traceback (most recent call last):
  ...TypeError: the base ring must be a field
```

**left_mod**(other)

Return the remainder of left division of self by other.

**EXAMPLES:**

```python
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = 1 + t*x^2
sage: b = x + 1
sage: a.left_mod(b)
2*t^2 + 4*t
```

**left_monic()**

Return the unique monic skew polynomial \( m \) which divides self on the left and has the same degree.

Given a skew polynomial \( p \) of degree \( n \), its left monic is given by \( m = p \sigma^{-n}(1/k) \), where \( k \) is the leading coefficient of \( p \), i.e. by the appropriate scalar multiplication on the right.

**EXAMPLES:**

```python
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = (3*t^2 + 3*t + 2)*x^3 + (2*t^2 + 3)*x^2 + (4*t^2 + t + 4)*x + 2*t^2 + 2
sage: b = a.left_monic(); b
x^3 + (4*t^2 + 3*t)*x^2 + (4*t + 2)*x + 2*t^2 + 4*t + 3
```

Check list:

```python
sage: b.degree() == a.degree()
True
sage: b.is_left_divisible_by(a)
True
sage: twist = S.twist_map(-a.degree())
sage: a == b * twist(a.leading_coefficient())
True
```

Note that \( b \) does not divide \( a \) on the right:
This function does not work if the leading coefficient is not a unit:

```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = t*x
sage: a.left_monic()
Traceback (most recent call last):
  ...  
NotImplementedError: the leading coefficient is not a unit
```

```
def left_xgcd(other, monic=True):
    Return the left gcd of self and other along with the coefficients for the linear combination.
    If a is self and b is other, then there are skew polynomials u and v such that g = au + bv, where g is
    the left gcd of a and b. This method returns (g, u, v).

    INPUT:
    • other - a skew polynomial in the same ring as self
    • monic - boolean (default: True). Return whether the left gcd should be normalized to be monic.

    OUTPUT:
    • The left gcd of self and other, that is a skew polynomial g with the following property: any skew
      polynomial is divisible on the left by g iff it is divisible on the left by both self and other. If
      monic is True, g is in addition monic. (With this extra condition, it is uniquely determined.)
    • Two skew polynomials u and v such that:
      
g = a * u + b * v,
    where s is self and b is other.

    Note: Works only if following two conditions are fulfilled (otherwise left gcd do not exist in general): 1) the base ring is a field and 2) the twist map on this field is bijective.

    EXAMPLES:
    ```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = (x + t) * (x^2 + t*x + 1)
sage: b = 2 * (x + t) * (x^3 + (t+1)*x^2 + t^2)
sage: g,u,v = a.left_xgcd(b); g
x + t
sage: a*u + b*v == g
True
```
```
Specifying monic=False, we can get a nonmonic gcd:
```
```
sage: g,u,v = a.left_xgcd(b, monic=False); g
2*t*x + 4*t + 2
sage: a*u + b*v == g
True
```
The base ring must be a field:

```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = (x + t) * (x^2 + t*x + 1)
sage: b = 2 * (x + t) * (x^3 + (t+1)*x^2 + t^2)
sage: a.left_xgcd(b)
Traceback (most recent call last):
...
TypeError: the base ring must be a field
```

And the twist map must be bijective:

```
sage: FR = R.fraction_field()
sage: f = FR.hom([FR(t)^2])
sage: S.<x> = FR['x',f]
sage: a = (x + t) * (x^2 + t*x + 1)
sage: b = 2 * (x + t) * (x^3 + (t+1)*x^2 + t^2)
sage: a.left_xgcd(b)
Traceback (most recent call last):
...
NotImplementedError: inversion of the twist map
```

```python
multi_point_evaluation(eval_pts)
```
Evaluate self at list of evaluation points.

**INPUT:**
- eval_pts – list of points at which self is to be evaluated

**OUTPUT:**
List of values of self at the eval_pts.

**Todo:** This method currently trivially calls the evaluation function repeatedly. If fast skew polynomial multiplication is available, an asymptotically faster method is possible using standard divide and conquer techniques and `sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_general.minimal_vanishing_polynomial()`.

**EXAMPLES:**

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = x + t
sage: eval_pts = [1, t, t^2]
sage: c = a.multi_point_evaluation(eval_pts); c
[t + 1, 3*t^2 + 4*t + 4, 4*t]
sage: c == [ a(e) for e in eval_pts ]
True
```

```python
number_of_terms()
```
Return the number of non-zero coefficients of self.

This is also known as the weight, hamming weight or sparsity.
EXAMPLES:

```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = 1 + x^4 + (t+1)*x^2 + t^2
sage: a.number_of_terms()
3
```

This is also an alias for `hamming_weight`:

```
sage: a.hamming_weight()
3
```

**operator_eval**(eval_pt)

Evaluate `self` at `eval_pt` by the operator evaluation method.

**INPUT:**

- `eval_pt` – element of the base ring of `self`

**OUTPUT:**

The value of the polynomial at the point specified by the argument.

**EXAMPLES:**

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: T.<x> = k['x',Frob]
sage: a = 3*t^2*x^2 + (t + 1)*x + 2
sage: a(t) #indirect test
2*t^2 + 2*t + 3
sage: a.operator_eval(t)
2*t^2 + 2*t + 3
```

Evaluation points outside the base ring is usually not possible due to the twist map:

```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = t*x + 1
sage: a.operator_eval(1/t)
Traceback (most recent call last):
...
TypeError: 1/t fails to convert into the map's domain Univariate Polynomial
˓Ring in t over Rational Field, but a `pushforward` method is not properly
˓implemented
```

**padded_list**(n=None)

Return list of coefficients of `self` up to (but not including) degree `n`.

Includes 0.

**INPUT:**

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

**EXAMPLES:** 4.1. Univariate Skew Polynomials 483
```python
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = 1 + t*x^3 + t^2*x^5
sage: a.padded_list()
[1, 0, 0, t, 0, t^2]
sage: a.padded_list(10)
[1, 0, 0, t, 0, t^2, 0, 0, 0, 0]
sage: len(a.padded_list(10))
10
sage: a.padded_list(3)
[1, 0, 0]
sage: a.padded_list(0)
[]
sage: a.padded_list(-1)
Traceback (most recent call last):
  ... ValueError: n must be at least 0
```

**prec()**

Return the precision of `self`.

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

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: x.prec()
+Infinity
```

**right_divides**(other)

Check if `self` divides `other` on the right.

**INPUT:**

- other – a skew polynomial in the same ring as `self`

**OUTPUT:**

Return `True` or `False`.

**EXAMPLES:**

```python
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = x^2 + t*x + t^2 + 3
sage: b = x^3 + (t + 1)*x^2 + 1
sage: c = a*b
sage: a.right_divides(c)
False
sage: b.right_divides(c)
True
```

Divisibility by 0 does not make sense:

```python
sage: S(0).right_divides(c)
Traceback (most recent call last):
(continues on next page)
```
ZeroDivisionError: division by zero is not valid

This function does not work if the leading coefficient of the divisor is not a unit:

```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x^2 + 2*x + t
sage: b = (t+1)*x + t^2
sage: c = a*b
sage: b.right_divides(c)
Traceback (most recent call last):
...
NotImplementedError: the leading coefficient of the divisor is not invertible
```

```
right_gcd(other, monic=True)
```

Return the right gcd of self and other.

**INPUT:**

- `other` — a skew polynomial in the same ring as self
- `monic` — boolean (default: True). Return whether the right gcd should be normalized to be monic.

**OUTPUT:**

The right gcd of self and other, that is a skew polynomial $g$ with the following property: any skew polynomial is divisible on the right by $g$ iff it is divisible on the right by both self and other. If monic is True, $g$ is in addition monic. (With this extra condition, it is uniquely determined.)

**Note:** Works only if the base ring is a field (otherwise right gcd do not exist in general).

**EXAMPLES:**

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = (x^2 + t*x + 1) * (x + t)
sage: b = 2 * (x^3 + (t+1)*x^2 + t^2) * (x + t)
sage: a.right_gcd(b)
x + t
```

Specifying `monic=False`, we can get a nonmonic gcd:

```
sage: a.right_gcd(b,monic=False)
(4*t^2 + 4*t + 1)*x + 4*t^2 + 4*t + 3
```

The base ring need to be a field:

```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = (x^2 + t*x + 1) * (x + t)
sage: b = 2 * (x^3 + (t+1)*x^2 + t^2) * (x + t)
sage: a.right_gcd(b)
```
right_lcm(other, monic=True)

Return the right lcm of self and other.

INPUT:

• other – a skew polynomial in the same ring as self
• monic – boolean (default: True). Return whether the right lcm should be normalized to be monic.

OUTPUT:

The right lcm of self and other, that is a skew polynomial $g$ with the following property: any skew polynomial divides $g$ on the left iff it divides both self and other on the left. If monic is True, $g$ is in addition monic. (With this extra condition, it is uniquely determined.)

Note: Works only if two following conditions are fulfilled (otherwise right lcm do not exist in general): 1) the base ring is a field and 2) the twist map on this field is bijective.

EXAMPLES:

```sage
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = (x + t) * (x + t^2)
sage: b = 2 * (x + t) * (x^2 + t + 1)
sage: c = a.right_lcm(b); c
x^4 + (2*t^2 + t + 2)*x^3 + (3*t^2 + 4*t + 1)*x^2 + (3*t^2 + 4*t + 1)*x + t^2
˓→ 4
sage: c.is_left_divisible_by(a)
True
sage: c.is_left_divisible_by(b)
True
sage: a.degree() + b.degree() == c.degree() + a.left_gcd(b).degree()
True
```

Specifying monic=False, we can get a nonmonic gcd:

```sage
sage: a.right_lcm(b,monic=False)
2*t*x^4 + (3*t + 1)*x^3 + (4*t^2 + 4*t + 3)*x^2 + (3*t^2 + 4*t + 2)*x + 3*t^2 + 2*t + 3
```

The base ring needs to be a field:

```sage
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = (x + t) * (x + t^2)
sage: b = 2 * (x + t) * (x^2 + t + 1)
sage: a.right_lcm(b)
Traceback (most recent call last):
...
TypeError: the base ring must be a field
```
And the twist map needs to be bijective:

```python
sage: FR = R.fraction_field()
sage: f = FR.hom([FR(t)^2])
sage: S.<x> = FR['x',f]
sage: a = (x + t) * (x + t^2)
sage: b = 2 * (x + t) * (x^2 + t + 1)
sage: a.right_lcm(b)
Traceback (most recent call last):
  ... 
NotImplementedError: inversion of the twist map 
Ring endomorphism of Fraction Field of Univariate Polynomial Ring in t over Rational Field
  Defn: t |--> t^2
```

right_mod(other)

Return the remainder of right division of self by other.

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = 1 + t*x^2
sage: b = x + 1
sage: a % b
t + 1
sage: (x^3 + x - 1).right_mod(x^2 - 1)
2*x - 1
```

right_monic()

Return the unique monic skew polynomial \( m \) which divides self on the right and has the same degree.

Given a skew polynomial \( p \) of degree \( n \), its left monic is given by \( m = (1/k) \times p \), where \( k \) is the leading coefficient of \( p \), i.e. by the appropriate scalar multiplication on the left.

**EXAMPLES:**

```python
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = (3*t^2 + 3*t + 2)*x^3 + (2*t^2 + 3)*x^2 + (4*t^2 + t + 4)*x + 2*t^2
sage: b = a.right_monic(); b
x^3 + (2*t^2 + 3*t + 4)*x^2 + (3*t^2 + 4*t + 1)*x + 2*t^2 + 4*t + 3
```

Check list:

```python
sage: b.degree() == a.degree()
True
sage: b.is_right_divisible_by(a)
True
sage: a == a.leading_coefficient() * b
True
```

Note that \( b \) does not divide \( a \) on the right:

```python
sage: a.is_left_divisible_by(b)
False
```

This function does not work if the leading coefficient is not a unit:
sage: R.<t> = QQ[]  
sage: sigma = R.hom([t+1])  
sage: S.<x> = R['x',sigma]  
sage: a = t*x  
sage: a.right_monic()  
Traceback (most recent call last):  
...  
NotImplementedError: the leading coefficient is not a unit

right_xgcd(other, monic=True)
Return the right gcd of self and other along with the coefficients for the linear combination.

If a is self and b is other, then there are skew polynomials u and v such that \( g = ua + vb \), where \( g \) is the right gcd of \( a \) and \( b \). This method returns \((g, u, v)\).

INPUT:
- other – a skew polynomial in the same ring as self
- monic – boolean (default: True). Return whether the right gcd should be normalized to be monic.

OUTPUT:
- The right gcd of self and other, that is a skew polynomial \( g \) with the following property: any skew polynomial is divisible on the right by \( g \) iff it is divisible on the right by both self and other. If monic is True, \( g \) is in addition monic. (With this extra condition, it is uniquely determined.)
- Two skew polynomials \( u \) and \( v \) such that:

\[
g = u*a + v*b
\]

where \( a \) is self and \( b \) is other.

Note: Works only if the base ring is a field (otherwise right gcd do not exist in general).

EXAMPLES:

```python
sage: k.<t> = GF(5^3)  
sage: Frob = k.frobenius_endomorphism()  
sage: S.<x> = k['x',Frob]  
sage: a = (x^2 + t*x + 1) * (x + t)  
sage: b = 2 * (x^3 + (t+1)*x^2 + t^2) * (x + t)  
sage: g,u,v = a.right_xgcd(b); g  
x + t
sage: u*a + v*b == g  
True
```

Specifying monic=False, we can get a nonmonic gcd:

```python
sage: g,u,v = a.right_xgcd(b,monic=False); g  
(4*t^2 + 4*t + 1)*x + 4*t^2 + 4*t + 3  
sage: u*a + v*b == g  
True
```

The base ring must be a field:
```python
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = (x^2 + t*x + 1) * (x + t)
sage: b = 2 * (x^3 + (t+1)*x^2 + t^2) * (x + t)
sage: a.right_xgcd(b)
Traceback (most recent call last):
...  
TypeError: the base ring must be a field
```

### shift($n$)

Return self multiplied on the right by the power $x^n$.

If $n$ is negative, terms below $x^n$ will be discarded.

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x^5 + t^4*x^4 + t^2*x^2 + t^10
sage: a.shift(0)
x^5 + t^4*x^4 + t^2*x^2 + t^10
sage: a.shift(-1)
x^4 + t^4*x^3 + t^2*x
sage: a.shift(-5)
1
sage: a.shift(2)
x^7 + t^4*x^6 + t^2*x^4 + t^10*x^2
```

One can also use the infix shift operator:

```python
sage: a >> 2
x^3 + t^4*x^2 + t^2
sage: a << 2
x^7 + t^4*x^6 + t^2*x^4 + t^10*x^2
```

### square()

Return the square of self.

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x + t; a
x + t
sage: a.square()
x^2 + (2*t + 1)*x + t^2
sage: a.square() == a*a
True
```

### variable_name()

Return the string name of the variable used in self.

**EXAMPLES:**

```python
```
```python
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x', sigma]
sage: a = x + t
sage: a.variable_name()
'x'
```

**class** `sage.rings.polynomial.skew_polynomial_element.SkewPolynomialBaseringInjection`  

**Bases:** `sage.categories.morphism.Morphism`  

Representation of the canonical homomorphism from a ring $R$ into a skew polynomial ring over $R$.

This class is necessary for automatic coercion from the base ring to the skew polynomial ring.

**See also:**

`PolynomialBaseringInjection`

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x', sigma]
sage: S.coerce_map_from(S.base_ring()) #indirect doctest
Skew Polynomial base injection morphism:
    From: Univariate Polynomial Ring in t over Rational Field
    To:   Skew Polynomial Ring in x over Univariate Polynomial Ring in t over Rational Field twisted by t |--> t + 1
```

**an_element()**  

Return an element of the codomain of the ring homomorphism.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.skew_polynomial_element import SkewPolynomialBaseringInjection
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x', Frob]
sage: m = SkewPolynomialBaseringInjection(k, k['x', Frob])
sage: m.an_element()
```

**section()**  

Return the canonical homomorphism from the constants of a skew polynomial ring to the base ring according to `self`.

**class** `sage.rings.polynomial.skew_polynomial_element.SkewPolynomial_generic_dense`  

**Bases:** `sage.rings.polynomial.skew_polynomial_element.SkewPolynomial`  

Generic implementation of dense skew polynomial supporting any valid base ring and twist map.

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

Return the coefficients of the monomials appearing in `self`.

If `sparse=True` (the default), return only the non-zero coefficients. Otherwise, return the same value as `self.list()`.

**EXAMPLES:**
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])

sage: S.<x> = R['x', sigma]

sage: a = 1 + x^4 + (t+1)*x^2 + t^2

sage: a.coefficients()
[t^2 + 1, t + 1, 1]

sage: a.coefficients(sparse=False)
[t^2 + 1, 0, t + 1, 0, 1]

degree()

Return the degree of self.

By convention, the zero skew polynomial has degree $-1$.

EXAMPLES:

sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])

sage: S.<x> = R['x', sigma]

sage: a = x^2 + t*x^3 + t^2*x + 1

sage: a.degree()
3

By convention, the degree of 0 is $-1$:

sage: S(0).degree()
-1

dict()

Return a dictionary representation of self.

EXAMPLES:

sage: R.<t> = ZZ[]

sage: sigma = R.hom([t+1])

sage: S.<x> = R['x', sigma]

sage: a = x^2012 + t*x^1006 + t^3 + 2*t

sage: a.dict()
{0: t^3 + 2*t, 1006: t, 2012: 1}

left_power_mod(exp, modulus)

Return the remainder of self**exp in the left euclidean division by modulus.

INPUT:

- `exp` - an Integer
- `modulus` - a skew polynomial in the same ring as self

OUTPUT:

Remainder of self**exp in the left euclidean division by modulus.

REMARK:

The quotient of the underlying skew polynomial ring by the principal ideal generated by modulus is in general not a ring.

As a consequence, Sage first computes exactly self**exp and then reduce it modulo modulus.

EXAMPLES:
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = x + t
sage: modulus = x^3 + t*x^2 + (t+3)*x - 2
sage: a.left_power_mod(100,modulus)
(4*t^2 + t + 1)*x^2 + (t^2 + 4*t + 1)*x + 3*t^2 + 3*t

left_quo_rem(other)
Return the quotient and remainder of the left euclidean division of self by other.

INPUT:
• other – a skew polynomial in the same ring as self

OUTPUT:
• the quotient and the remainder of the left euclidean division of this skew polynomial by other

Note: This will fail if the leading coefficient of other is not a unit or if Sage can’t invert the twist map.

EXAMPLES:

sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = (3*t^2 + 3*t + 2)*x^3 + (2*t^2 + 3)*x^2 + (4*t^2 + t + 4)*x + 2*t^2
˓→
sage: b = (3*t^2 + 4*t + 2)*x^2 + (2*t^2 + 3)*x + 2*t^2 + t + 1
sage: q,r = a.left_quo_rem(b)
sage: a == b*q + r
True

In the following example, Sage does not know the inverse of the twist map:

sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = (-2*t^2 - t + 1)*x^3 + (-t^2 + t)*x^2 + (-12*t - 2)*x - t^2 - 95*t
˓→
sage: b = x^2 + (5*t - 6)*x - 4*t^2 + 4*t - 1
sage: a.left_quo_rem(b)
Traceback (most recent call last):
  ... NotImplementedError: inversion of the twist map Ring endomorphism of
  Univariate Polynomial Ring in t over Integer Ring
  Defn: t |--> t + 1

list (copy=True)
Return a list of the coefficients of self.

EXAMPLES:

sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = 1 + x^4 + (t+1)*x^2 + t^2
(continues on next page)
Note that \( l \) is a list, it is mutable, and each call to the list method returns a new list:

\[
sage: \text{type}(l) \\
<... 'list'> \\
sage: l[0] = 5 \\
sage: a.list() \\
[t^2 + 1, 0, t + 1, 0, 1]
\]

\[\text{right\_power\_mod}(\text{exp}, \text{modulus})\]
Return the remainder of \( \text{self}^{\text{exp}} \) in the right euclidean division by \( \text{modulus} \).

**INPUT:**
- \( \text{exp} \) – an Integer
- \( \text{modulus} \) – a skew polynomial in the same ring as \( \text{self} \)

**OUTPUT:**
Remainder of \( \text{self}^{\text{exp}} \) in the right euclidean division by \( \text{modulus} \).

**REMARK:**
The quotient of the underlying skew polynomial ring by the principal ideal generated by \( \text{modulus} \) is in general *not* a ring.

As a consequence, Sage first computes exactly \( \text{self}^{\text{exp}} \) and then reduce it modulo \( \text{modulus} \).

**EXAMPLES:**

\[
sage: k.<t> = GF(5^3) \\
sage: Frob = k.frobenius_endomorphism() \\
sage: S.<x> = k['x',Frob] \\
sage: a = x + t \\
sage: b = a^10 \quad \# \text{short form for } \`\text{a\_pow\_}(10)`\' \\
sage: b == a*a*a*a*a*a*a*a*a*a \\
True \\
sage: modulus = x^3 + t*x^2 + (t+3)*x - 2 \\
sage: br = a.right\_power\_mod(10,modulus); br \\
(t^2 + t)*x^2 + (3*t^2 + 1)*x + t^2 + t \\
sage: rq, rr = b.right\_quo\_rem(modulus) \\
sage: br == rr \\
True \\
sage: a.right\_power\_mod(100,modulus) \\
(2*t^2 + 3)*x^2 + (t^2 + 4*t + 2)*x + t^2 + 2*t + 1
\]

\[\text{right\_quo\_rem}(\text{other})\]
Return the quotient and remainder of the right euclidean division of \( \text{self} \) by \( \text{other} \).

**INPUT:**
- \( \text{other} \) – a skew polynomial in the same ring as \( \text{self} \)

**OUTPUT:**
- the quotient and the remainder of the left euclidean division of this skew polynomial by \( \text{other} \)
Note: This will fail if the leading coefficient of the divisor is not a unit.

EXAMPLES:

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = S.random_element(degree=4); a
(-t - 95)*x^4 + x^3 + (2*t - 1)*x
sage: b = S.random_element(monic=True); b
x^2 + (-12*t - 2)*x
sage: q, r = a.right_quo_rem(b)
sage: a == q*b + r
True
```

The leading coefficient of the divisor need to be invertible:

```python
sage: c = S.random_element(); c
(t - 1)*x^2 + t^2*x
sage: a.right_quo_rem(c)
Traceback (most recent call last):
  ...
NotImplementedError: the leading coefficient of the divisor is not invertible
```

`truncate(n)`
Return the polynomial resulting from discarding all monomials of degree at least \( n \).

EXAMPLES:

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = t*x^3 + x^4 + (t+1)*x^2
sage: a.truncate(4)
t*x^3 + (t + 1)*x^2
sage: a.truncate(3)
(t + 1)*x^2
```

`valuation()`
Return the minimal degree of a non-zero monomial of `self`.

By convention, the zero skew polynomial has valuation \( +\infty \).

EXAMPLES:

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x^2 + t*x^3 + t^2*x
sage: a.valuation()
1
```

By convention, the valuation of 0 is \( +\infty \):

```python
sage: S(0).valuation()
+Infinity
```
4.2 Constructor for skew polynomial rings

This module provides the function `SkewPolynomialRing()`, which constructs rings of univariate skew 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).

**AUTHOR:**

- Xavier Caruso (2012-06-29): initial version
- Arpit Merchant (2016-08-04): improved docstrings, added doctests and refactored method
- Johan Rosenkilde (2016-08-03): changes to import format

```python
sage.rings.polynomial.skew_polynomial_ring_constructor.SkewPolynomialRing(base_ring, base_ring_automorphism=None, names=None, sparse=False)
```

Return the globally unique skew polynomial ring with the given properties and variable names.

Given a ring $R$ and a ring automorphism $\sigma$ of $R$, the ring of skew polynomials $R[X, \sigma]$ is the usual abelian group polynomial $R[X]$ equipped with the modification multiplication deduced from the rule $Xa = \sigma(a)X$.

**See also:**

- `sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_general`
- `sage.rings.polynomial.skew_polynomial_element.SkewPolynomial`

**INPUT:**

- `base_ring` – a commutative ring
- `base_ring_automorphism` – an automorphism of the base ring (also called twisting map)
- `names` – a string or a list of strings

**Note:** The current implementation of skew polynomial rings does not support derivations. Sparse skew polynomials and multivariate skew polynomials are also not implemented.

**OUTPUT:**

A univariate skew polynomial ring over `base_ring` twisted by `base_ring_automorphism` when `names` is a string with no commas (,) or a list of length 1. Otherwise we raise a `NotImplementedError` as multivariate skew polynomial rings are not yet implemented.

**UNIQUENESS and IMMUTABILITY:**

In Sage, there is exactly one skew polynomial ring for each triple (base ring, twisting map, name of the variable).

**EXAMPLES** of VARIABLE NAME CONTEXT:

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = SkewPolynomialRing(R, sigma); S
Skew Polynomial Ring in x over Univariate Polynomial Ring in t over Integer Ring
  twisted by t |--> t + 1
```
The names of the variables defined above cannot be arbitrarily modified because each skew polynomial ring is unique in Sage and other objects in Sage could have pointers to that skew polynomial ring.

However, the variable can be changed within the scope of a `with` block using the `localvars` context:

```python
sage: with localvars(S, ['y']):
    print(S)
Skew Polynomial Ring in y over Univariate Polynomial Ring in t over Integer Ring
twisted by t |--> t + 1
```

**SQUARE BRACKETS NOTATION:**

You can alternatively create a skew polynomial ring over $R$ twisted by `base_ring_automorphism` by writing `$R['varname', base_ring_automorphism]`.

**EXAMPLES:**

We first define the base ring:

```python
sage: R.<t> = ZZ[]; R
Univariate Polynomial Ring in t over Integer Ring
```

and the twisting map:

```python
sage: base_ring_automorphism = R.hom([t+1]); base_ring_automorphism
Ring endomorphism of Univariate Polynomial Ring in t over Integer Ring
    Defn: t |--> t + 1
```

Now, we are ready to define the skew polynomial ring:

```python
sage: S = SkewPolynomialRing(R, base_ring_automorphism, names='x'); S
Skew Polynomial Ring in x over Univariate Polynomial Ring in t over Integer Ring
twisted by t |--> t + 1
```

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

```python
sage: S.<x> = SkewPolynomialRing(R, base_ring_automorphism)
sage: (x + t)^2
x^2 + (2*t + 1)*x + t^2
```

Here is an example with the square bracket notations:

```python
sage: S.<x> = R['x', base_ring_automorphism]; S
Skew Polynomial Ring in x over Univariate Polynomial Ring in t over Integer Ring
twisted by t |--> t + 1
```

Rings with different variables names are different:

```python
sage: R['x', base_ring_automorphism] == R['y', base_ring_automorphism]
False
```

**Todo:**

- Sparse Skew Polynomial Ring
- Multivariate Skew Polynomial Ring
- Add derivations.
4.3 Skew Univariate Polynomial Rings

This module provides the `SkewPolynomialRing_general`, which constructs a general dense univariate skew polynomials over commutative base rings with automorphisms over the base rings. This is usually accessed only indirectly through the constructor `sage.rings.polynomial.skew_polynomial_constructor.SkewPolynomialRing()`.

See `SkewPolynomialRing_general` for a definition of a univariate skew polynomial ring.

AUTHOR:

- Xavier Caruso (2012-06-29): initial version
- Arpit Merchant (2016-08-04): improved docstrings, fixed doctests and refactored classes and methods
- Johan Rosenkilde (2016-08-03): changes for bug fixes, docstring and doctest errors

```python
class sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_general(base_ring, twist_map, name, sparse, element_class)
```

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

A general implementation of univariate skew polynomial ring over a commutative ring.

Let $R$ be a commutative ring, and let $\sigma$ be an automorphism of $R$. The ring of skew polynomials $R[X, \sigma]$ is the polynomial ring $R[X]$, where the addition is the usual polynomial addition, but the multiplication operation is defined by the modified rule

$$X \cdot a = \sigma(a)X.$$

This means that $R[X, \sigma]$ is a non-commutative ring. Skew polynomials were first introduced by Ore [Ore1933].

EXAMPLES:

```python
sage: R.<t> = ZZ
sage: sigma = R.hom([t+1])
S.<x> = SkewPolynomialRing(R,sigma); S
Skew Polynomial Ring in x over Univariate Polynomial Ring in t over Integer Ring
  twisted by t |--> t + 1
```

One can also use a shorter syntax:

```python
S.<x> = R['x',sigma]; S
Skew Polynomial Ring in x over Univariate Polynomial Ring in t over Integer Ring
  twisted by t |--> t + 1
```

If we omit the diamond notation, the variable holding the indeterminate is not assigned:

```python
Sy = R['y',sigma]
sage: y
Traceback (most recent call last):
```
Note however that contrary to usual polynomial rings, we cannot omit the variable name on the RHS, since this collides with the notation for creating polynomial rings:

```
sage: Sz.<z> = R[sigma]
Traceback (most recent call last):
  ...ValueError: variable name 'Ring endomorphism of Univariate Polynomial Ring in t
                        \rightarrow over Integer Ring
                        Defn: t |--> t + 1' is not alphanumeric
```

Of course, skew polynomial rings with different twist maps are not equal either:

```
sage: R['x',sigma] == R['x',sigma^2]
False
```

Saving and loading of polynomial rings works:

```
sage: loads(dumps(R['x',sigma])) == R['x',sigma]
True
```

There is a coercion map from the base ring of the skew polynomial rings:

```
sage: S.has_coerce_map_from(R)
True
sage: x.parent()
Skew Polynomial Ring in x over Univariate Polynomial Ring in t over Integer Ring
        twisted by t |--> t + 1
sage: t.parent()
Univariate Polynomial Ring in t over Integer Ring
sage: y = x+t; y
x + t
sage: y.parent() is S
True
```

See also:

`sage.rings.polynomial.skew_polynomial_ring_constructor.
SkewPolynomialRing()` `sage.rings.polynomial.skew_polynomial_element`

**change_var**(var)

Return the skew polynomial ring in variable var with the same base ring and twist map as self.

**INPUT:**

• var – a string representing the name of the new variable.

**OUTPUT:**

self with variable name changed to var.

**EXAMPLES:**
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: R.<x> = SkewPolynomialRing(k,Frob); R
Skew Polynomial Ring in x over Finite Field in t of size 5^3 twisted by t |--> t^5
sage: Ry = R.change_var('y'); Ry
Skew Polynomial Ring in y over Finite Field in t of size 5^3 twisted by t |--> t^5
sage: Ry is R.change_var('y')
True

characteristic()
Return the characteristic of the base ring of self.

EXAMPLES:

sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: R['x',sigma].characteristic()
0
sage: k.<u> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: k['y',Frob].characteristic()
5

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

INPUT:

• n – index of generator to return (default: 0). Exists for compatibility with other polynomial rings.

EXAMPLES:

sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]; S
Skew Polynomial Ring in x over Univariate Polynomial Ring in t over Rational Field twisted by t |--> t + 1
sage: y = S.gen(); y
x
sage: y == x
True
sage: y is x
True
sage: S.gen(0)

This is also known as the parameter:

sage: S.parameter() is S.gen()
True

gens_dict()
Return a {name: variable} dictionary of the generators of self.

EXAMPLES:
is_commutative()
Return True if this skew polynomial ring is commutative, i.e. if the twist map is the identity.

EXAMPLES:

```sage
def k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: S.is_commutative()
False
sage: T.<y> = k['y',Frob^3]
sage: T.is_commutative()
True
```

is_exact()
Return True if elements of this skew polynomial ring are exact. This happens if and only if elements of
the base ring are exact.

EXAMPLES:

```sage
def k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: S.is_exact()
True
def k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: T.<y> = k['y',Frob^3]
sage: T.is_exact()
False
```

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

EXAMPLES:

```sage
def k.<t> = GF(5^3)
sage: k.is_finite()
True
def k.<t> = GF(5^3)
sage: k.is_finite()
True
def k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: S.is_finite()
False
```

is_sparse()
Return True if the elements of this polynomial ring are sparsely represented.
Warning: Since sparse skew polynomials are not yet implemented, this function always returns False.

EXAMPLES:

```
sage: R.<t> = RR[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: S.is_sparse()
False
```

lagrange_polynomial (points)
Return the minimal-degree polynomial which interpolates the given points.

More precisely, given \( n \) pairs \((x_1, y_1), \ldots, (x_n, y_n) \in R^2\), where \( R \) is \( \text{self.base_ring()} \), compute a skew polynomial \( p(x) \) such that \( p(x_i) = y_i \) for each \( i \), under the condition that the \( x_i \) are linearly independent over the fixed field of \( \text{self.twist_map()} \).

If the \( x_i \) are linearly independent over the fixed field of \( \text{self.twist_map()} \) then such a polynomial is guaranteed to exist. Otherwise, it might exist depending on the \( y_i \), but the algorithm used in this implementation does not support that, and so an error is always raised.

INPUT:

- points—a list of pairs \((x_1, y_1), \ldots, (x_n, y_n)\) of elements of the base ring of \( \text{self} \).
  The \( x_i \) should be linearly independent over the fixed field of \( \text{self.twist_map()} \).

OUTPUT:
The Lagrange polynomial.

EXAMPLES:

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: points = [(t, 3*t^2 + 4*t + 4), (t^2, 4*t)]
sage: d = S.lagrange_polynomial(points); d
x + t
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: T.<x> = R['x', sigma]
sage: points = [ (1, t^2 + 3*t + 4), (t, 2*t^2 + 3*t + 1), (t^2, t^2 + 3*t + 4) ]
sage: p = T.lagrange_polynomial(points); p
(-t^4 - 2*t - 3)/-2)*x^2 + (-t^4 - t^3 - t^2 - 3*t - 2)*x + (-t^4 - 2*t^3 -
-4*t^2 - 10*t - 9)/-2
sage: p.multi_point_evaluation([1, t, t^2]) == [ t^2 + 3*t + 4, 2*t^2 + 3*t +
-4*t + 4 ]
True
```

If the \( x_i \) are linearly dependent over the fixed field of \( \text{self.twist_map()} \), then an error is raised:

```
sage: T.lagrange_polynomial([ (t, 1), (2*t, 3) ])  
Traceback (most recent call last):
...  
ValueError: the given evaluation points are linearly dependent over the fixed 
field of the twist map, so a Lagrange polynomial could not be determined. 
(continues on next page)```
minimal_vanishing_polynomial(eval_pts)

Return the minimal-degree, monic skew polynomial which vanishes at all the given evaluation points.

The degree of the vanishing polynomial is at most the length of eval_pts. Equality holds if and only if the elements of eval_pts are linearly independent over the fixed field of self.twist_map().

INPUT:
- eval_pts – list of evaluation points which are linearly independent over the fixed field of the twist map of the associated skew polynomial ring

OUTPUT:
The minimal vanishing polynomial.

EXAMPLES:

```sage
def main()
    k.<t> = GF(5^3)
    Frob = k.frobenius_endomorphism()
    S.<x> = k['x', Frob]
    eval_pts = [1, t, t^2]
    b = S.minimal_vanishing_polynomial(eval_pts)
    print(b)
    eval = b.multi_point_evaluation(eval_pts)
    print(eval)
```

```sage
sage: main()
x^3 + 4
[0, 0, 0]
```

The minimal vanishing polynomial evaluates to 0 at each of the evaluation points:

```sage
sage: main()
x^3 + 4
[0, 0, 0]
```

If the evaluation points are linearly dependent over the fixed field of the twist map, then the returned polynomial has lower degree than the number of evaluation points:

```sage
sage: main()
x + 3*t^2 + 3*t
```

ngens()

Return the number of generators of this skew polynomial ring, which is 1.

EXAMPLES:

```sage
def main()
    R.<t> = RR[]
    sigma = R.hom([t+1])
    S.<x> = R['x', sigma]
    print(S.ngens())
```

```sage
sage: main()
1
```

parameter(n=0)

Return the indeterminate generator of this skew polynomial ring.

INPUT:
- n – index of generator to return (default: 0). Exists for compatibility with other polynomial rings.

EXAMPLES:
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]; S
Skew Polynomial Ring in x over Univariate Polynomial Ring in t over Rational Field twisted by t |--> t + 1
sage: y = S.gen(); y
x
sage: y == x
True
sage: y is x
True
sage: S.gen(0)

This is also known as the parameter:

sage: S.parameter() is S.gen()
True

random_element\(\text{\textit{(degree=2, monic=False, }\ast\text{args, }\ast\ast\text{kwds)}}\)
Return a random skew polynomial in self.

INPUT:

- degree – (default: 2) integer with degree or a tuple of integers with minimum and maximum degrees
- monic – (default: False) if True, return a monic skew polynomial
- *args, **kwds – passed on to the random_element method for the base ring

OUTPUT:

Skew polynomial such that the coefficients of \(x^i\), for \(i\) up to degree, are random elements from the base ring, randomized subject to the arguments *args and **kwds.

EXAMPLES:

sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x', Frob]
sage: S.random_element() # random
\((2*t^2 + 3)*x^2 + (4*t^2 + t + 4)*x + 2*t^2 + 2\)
sage: S.random_element(monic=True) # random
\(x^2 + (2*t^2 + t + 1)*x + 3*t^2 + 3*t + 2\)

Use degree to obtain polynomials of higher degree

sage: p = S.random_element(degree=5) # random \((t^2 + 3*t)*x^4 + (4*t + 4)*x^3 + (4*t^2 + 4*t)*x^2 + (2*t^2 + 1)*x + 3\)

When monic is False, the returned skew polynomial may have a degree less than degree (it happens when the random leading coefficient is zero). However, if monic is True, this can’t happen:

sage: p = S.random_element(degree=4, monic=True)
True
sage: p.degree() == 4
True

If a tuple of two integers is given for the degree argument, a random integer will be chosen between the first and second element of the tuple as the degree, both inclusive:
sage: S.random_element(degree=(2,7))  # random
(3*t^2 + 1)*x^4 + (4*t + 2)*x^3 + (4*t + 1)*x^2
+ (t^2 + 3*t + 3)*x + 3*t^2 + 2*t + 2

If the first tuple element is greater than the second, a ValueError is raised:

sage: S.random_element(degree=(5,4))
Traceback (most recent call last):
  ...
ValueError: first degree argument must be less or equal to the second

\textbf{twist_map} (\(n=1\))

Return the twist map, the automorphism of the base ring of self, iterated \(n\) times.

INPUT:

- \(n\) - an integer (default: 1)

OUTPUT:

\(n\)-th iterative of the twist map of this skew polynomial ring.

EXAMPLES:

sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: S.twist_map()  
Ring endomorphism of Univariate Polynomial Ring in t over Rational Field
  Defn: t |--> t + 1
sage: S.twist_map() == sigma
True
sage: S.twist_map(10)
Ring endomorphism of Univariate Polynomial Ring in t over Rational Field
  Defn: t |--> t + 10

If \(n\) in negative, Sage tries to compute the inverse of the twist map:

sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: T.<y> = k['y',Frob]
sage: T.twist_map(-1)
Frobenius endomorphism t |---> t^(5^2) on Finite Field in t of size 5^3

Sometimes it fails, even if the twist map is actually invertible:

sage: S.twist_map(-1)
Traceback (most recent call last):
  ...
NotImplementedError: inversion of the twist map Ring endomorphism of
  Univariate Polynomial Ring in t over Rational Field
  Defn: t |--> t + 1
5.1 Fraction Field of Integral Domains

AUTHORS:
- William Stein (with input from David Joyner, David Kohel, and Joe Wetherell)
- Burcin Erocal
- Julian Rüth (2017-06-27): embedding into the field of fractions and its section

EXAMPLES:
Quotienting is a constructor for an element of the fraction field:

```sage
R.<x> = QQ[]
sage: (x^2-1)/(x+1)
x - 1
sage: parent((x^2-1)/(x+1))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

The GCD is not taken (since it doesn’t converge sometimes) in the inexact case:

```sage
Z.<z> = CC[]
sage: I = CC.gen()
sage: (1+I+z)/(z+0.1*I)
(z + 1.00000000000000 + I)/(z + 0.100000000000000*I)
sage: (1+I*z)/(z+1.1)
(I*z + 1.00000000000000)/(z + 1.10000000000000)
```

Create the fraction field of the integral domain `R`.

```python
sage.rings.fraction_field.FractionField(R, names=None)
```

INPUT:
- `R` – an integral domain
- `names` – ignored

EXAMPLES:
We create some example fraction fields:

```sage
FractionField(IntegerRing())
Rational Field
sage: FractionField(PolynomialRing(RationalField(),'x'))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

(continues on next page)
Dividing elements often implicitly creates elements of the fraction field:

```python
sage: x = PolynomialRing(RationalField(), 'x').gen()
sage: f = x/(x+1)
sage: g = x**3/(x+1)
sage: f/g
1/x^2
sage: g/f
x^2
```

The input must be an integral domain:

```python
sage: Frac(Integers(4))
Traceback (most recent call last):
...
TypeError: R must be an integral domain.
```

```python
class sage.rings.fraction_field.FractionFieldEmbedding
Bases: sage.structure.coerce_maps.DefaultConvertMap_unique

The embedding of an integral domain into its field of fractions.

EXAMPLES:

```python
sage: R.<x> = QQ[]
sage: f = R.fraction_field().coerce_map_from(R); f
Coercion map:
  From: Univariate Polynomial Ring in x over Rational Field
  To:   Fraction Field of Univariate Polynomial Ring in x over Rational Field

is_injective()  
    Return whether this map is injective.

EXAMPLES:

    The map from an integral domain to its fraction field is always injective:

```python
sage: R.<x> = QQ[]
sage: R.fraction_field().coerce_map_from(R).is_injective()
True
```

```python
is_surjective()  
    Return whether this map is surjective.

EXAMPLES:

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

```python
section()  
    Return a section of this map.

EXAMPLES:
```

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

```python
sage: R.fraction_field().coerce_map_from(R).section()
```

```python
Section map:
  From: Fraction Field of Univariate Polynomial Ring in x over Rational Field
  To:  Univariate Polynomial Ring in x over Rational Field
```

```python
class sage.rings.fraction_field.FractionFieldEmbeddingSection
    Bases: sage.categories.map.Section
```

The section of the embedding of an integral domain into its field of fractions.

EXAMPLES:

```python
sage: R.<x> = QQ[]
sage: f = R.fraction_field().coerce_map_from(R).section(); f
```

```python
Section map:
  From: Fraction Field of Univariate Polynomial Ring in x over Rational Field
  To:  Univariate Polynomial Ring in x over Rational Field
```

```python
class sage.rings.fraction_field.FractionField_1poly_field(R, element_class=<class 'sage.rings.fraction_field_element.FractionFieldElement_1poly_field'>)
    Bases: sage.rings.fraction_field.FractionField_generic
```

The fraction field of a univariate polynomial ring over a field.

Many of the functions here are included for coherence with number fields.

```python
class_number()
    Here for compatibility with number fields and function fields.
```

EXAMPLES:

```python
sage: R.<t> = GF(5)[]; K = R.fraction_field()
sage: K.class_number()
1
```

```python
function_field()
    Return the isomorphic function field.
```

EXAMPLES:

```python
sage: R.<t> = GF(5)[]
sage: K = R.fraction_field()
sage: K.function_field()
Rational function field in t over Finite Field of size 5
```

See also:

sage.rings.function_field.RationalFunctionField.field()

```python
maximal_order()
    Return the maximal order in this fraction field.
```

EXAMPLES:

```python
sage: K = FractionField(GF(5)['t'])
sage: K.maximal_order()
Univariate Polynomial Ring in t over Finite Field of size 5
```

5.1. Fraction Field of Integral Domains  507
ring_of_integers()
Return the ring of integers in this fraction field.

EXAMPLES:

```python
sage: K = FractionField(GF(5)['t'])
sage: K.ring_of_integers()
Univariate Polynomial Ring in t over Finite Field of size 5
```

class sage.rings.fraction_field.FractionField_generic(R, element_class=<class 'sage.rings.fraction_field_element.FractionFieldElement'>, category=Category of quotient fields)

Bases: sage.rings.ring.Field
The fraction field of an integral domain.

base_ring()
Return the base ring of self.

This is the base ring of the ring which this fraction field is the fraction field of.

EXAMPLES:

```python
sage: R = Frac(ZZ['t'])
sage: R.base_ring()
Integer Ring
```

characteristic()
Return the characteristic of this fraction field.

EXAMPLES:

```python
sage: R = Frac(ZZ['t'])
sage: R.base_ring()
Integer Ring
sage: R = Frac(ZZ['t']); R.characteristic()
0
sage: R = Frac(GF(5)['w']); R.characteristic()
5
```

construction()
EXAMPLES:

```python
sage: Frac(ZZ['x']).construction()
(FractionField, Univariate Polynomial Ring in x over Integer Ring)
sage: K = Frac(GF(3)['t'])
sage: f, R = K.construction()
sage: f(R)
Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 3
sage: f(R) == K
True
```

gen(i=0)
Return the i-th generator of self.

EXAMPLES:

```python
sage: R = Frac(PolynomialRing(QQ,'z',10)); R
Fraction Field of Multivariate Polynomial Ring in z0, z1, z2, z3, z4, z5, z6, z7, z8, z9 over Rational Field
(continues on next page)```
sage: R.0
z0
sage: R.gen(3)
z3
sage: R.3
z3

is_exact()

Return if self is exact which is if the underlying ring is exact.

EXAMPLES:

sage: Frac(ZZ['x']).is_exact()
True
sage: Frac(CDF['x']).is_exact()
False

is_field(proof=True)

Return True, since the fraction field is a field.

EXAMPLES:

sage: Frac(ZZ).is_field()
True

is_finite()

Tells whether this fraction field is finite.

Note: A fraction field is finite if and only if the associated integral domain is finite.

EXAMPLES:

sage: Frac(QQ['a','b','c']).is_finite()
False

ngens()

This is the same as for the parent object.

EXAMPLES:

sage: R = Frac(PolynomialRing(QQ,'z',10)); R
Fraction Field of Multivariate Polynomial Ring in z0, z1, z2, z3, z4, z5, z6, ...
˓→z7, z8, z9 over Rational Field
sage: R.ngens()
10

random_element(*args, **kwds)

Return a random element in this fraction field.

The arguments are passed to the random generator of the underlying ring.

EXAMPLES:

sage: F = ZZ['x'].fraction_field()
sage: F.random_element()  # random
(2*x - 8)/(-x^2 + x)
sage: f = F.random_element(degree=5)
sage: f.numerator().degree()
5
sage: f.denominator().degree()
5

ring()
Return the ring that this is the fraction field of.

EXAMPLES:

sage: R = Frac(QQ['x,y'])
sage: R
Fraction Field of Multivariate Polynomial Ring in x, y over Rational Field
sage: R.ring()
Multivariate Polynomial Ring in x, y over Rational Field

some_elements()
Return some elements in this field.

EXAMPLES:

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

sage.rings.fraction_field.is_FractionField(x)
Test whether or not x inherits from FractionField_generic.

EXAMPLES:

sage: from sage.rings.fraction_field import is_FractionField
sage: is_FractionField(Frac(ZZ['x']))
True
sage: is_FractionField(QQ)
False

5.2 Fraction Field Elements

AUTHORS:

• William Stein (input from David Joyner, David Kohel, and Joe Wetherell)

• Sebastian Pancratz (2010-01-06): Rewrite of addition, multiplication and derivative to use Henrici’s algorithms [Hor1972]
class sage.rings.fraction_field_element.FractionFieldElement

Bases: sage.structure.element.FieldElement

EXAMPLES:

```
sage: K = FractionField(PolynomialRing(QQ, 'x'))
sage: K
Fraction Field of Univariate Polynomial Ring in x over Rational Field
dsage: loads(K.dumps()) == K
True
dsage: x = K.gen()
dsage: f = (x^3 + x)/(17 - x^19); f
(-x^3 - x)/(x^19 - 17)
dsage: loads(f.dumps()) == f
True
```

denominator()

Return the denominator of self.

EXAMPLES:

```
sage: R.<x,y> = ZZ[]
sage: f = x/y+1; f
(x + y)/y
sage: f.denominator()
y
```

is_one()

Return True if this element is equal to one.

EXAMPLES:

```
sage: F = ZZ['x,y'].fraction_field()
sage: x,y = F.gens()
sage: (x/x).is_one()
True
sage: (x/y).is_one()
False
```

is_square(root=False)

Return whether or not self is a perfect square.

If the optional argument root is True, then also returns a square root (or None, if the fraction field element is not square).

INPUT:

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

OUTPUT:

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

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: (1/t).is_square()
False
sage: (1/t^6).is_square()
```

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

True
sage: ((1+t)^4/t^6).is_square()
True
sage: (4*(1+t)^4/t^6).is_square()
True
sage: (2*(1+t)^4/t^6).is_square()
False
sage: ((1+t)/t^6).is_square()
False

sage: (4*(1+t)^4/t^6).is_square(root=True)
(True, (2*t^2 + 4*t + 2)/t^3)
sage: (2*(1+t)^4/t^6).is_square(root=True)
(False, None)

sage: R.<x> = QQ[]
sage: a = 2*(x+1)^2 / (2*(x-1)^2); a
(x^2 + 2*x + 1)/(x^2 - 2*x + 1)
sage: a.is_square()
True
sage: (0/x).is_square()
True

is_zero()
Return True if this element is equal to zero.

EXAMPLES:

sage: F = ZZ['x,y'].fraction_field()
sage: x,y = F.gens()
sage: t = F(0)/x
sage: t.is_zero()
True
sage: u = 1/x - 1/x
sage: u.is_zero()
True
sage: u.parent() is F
True

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

EXAMPLES:

sage: R = QQ['t'].fraction_field()
sage: t = R.gen()
sage: p = (t+1)^3 / (t^2+t-1)^3
sage: p.nth_root(3)
(t + 1)/(t^2 + t - 1)
sage: p = (t+1) / (t-1)
sage: p.nth_root(2)
Traceback (most recent call last):
  ... ValueError: not a 2nd power

numerator()
Return the numerator of self.
EXAMPLES:

```
sage: R.<x,y> = ZZ[
  sage: f = x/y+1; f
  (x + y)/y
  sage: f.numerator()
  x + y
```

**reduce()**
Reduce this fraction.
Divides out the gcd of the numerator and denominator. If the denominator becomes a unit, it becomes 1.
Additionally, depending on the base ring, the leading coefficients of the numerator and the denominator may be normalized to 1.
Automatically called for exact rings, but because it may be numerically unstable for inexact rings it must be called manually in that case.

EXAMPLES:

```
sage: R.<x> = RealField(10)[
  sage: f = (x^2+2*x+1)/(x+1); f
  (x^2 + 2.0*x + 1.0)/(x + 1.0)
  sage: f.reduce(); f
  x + 1.0
```

**specialization**(D=None, phi=None)
Returns the specialization of a fraction element of a polynomial ring

**valuation**(v=None)
Return the valuation of self, assuming that the numerator and denominator have valuation functions defined on them.

EXAMPLES:

```
sage: x = PolynomialRing(RationalField(), 'x').gen()
  sage: f = (x^3 + x)/(x^2 - 2*x^3)
  sage: f
  (-1/2*x^2 - 1/2)/(x^2 - 1/2*x)
  sage: f.valuation()
  -1
  sage: f.valuation(x^2+1)
  1
```

**class** sage.rings.fraction_field_element.FractionFieldElement_1poly_field

A fraction field element where the parent is the fraction field of a univariate polynomial ring over a field.
Many of the functions here are included for coherence with number fields.

**is_integral()**
Returns whether this element is actually a polynomial.

EXAMPLES:

```
sage: R.<t> = QQ[
  sage: elt = (t^2 + t - 2) / (t + 2); elt
  # == (t + 2)*(t - 1)/(t + 2)
  t - 1
  sage: elt.is_integral()
```
reduce() 
Pick a normalized representation of self.

In particular, for any a == b, after normalization they will have the same numerator and denominator.

EXAMPLES:

For univariate rational functions over a field, we have:

```sage
code
sage: R.<x> = QQ[]
sage: (2 + 2*x) / (4*x)  # indirect doctest
(1/2*x + 1/2)/x
```

Compare with:

```sage
code
sage: R.<x> = ZZ[]
sage: (2 + 2*x) / (4*x)
(x + 1)/(2*x)
```

support() 
Returns a sorted list of primes dividing either the numerator or denominator of this element.

EXAMPLES:

```sage
code
sage: R.<t> = QQ[]
sage: h = (t^14 + 2*t^12 - 4*t^11 - 8*t^9 + 6*t^8 + 12*t^6 - 4*t^5 - 8*t^3 +
→t^2 + 2)/(t^6 + 6*t^5 + 9*t^4 - 2*t^2 - 12*t - 18)
sage: h.support()
[t - 1, t + 3, t^2 + 2, t^2 + t + 1, t^4 - 2]
```

sage.rings.fraction_field_element.is_FractionFieldElement(x) 
Return whether or not x is a FractionFieldElement.

EXAMPLES:

```sage
code
sage: from sage.rings.fraction_field_element import is_FractionFieldElement
sage: R.<x> = ZZ[]
sage: is_FractionFieldElement(x/2)
False
sage: is_FractionFieldElement(2/x)
True
sage: is_FractionFieldElement(1/3)
False
```

sage.rings.fraction_field_element.make_element(parent, numerator, denominator) 
Used for unpickling FractionFieldElement objects (and subclasses).

EXAMPLES:

```sage
code
sage: from sage.rings.fraction_field_element import make_element
sage: R = ZZ['x,y']
sage: x, y = R.gens()
```
sage: F = R.fraction_field()
sage: make_element(F, 1+x, 1+y)
(x + 1)/(y + 1)

sage.rings.fraction_field_element.make_element_old(parent, cdict)
Used for unpicking old FractionFieldElement pickles.

EXAMPLES:

sage: from sage.rings.fraction_field_element import make_element_old
sage: R.<x,y> = ZZ[

5.3 Univariate rational functions over prime fields

class sage.rings.fraction_field_FpT.FpT(R, names=None)
Bases: sage.rings.fraction_field.FractionField_1poly_field

This class represents the fraction field GF(p)(T) for
2 < p < \sqrt{2^31 - 1}.

EXAMPLES:

sage: R.<T> = GF(71)[

iter (bound=None, start=None)
EXAMPLES:

sage: from sage.rings.fraction_field_FpT import *

5.3. Univariate rational functions over prime fields 515
denominator()  
Returns the denominator of this element, as an element of the polynomial ring.

EXAMPLES:

```python
sage: K = GF(11)['t'].fraction_field()
sage: t = K.gen(0); a = (t + 1/t)^3 - 1
sage: a.denom()
t^3
```

factor()  
EXAMPLES:

```python
sage: K = Frac(GF(5)['t'])
sage: t = K.gen()
sage: f = 2 * (t+1) * (t^2+t+1)^2 / (t-1)
sage: factor(f)
(2) * (t + 4)^-1 * (t + 1) * (t^2 + t + 1)^2
```

is_square()  
Returns True if this element is the square of another element of the fraction field.

EXAMPLES:

```python
sage: K = GF(13)['t'].fraction_field(); t = K.gen()
sage: t.is_square()  # False
sage: (1/t^2).is_square()  # True
sage: K(0).is_square()  # True
```

next()  
This function iterates through all polynomials, returning the “next” polynomial after this one.

The strategy is as follows:

- We always leave the denominator monic.
- We progress through the elements with both numerator and denominator monic, and with the denominator less than the numerator. For each such, we output all the scalar multiples of it, then all of the scalar multiples of its inverse.
- So if the leading coefficient of the numerator is less than p-1, we scale the numerator to increase it by 1.
- Otherwise, we consider the multiple with numerator and denominator monic.
  - If the numerator is less than the denominator (lexicographically), we return the inverse of that element.
  - If the numerator is greater than the denominator, we invert, and then increase the numerator (remaining monic) until we either get something relatively prime to the new denominator, or we reach the new denominator. In this case, we increase the denominator and set the numerator to 1.

EXAMPLES:
Returns the numerator of this element, as an element of the polynomial ring.

EXAMPLES:

```
sage: K = GF(11)['t'].fraction_field()
sage: t = K.gen(0); a = (t + 1/t)^3 - 1
sage: a.numerator()
t^6 + 3*t^4 + 10*t^3 + 3*t^2 + 1
```

Returns the numerator of this element, as an element of the polynomial ring.

EXAMPLES:

```
sage: K = GF(11)['t'].fraction_field()
sage: t = K.gen(0); a = (t + 1/t)^3 - 1
sage: a.numerator()
t^6 + 3*t^4 + 10*t^3 + 3*t^2 + 1
```

\texttt{sqrt (extend=True, all=False)}

Returns the square root of this element.

5.3. Univariate rational functions over prime fields
INPUT:

- `extend` - bool (default: True); if True, return a square root in an extension ring, if necessary. Otherwise, raise a `ValueError` if the square is not in the base ring.
- `all` - bool (default: False); if True, return all square roots of self, instead of just one.

EXAMPLES:

```python
sage: from sage.rings.fraction_field_FpT import *
sage: K = GF(7)['t'].fraction_field(); t = K.gen(0)
sage: p = (t + 2)^2/(3*t^3 + 1)^4
sage: p.sqrt()
(3*t + 6)/(t^6 + 3*t^3 + 4)
sage: p.sqrt()^2 == p
True
```

```python
subs (*args, **kwds)
```

EXAMPLES:

```python
sage: K = Frac(GF(11)['t'])
sage: t = K.gen()
sage: f = (t+1)/(t-1)
sage: f.subs(t=2)
3
sage: f.subs(X=2)
(t + 1)/(t + 10)
```

```python
valuation(v)
```

Returns the valuation of self at \( v \).

EXAMPLES:

```python
sage: R.<t> = GF(5)[]
sage: f = (t+1)^2 * (t^2+t+1) / (t-1)^3
sage: f.valuation(t+1)
2
sage: f.valuation(t-1)
-3
sage: f.valuation(t)
0
```

```python
class sage.rings.fraction_field_FpT.FpT_Fp_section
Bases: sage.categories.map.Section
```

This class represents the section from GF(p)(t) back to GF(p)[t]

EXAMPLES:

```python
sage: R.<t> = GF(5)[]
sage: K = R.fraction_field()
sage: f = GF(5).convert_map_from(K); f
Section map:
  From: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
  To:   Finite Field of size 5
sage: type(f)
<type 'sage.rings.fraction_field_FpT.FpT_Fp_section'>
```
Warning: Comparison of FpT_Fp_section objects is not currently implemented. See :trac: 23469.

```python
sage: fprime = loads(dumps(f))
sage: fprime == f
False
sage: fprime(3) == f(3)
True
```

class sage.rings.fraction_field_FpT.FpT_Polyring_section

Bases: sage.categories.map.Section

This class represents the section from GF(p)(t) back to GF(p)[t]

EXAMPLES:

```python
sage: R.<t> = GF(5)[]
sage: K = R.fraction_field()
sage: f = R.convert_map_from(K); f
Section map:
    From: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
    To:   Univariate Polynomial Ring in t over Finite Field of size 5
sage: type(f)
<type 'sage.rings.fraction_field_FpT.FpT_Polyring_section'>
```

Warning: Comparison of FpT_Polyring_section objects is not currently implemented. See :trac: 23469.

```python
sage: fprime = loads(dumps(f))
sage: fprime == f
False
sage: fprime(1+t) == f(1+t)
True
```

class sage.rings.fraction_field_FpT.FpT_iter

Bases: object

Returns a class that iterates over all elements of an FpT.

EXAMPLES:

```python
sage: K = GF(3)['t'].fraction_field()
sage: I = K.iter(1)
sage: list(I)
[0,
 1,
 2,
 t,
 t + 1,
 t + 2,
 2*t,
 2*t + 1,
 2*t + 2,
(continues on next page)
```

5.3. Univariate rational functions over prime fields
1/t,
2/t,
(t + 1)/t,
(t + 2)/t,
(2*t + 1)/t,
(2*t + 2)/t,
1/(t + 1),
2/(t + 1),
t/(t + 1),
(t + 2)/(t + 1),
2*t/(t + 1),
(2*t + 1)/(t + 1),
1/(t + 2),
2/(t + 2),
t/(t + 2),
(t + 1)/(t + 2),
2*t/(t + 2),
(2*t + 2)/(t + 2))

```
class sage.rings.fraction_field_FpT.Fp_FpT_coerce
Bases: sage.rings.morphism.RingHomomorphism

This class represents the coercion map from GF(p) to GF(p)(t)

EXAMPLES:
```
```sage
R.<t> = GF(5)[]
K = R.fraction_field()
f = K.coerce_map_from(GF(5)); f

Ring morphism:
  From: Finite Field of size 5
  To:  Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5

sage: type(f)
<type 'sage.rings.fraction_field_FpT.Fp_FpT_coerce'>
```

```
section()

Returns the section of this inclusion: the partially defined map from GF(p)(t) back to GF(p), defined on constant elements.

EXAMPLES:
```
```sage
R.<t> = GF(5)[]
K = R.fraction_field()
f = K.coerce_map_from(GF(5))
g = f.section(); g

Section map:
  From: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
  To:  Finite Field of size 5

sage: t = K.gen()
sage: g(f(1,3,reduce=False))
2
```
```
```
class sage.rings.fraction_field_FpT.Polyring_FpT_coerce
    Bases: sage.rings.morphism.RingHomomorphism

    This class represents the coercion map from GF(p)[t] to GF(p)(t)

    EXAMPLES:

    sage: R.<t> = GF(5)[]
    sage: K = R.fraction_field()
    sage: f = K.coerce_map_from(R); f
    Ring morphism:
    From: Univariate Polynomial Ring in t over Finite Field of size 5
    To:   Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
    sage: type(f)
    <type 'sage.rings.fraction_field_FpT.Polyring_FpT_coerce'>

    section()
    Returns the section of this inclusion: the partially defined map from GF(p)(t) back to GF(p)[t], defined on elements with unit denominator.

    EXAMPLES:

    sage: R.<t> = GF(5)[]
    sage: K = R.fraction_field()
    sage: f = K.coerce_map_from(R)
    sage: g = f.section(); g
    Section map:
    From: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
    To:   Univariate Polynomial Ring in t over Finite Field of size 5
    sage: t = K.gen()
    sage: g(t)
    t
    sage: g(1/t)
    Traceback (most recent call last):
    ...
    ValueError: not integral

class sage.rings.fraction_field_FpT.ZZ_FpT_coerce
    Bases: sage.rings.morphism.RingHomomorphism

    This class represents the coercion map from ZZ to GF(p)(t)

    EXAMPLES:

    sage: R.<t> = GF(17)[]
    sage: K = R.fraction_field()
    sage: f = K.coerce_map_from(ZZ); f
    Ring morphism:
    From: Integer Ring
    To:   Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 17

(continues on next page)
section()

Returns the section of this inclusion: the partially defined map from \( \text{GF}(p)(t) \) back to \( \mathbb{Z} \), defined on constant elements.

EXAMPLES:

```
sage: R.<t> = GF(5)[]
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(ZZ)
sage: g = f.section(); g
Composite map:
  From: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
  To: Integer Ring
  Defn: Section map:
          From: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
          To: Finite Field of size 5
          then
          Lifting map:
            From: Finite Field of size 5
            To: Integer Ring
sage: t = K.gen()
sage: g(f(1,3,reduce=False))
2
sage: g(t)
Traceback (most recent call last):
  ... ValueError: not constant
sage: g(1/t)
Traceback (most recent call last):
  ... ValueError: not integral
```

```
sage.rings.fraction_field_FpT.unpickle_FpT_element(K, numer, denom)
Used for pickling.
```
6.1 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

$$R[x_1, y_1, x_2, y_2, \ldots, x_n, y_n]/(x_1y_1 - 1, x_2y_2 - 1, \ldots, x_ny_n - 1).$$

AUTHORS:
- David Roe (2008-2-23): created
- David Loeffler (2009-07-10): cleaned up docstrings

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
### Examples

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

   ```python
   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:

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

   You must specify a name:

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

   ```python
   sage: R.<abc> = LaurentPolynomialRing(QQ, sparse=True); R
   Univariate Laurent Polynomial Ring in abc over Rational Field
   ```

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

   Rings with different variables are different:

   ```python
   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 Field

sage: LaurentPolynomialRing(GF(7), 'y', 5)
 Multivariate Laurent Polynomial Ring in y0, y1, y2, y3, y4 over Finite Field of size 7

sage: LaurentPolynomialRing(QQ, 'y', 3, sparse=True)
 Multivariate Laurent Polynomial Ring in y0, y1, y2 over Rational Field

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

sage: R = LaurentPolynomialRing(GF(7),15,'w'); R
 Multivariate Laurent Polynomial Ring in w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13, w14 over Finite Field of size 7
sage: R.inject_variables()
 Defining w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13, w14
sage: (w0 + 2*w8 + w13)^2
 w0^2 + 4*w0*w8 + 4*w8^2 + 2*w0*w13 + 4*w8*w13 + w13^2

class sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic(R)
 Bases: sage.rings.ring.CommutativeRing, sage.structure.parent.Parent

Laurent polynomial ring (base class).

EXAMPLES:
This base class inherits from `CommutativeRing`. Since trac ticket #11900, it is also initialised as such:

```python
sage: R.<x1,x2> = LaurentPolynomialRing(QQ)
sage: R.category()
Category of commutative rings
sage: TestSuite(R).run()
```

### change_ring

**Signature:**

```
change_ring(base_ring=None, names=None, sparse=False, order=None)
```

**Examples:**

```python
sage: R = LaurentPolynomialRing(QQ,2,'x')
sage: R.change_ring(ZZ)
Multivariate Laurent Polynomial Ring in x0, x1 over Integer Ring
```

### characteristic

**Description:**

Returns the characteristic of the base ring.

**Examples:**

```python
sage: LaurentPolynomialRing(QQ,2,'x').characteristic()
0
sage: LaurentPolynomialRing(GF(3),2,'x').characteristic()
3
```

### completion

**Signature:**

```
completion(p, prec=None, extras=None)
```

**Examples:**

```python
sage: P.<x>=LaurentPolynomialRing(QQ)
sage: P
Univariate Laurent Polynomial Ring in x over Rational Field
sage: PP=P.completion(x)
sage: PP
Laurent Series Ring in x over Rational Field
sage: f=1-1/x
sage: PP(f)
-x^-1 + 1
sage: 1/PP(f)
-x - x^2 - x^3 - x^4 - x^5 - x^6 - x^7 - x^8 - x^9 - x^10 - x^11 - x^12 - x^13 - x^14 - x^15 - x^16 - x^17 - x^18 - x^19 - x^20 + O(x^21)
```

### construction

**Description:**

Return the construction of `self`.

**Examples:**

```python
sage: LaurentPolynomialRing(QQ,2,'x,y').construction()
(LaurentPolynomialFunctor,
 Univariate Laurent Polynomial Ring in x over Rational Field)
```

### fraction_field

**Description:**

The fraction field is the same as the fraction field of the polynomial ring.

**Examples:**

```python
sage: L.<x> = LaurentPolynomialRing(QQ)
sage: L.fraction_field()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: (x^-1 + 2) / (x - 1)
(2*x + 1)/(x^2 - x)```
gen \((i=0)\)
Returns the \(i^{th}\) generator of self. If \(i\) is not specified, then the first generator will be returned.

**EXAMPLES:**

```
sage: LaurentPolynomialRing(QQ, 2, 'x').gen()
x0
sage: LaurentPolynomialRing(QQ, 2, 'x').gen(0)
x0
sage: LaurentPolynomialRing(QQ, 2, 'x').gen(1)
x1
```

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 trac ticket #7530:

```
sage: L = LaurentPolynomialRing(ZZ, 'X')
sage: L['Y']
Univariate Polynomial Ring in Y over Univariate Laurent Polynomial Ring in X over Integer Ring
```

is_noetherian ()

Returns True if self is Noetherian.

6.1. Ring of Laurent Polynomials
EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ,2,'x').is_noetherian()
Traceback (most recent call last):
...  
NotImplementedError
```

**krull_dimension()**

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ,2,'x').krull_dimension()
Traceback (most recent call last):
...  
NotImplementedError
```

**ngens()**

Return the number of generators of self.

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ,2,'x').ngens()
2
sage: LaurentPolynomialRing(QQ,1,'x').ngens()
1
```

**polynomial_ring()**

Returns the polynomial ring associated with self.

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ,2,'x').polynomial_ring()
Multivariate Polynomial Ring in x0, x1 over Rational Field
sage: LaurentPolynomialRing(QQ,1,'x').polynomial_ring()
Multivariate Polynomial Ring in x over Rational Field
```

**random_element** *(low_degree=-2, high_degree=2, terms=5, choose_degree=False, *args, **kwds)*

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ,2,'x').random_element()
Traceback (most recent call last):
...  
NotImplementedError
```

**remove_var** *(var)*

EXAMPLES:

```python
sage: R = LaurentPolynomialRing(QQ,'x,y,z')
sage: R.remove_var('x')
Multivariate Laurent Polynomial Ring in y, z over Rational Field
sage: R.remove_var('x').remove_var('y')
Univariate Laurent Polynomial Ring in z over Rational Field
```

**term_order()**

Returns the term order of self.

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ,2,'x').term_order()
Degree reverse lexicographic term order
```
variable_names_recursive (depth=+Infinity)
Return the list of variable names of this ring and its base rings, as if it were a single multi-variate Laurent polynomial.

INPUT:
• depth – an integer or Infinity.

OUTPUT:
A tuple of strings.

EXAMPLES:

```
sage: T = LaurentPolynomialRing(QQ, 'x')
sage: S = LaurentPolynomialRing(T, 'y')
sage: R = LaurentPolynomialRing(S, 'z')
sage: R.variable_names_recursive()
('x', 'y', 'z')
sage: R.variable_names_recursive(2)
('y', 'z')
```

class sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_mpair(R)

Bases:
sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic

EXAMPLES:

```
sage: L = LaurentPolynomialRing(QQ,2,'x')
sage: type(L)
<class 'sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_mpair_with_category'>
sage: L == loads(dumps(L))
True
```

Element

alias of sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair

monomial (*args)
Return the monomial whose exponents are given in argument.

EXAMPLES:

```
sage: L = LaurentPolynomialRing(QQ, 'x', 2)
sage: L.monomial(-3, 5)
x0^-3*x1^5
sage: L.monomial(1, 1)
x0*x1
sage: L.monomial(0, 0)
1
sage: L.monomial(-2, -3)
x0^-2*x1^-3
sage: x0, x1 = L.gens()
sage: L.monomial(-1, 2) == x0^-1 * x1^2
True
sage: L.monomial(1, 2, 3)
Traceback (most recent call last):
```
TypeError: tuple key must have same length as ngens

class sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_univariate(R)
Bases: sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic

EXAMPLES:

```
sage: L = LaurentPolynomialRing(QQ, 'x')
sage: type(L)
<class 'sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_univariate_with_category'>
sage: L == loads(dumps(L))
True
```

Element
alias of sage.rings.polynomial.laurent_polynomial.LaurentPolynomial

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
```

6.2 Elements of Laurent polynomial rings

class sage.rings.polynomial.laurent_polynomial.LaurentPolynomial
Bases: sage.structure.element.CommutativeAlgebraElement

Base class for Laurent polynomials.

```
change_ring(R)
Return a copy of this Laurent polynomial, with coefficients in R.

EXAMPLES:

```
sage: R.<x> = LaurentPolynomialRing(QQ)
sage: a = x^2 + 3*x^3 + 5*x^-1
sage: a.change_ring(GF(3))
2*x^-1 + x^2
```

Check that trac ticket #22277 is fixed:

```
sage: R.<x, y> = LaurentPolynomialRing(QQ)
sage: a = 2*x^-2 + 3*x^3 + 4*x^-1
```

dict()
Abstract dict method.

EXAMPLES:

```python
sage: R.<x> = LaurentPolynomialRing(ZZ)
sage: from sage.rings.polynomial.laurent_polynomial import LaurentPolynomial
sage: LaurentPolynomial.dict(x)
Traceback (most recent call last):
...,
NotImplementedError
```

hamming_weight()
Return the hamming weight of self.
The hamming weight is number of non-zero coefficients and also known as the weight or sparsity.

EXAMPLES:

```python
sage: R.<x> = LaurentPolynomialRing(ZZ)
sage: f = x^3 - 1
sage: f.hamming_weight()
2
```

number_of_terms()
Abstract method for number of terms

EXAMPLES:

```python
sage: R.<x> = LaurentPolynomialRing(ZZ)
sage: from sage.rings.polynomial.laurent_polynomial import LaurentPolynomial
sage: LaurentPolynomial.number_of_terms(x)
Traceback (most recent call last):
...,
NotImplementedError
```

class sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair
Bases: sage.rings.polynomial.laurent_polynomial.LaurentPolynomial
Multivariate Laurent polynomials.

coefficient(mon)
Return the coefficient of mon in self, where mon must have the same parent as self.
The coefficient is defined as follows. If f is this polynomial, then the coefficient \( c_m \) is sum:

\[
    c_m := \sum_{T} \frac{T}{m}
\]

where the sum is over terms \( T \) in \( f \) that are exactly divisible by \( m \).
A monomial \( m(x, y) \) 'exactly divides' \( f(x, y) \) if \( m(x, y)|f(x, y) \) and neither \( x \cdot m(x, y) \) nor \( y \cdot m(x, y) \) divides \( f(x, y) \).

INPUT:
- mon – a monomial
OUTPUT:
Element of the parent of `self`.

**Note:** To get the constant coefficient, call `constant_coefficient()`.

**EXAMPLES:**

```python
sage: P.<x,y> = LaurentPolynomialRing(QQ)
```

The coefficient returned is an element of the parent of `self`; in this case, `P`.

```python
sage: f = 2 * x * y
sage: c = f.coefficient(x*y); c
2
sage: c.parent()
Multivariate Laurent Polynomial Ring in x, y over Rational Field
```

```python
sage: P.<x,y> = LaurentPolynomialRing(QQ)
sage: f = (y^2 - x^9 - 7*x*y^2 + 5*x*y)*x^-3; f
-x^6 - 7*x^-2*y^2 + 5*x^-2*y + x^-3*y^2
sage: f.coefficient(y)
5*x^-2
sage: f.coefficient(y^2)
-7*x^-2 + x^-3
sage: f.coefficient(x*y)
0
sage: f.coefficient(x^-2)
-7*y^2 + 5*y
sage: f.coefficient(x^-2*y^2)
-7
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 `self` in a list.

The returned list is decreasingly ordered by the term ordering of `self.parent()`.

**EXAMPLES:**

```python
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:**

```python
sage: P.<x,y> = LaurentPolynomialRing(QQ)
sage: f = (y^2 - x^9 - 7*x*y^2 + 5*x*y)*x^-3; f
-x^6 - 7*x^-2*y^2 + 5*x^-2*y + x^-3*y^2
```

(continues on next page)
degree (x=None)
Return the degree of x in self.

EXAMPLES:

```
sage: R.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.degree(x)
7
sage: f.degree(y)
1
sage: f.degree(z)
0
```

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

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

See also:

_ derivative()

EXAMPLES:

```
sage: R = LaurentPolynomialRing(ZZ, 'x, y')
sage: x, y = R.gens()
sage: t = x**4*y + x*y + y + 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: sorted(f.dict().items())
[((3, 1, 0), 3), ((4, 0, -2), 2), ((6, -7, 0), 1), ((7, 0, -1), 4)]
```

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

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

See also:

_ derivative()

EXAMPLES:
```python
sage: R = LaurentPolynomialRing(ZZ, 'x, y')
sage: x, y = R.gens()
sage: t = x**4*y + x*y + y + x**(-1) + y**(-3)
sage: t.derivative(x, x)
12*x^2*y + 2*x^-3
sage: t.derivative(y, 2)
12*y^-5
```

**differentiate(**args**)

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

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

See also:

*_derivative_*

**EXAMPLES:**

```python
sage: R = LaurentPolynomialRing(ZZ, 'x, y')
sage: x, y = R.gens()
sage: t = x**4*y + x*y + y + x**(-1) + y**(-3)
sage: t.derivative(x, x)
12*x^2*y + 2*x^-3
sage: t.derivative(y, 2)
12*y^-5
```

**exponents**()

Return a list of the exponents of self.

**EXAMPLES:**

```python
sage: L.<w,z> = LaurentPolynomialRing(QQ)
sage: a = w^2*z^-1 + 3; a
w^2*z^-1 + 3
sage: e = a.exponents()
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:**

```python
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.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:**

```python
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.has_any_inverse()
True
```

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:

```python
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.has_inverse_of(0)
False
sage: f.has_inverse_of(1)
True
sage: f.has_inverse_of(2)
True
```

is_constant()

Return whether this Laurent polynomial is constant.

EXAMPLES:

```python
sage: L.<a, b> = LaurentPolynomialRing(QQ)
sage: L(0).is_constant()
True
sage: L(42).is_constant()
True
sage: a.is_constant()
False
sage: (1/b).is_constant()
False
```

is_monomial()

Return True if self is a monomial.

EXAMPLES:

```python
sage: k.<y,z> = LaurentPolynomialRing(QQ)
sage: z.is_monomial()
True
sage: k(1).is_monomial()
True
sage: (z+1).is_monomial()
False
sage: (z^-2909).is_monomial()
True
sage: (38*z^-2909).is_monomial()
False
```

is_square(root=False)

Test whether this Laurent polynomial is a square root.

6.2. Elements of Laurent polynomial rings 535
INPUT:

- root - boolean (default False) - if set to True then return a pair (True, sqrt) with sqrt a square root of this Laurent polynomial when it exists or (False, None).

EXAMPLES:

```
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: p = (1 + x*y + z^-3)
sage: (p**2).is_square()
True
sage: (p**2).is_square(root=True)
(True, x*y + 1 + z^-3)
sage: x.is_square()
False
sage: x.is_square(root=True)
(False, None)
sage: (x**-4 * (1 + z)).is_square(root=False)
False
sage: (x**-4 * (1 + z)).is_square(root=True)
(False, None)
```

**is_unit()**

Return True if self is a unit.

The ground ring is assumed to be an integral domain.

This means that the Laurent polynomial is a monomial with unit coefficient.

EXAMPLES:

```
sage: L.<x,y> = LaurentPolynomialRing(QQ)
sage: (x*y/2).is_unit()
True
sage: (x + y).is_unit()
False
sage: (L.zero()).is_unit()
False
sage: (L.one()).is_unit()
True
sage: L.<x,y> = LaurentPolynomialRing(ZZ)
sage: (2*x*y).is_unit()
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
```
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
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
sage: P.<x,y> = LaurentPolynomialRing(QQ)
sage: f = (y^2 - x^9 - 7*x*y^3 + 5*x*y)*x^-3
sage: sorted(f.monomials())
[x^-3*y^2, x^-2*y, x^-2*y^3, x^6]
```

number_of_terms ()

Return the number of non-zero coefficients of self.

Also called weight, hamming weight or sparsity.

EXAMPLES:

```sage
sage: R.<x, y> = LaurentPolynomialRing(ZZ)
sage: f = x^3 - y
sage: f.number_of_terms()
2
sage: R(0).number_of_terms()
0
sage: f = (x+1/y)^100
sage: f.number_of_terms()
101
```

The method hamming_weight() is an alias:

```sage
sage: f.hamming_weight()
101
```
**quo_rem** (*right*)

Divide this Laurent polynomial by *right* and return a quotient and a remainder.

**INPUT:**

- *right* – a Laurent polynomial

**OUTPUT:**

A pair of Laurent polynomials.

**EXAMPLES:**

```sage
sage: R.<s, t> = LaurentPolynomialRing(QQ)
sage: (s^2-t^2).quo_rem(s-t)
(s + t, 0)
sage: (s^-2-t^2).quo_rem(s-t)
(s + t, -s^4 + 1)
sage: (s^-2-t^2).quo_rem(s^-1-t)
t + s^-1, 0
```

**subs** (*in_dict=None, **kwds*)

Substitute some variables in this Laurent polynomial.

Variable/value pairs for the substitution may be given as a dictionary or via keyword-value pairs. If both are present, the latter take precedence.

**INPUT:**

- *in_dict* – dictionary (optional)
- **kwargs** – keyword arguments

**OUTPUT:**

A Laurent polynomial.

**EXAMPLES:**

```sage
sage: L.<x, y, z> = LaurentPolynomialRing(QQ)
sage: f = x + 2*y + 3*z
2*y + 3*z + 1
sage: f.subs(x=1)
x + 3*z + 2
sage: f.subs(y=1)
x + 2*z + 3
sage: f.subs(z=1)
x + 2*y + 3
sage: f.subs(x=1, y=1, z=1)
6
sage: f = x^-1
1/2
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
```
univariate_polynomial($R=None$)
Returns a univariate polynomial associated to this multivariate polynomial.

INPUT:

- $R$ - (default: None) a univariate Laurent polynomial ring

If this polynomial is not in at most one variable, then a ValueError exception is raised. The new polynomial is over the same base ring as the given LaurentPolynomial and in the variable $x$ if no ring $R$ is provided.

EXAMPLES:

```sage
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
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
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

A univariate Laurent polynomial in the form of $t^n \cdot f$ where $f$ is a polynomial in $t$.

INPUT:

- $parent$ – a Laurent polynomial ring
- $f$ – a polynomial (or something can be coerced to one)
• \(n\) – (default: 0) an integer

**AUTHORS:**

• Tom Boothby (2011) copied this class almost verbatim from `laurent_series_ring_element.pyx`, so most of the credit goes to William Stein, David Joyner, and Robert Bradshaw

• Travis Scrimshaw (09-2013): Cleaned-up and added a few extra methods

**coefficients()**

Return the nonzero coefficients of `self`.

**EXAMPLES:**

```python
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:**

```python
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: f = 3*t^-2 - t^-1 + 3 + t^2
sage: f.constant_coefficient()
3
sage: g = -2*t^-2 + t^-1 + 3*t
sage: g.constant_coefficient()
0
```

**degree()**

Return the degree of `self`.

**EXAMPLES:**

```python
sage: R.<x> = LaurentPolynomialRing(ZZ)
sage: g = x^2 - x^4
sage: g.degree()
4
sage: g = -10/x^5 + x^2 - x^7
sage: g.degree()
7
```

**derivative(*args)**

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

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

**See also:**

omers\_derivative()

**EXAMPLES:**

```python
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
```

(continues on next page)
\section{Elements of Laurent polynomial rings}

The formal integral of this Laurent series with 0 constant term.

\section{Sage Reference Manual: Polynomials, Release 9.0}

\begin{Verbatim}
\begin{Verbatim}
sage: g.derivative(x)
-10*x^-11 - 1 + 2*x - 4*x^3
\end{Verbatim}
\end{Verbatim}

\begin{Verbatim}
\begin{Verbatim}
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
\end{Verbatim}
\end{Verbatim}

\begin{Verbatim}
dict()
Return a dictionary representing self.

\begin{Verbatim}
EXAMPLES::
sage: R.<x,y> = ZZ[] sage: Q.<t> = LaurentPolynomialRing(R)
sage: f = (x^3 + y*t^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}
\end{Verbatim}

\begin{Verbatim}
exponents()
Return the exponents appearing in self with nonzero coefficients.

\begin{Verbatim}
EXAMPLES:

\begin{Verbatim}
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: f = -5/t^2 + t + t^2 - 10/3*t^3
sage: f.exponents()
[-2, 1, 2, 3]
\end{Verbatim}
\end{Verbatim}

\begin{Verbatim}
factor()
Return a Laurent monomial (the unit part of the factorization) and a factored polynomial.

\begin{Verbatim}
EXAMPLES:

\begin{Verbatim}
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)
\end{Verbatim}
\end{Verbatim}

\begin{Verbatim}
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.

\begin{Verbatim}
EXAMPLES:

\begin{Verbatim}
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
\end{Verbatim}
\end{Verbatim}

\begin{Verbatim}
integral()
The formal integral of this Laurent series with 0 constant term.

\begin{Verbatim}
EXAMPLES:
\end{Verbatim}
\end{Verbatim}
The integral may or may not be defined if the base ring is not a field.

```python
sage: t = LaurentPolynomialRing(ZZ, 't').0
sage: f = 2*t^-3 + 3*t^2
sage: f.integral()
-t^-2 + t^3
```

```python
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:

```python
sage: t = LaurentPolynomialRing(ZZ,'t').0
sage: f = -1/t^3 - 31/t
sage: f.integral()
Traceback (most recent call last):
  ... ArithmeticError: the integral of is not a Laurent polynomial, since t^-1 has → nonzero coefficient
```

Another example with just one negative coefficient:

```python
sage: A.<t> = LaurentPolynomialRing(QQ)
sage: f = -2*t^(-4)
```

```python
sage: f.integral()
2/3*t^-3
```

```python
sage: f.integral().derivative() == f
True
```

**inverse_of_unit()**

Return the inverse of `self` if a unit.

**EXAMPLES:**

```python
sage: R.<t> = LaurentPolynomialRing(QQ)
```

```python
sage: (t^-2).inverse_of_unit()
t^2
```

```python
sage: (t + 2).inverse_of_unit()
Traceback (most recent call last):
  ... ArithmeticError: element is not a unit
```

**is_constant()**

Return whether this Laurent polynomial is constant.

**EXAMPLES:**

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

```python
sage: x.is_constant()
False
```

```python
sage: R.one().is_constant()
True
```

```python
sage: (x^-2).is_constant()
False
```

```python
sage: (x^2).is_constant()
```

(continues on next page)
False

```
sage: (x^-2 + 2).is_constant()
False
sage: R(0).is_constant()
True
sage: R(42).is_constant()
True
sage: x.is_constant()
False
sage: (1/x).is_constant()
False
```

**is_monomial()**

Return `True` if `self` is a monomial; that is, if `self` is $x^n$ for some integer $n$.

**EXAMPLES:**

```
sage: k.<z> = LaurentPolynomialRing(QQ)
sage: z.is_monomial()
True
sage: k(1).is_monomial()
True
sage: (z+1).is_monomial()
False
sage: (z^-2909).is_monomial()
True
sage: (38*z^-2909).is_monomial()
False
```

**is_square**(root=False)

Return whether this Laurent polynomial is a square. If `root` is set to `True` then return a pair made of the boolean answer together with `None` or a square root.

**EXAMPLES:**

```
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: R.one().is_square()
True
sage: R(2).is_square()
False
sage: t.is_square()
False
sage: (t**-2).is_square()
True
```

Usage of the `root` option:

```
sage: p = (1 + t^-1 - 2*t^3)
```

```
sage: p.is_square(root=True)
(False, None)
sage: (p**2).is_square(root=True)
(True, -t^-1 - 1 + 2*t^3)
```

The answer is dependent of the base ring:
\begin{Verbatim}
sage: S.<u> = LaurentPolynomialRing(QQbar)
sage: (2 + 4*t + 2*t^2).is_square()
False
sage: (2 + 4*u + 2*u^2).is_square()
True
\end{Verbatim}

**is_unit()**

Return True if this Laurent polynomial is a unit in this ring.

**EXAMPLES:**

\begin{Verbatim}
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
\end{Verbatim}

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:**

\begin{Verbatim}
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
\end{Verbatim}

**number_of_terms()**

Return the number of non-zero coefficients of self.

Also called weight, hamming weight or sparsity.

**EXAMPLES:**

\begin{Verbatim}
sage: R.<x> = LaurentPolynomialRing(ZZ)
sage: f = x^3 - 1
sage: f.number_of_terms()
2
sage: R(0).number_of_terms()
0
sage: f = (x+1)^100
sage: f.number_of_terms()
101
\end{Verbatim}
The method `hamming_weight()` is an alias:

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

`polynomial_construction()`

Return the polynomial and the shift in power used to construct the Laurent polynomial $t^n u$.

**OUTPUT:**

A tuple $(u, n)$ where $u$ is the underlying polynomial and $n$ is the power of the exponent shift.

**EXAMPLES:**

```python
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:**

```python
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)
(0, 1 + 3*t^2 + t^3)
```

`residue()`

Return the residue of `self`.

The residue is the coefficient of $t^{-1}$.

**EXAMPLES:**

```python
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:**

```python
sage: R.<t> = LaurentPolynomialRing(QQ['y'])
sage: f = (t+t^-1)^4; f
```

(continues on next page)
\[ t^{-14} + 4t^{-12} + 6t^{-10} + 4t^{-8} + t^{-6} \]

```python
sage: f << 4
1 + 4t^2 + 6t^4 + 4t^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:**

```python
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** \((p=\text{None})\)

Return the valuation of \(self\).

The valuation of a Laurent polynomial \(t^n u\) is \(n\) plus the valuation of \(u\).

**EXAMPLES:**

```python
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:**

```python
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:**

```python
sage: R.<x> = LaurentPolynomialRing(QQ)
sage: f = 1/x + x^2 + 3*x^4
sage: f.variables()
(x,)
```
6.3 MacMahon’s Partition Analysis Omega Operator

This module implements *MacMahon’s Omega Operator* [Mac1915], which takes a quotient of Laurent polynomials and removes all negative exponents in the corresponding power series.

6.3.1 Examples

In the following example, all negative exponents of \( \mu \) are removed. The formula

\[
\Omega \geq \frac{1}{(1-x\mu)(1-y/\mu)} = \frac{1}{(1-x)(1-xy)}
\]

can be calculated and verified by

```sage
L.<mu, x, y> = LaurentPolynomialRing(ZZ)
sage: MacMahonOmega(mu, 1, [1 - x*mu, 1 - y/mu])
1 * (-x + 1)^-1 * (-x*y + 1)^-1
```

6.3.2 Various

AUTHORS:

- Daniel Krenn (2016)

ACKNOWLEDGEMENT:

- Daniel Krenn is supported by the Austrian Science Fund (FWF): P 24644-N26.

6.3.3 Functions

```python
sage.rings.polynomial.omega.MacMahonOmega(var, expression, denominator=None, op=<built-in function ge>, Factorization_sort=False, Factorization_simplify=True)
```

Return \( \Omega_{op} \) of expression with respect to var.

To be more precise, calculate

\[
\Omega_{op} = \frac{n}{d_1 \ldots d_n}
\]

for the numerator \( n \) and the factors \( d_1, \ldots, d_n \) of the denominator, all of which are Laurent polynomials in var and return a (partial) factorization of the result.

INPUT:

- var – a variable or a representation string of a variable
- expression – a Factorization of Laurent polynomials or, if denominator is specified, a Laurent polynomial interpreted as the numerator of the expression
- denominator – a Laurent polynomial or a Factorization (consisting of Laurent polynomial factors) or a tuple/list of factors (Laurent polynomials)
- op – (default: operator.ge) an operator
  At the moment only operator.ge is implemented.
• Factorization_sort (default: False) and Factorization_simplify (default: True) – are passed on to sage.structure.factorization.Factorization when creating the result

OUTPUT:
A (partial) Factorization of the result whose factors are Laurent polynomials

Note: The numerator of the result may not be factored.

REFERENCES:
• [Mac1915]
• [APR2001]

EXAMPLES:

```
sage: L.<mu, x, y, z, w> = LaurentPolynomialRing(ZZ)
sage: MacMahonOmega(mu, 1, [1 - x*mu, 1 - y/mu])
1 * (-x + 1)^-1 * (-x*y + 1)^-1

sage: MacMahonOmega(mu, 1, [1 - x*mu, 1 - y/mu, 1 - z/mu])
1 * (-x + 1)^-1 * (-x*y + 1)^-1 * (-x*z + 1)^-1

sage: MacMahonOmega(mu, 1, [1 - x*mu, 1 - y/mu, 1 - z/mu])
(-x*y*z + 1) * (-x + 1)^-1 * (-y + 1)^-1 * (-x*z + 1)^-1 * (-y*z + 1)^-1

sage: MacMahonOmega(mu, 1, [1 - x*mu, 1 - y/mu^2])
1 * (-x + 1)^-1 * (-x^2*y + 1)^-1

sage: MacMahonOmega(mu, 1, [1 - x*mu^2, 1 - y/mu])
(x*y + 1) * (-x + 1)^-1 * (-x*y^2 + 1)^-1

sage: MacMahonOmega(mu, 1, [1 - x*mu, 1 - y/mu^2, 1 - z/mu])
(-x^2*y*z - x*y^2*z + x*y*z + 1) * (-x + 1)^-1 * (-y + 1)^-1 * (-x^2*z + 1)^-1 * (-y^2*z + 1)^-1

sage: MacMahonOmega(mu, 1, [1 - x*mu, 1 - y/mu^3])
1 * (-x + 1)^-1 * (-x^3*y + 1)^-1

sage: MacMahonOmega(mu, 1, [1 - x*mu^2, 1 - y*mu, 1 - z/mu])
(x*y*z + x*y + x*z + 1) * (-x + 1)^-1 * (-y + 1)^-1 * (-x*y^2 + 1)^-1 * (-x*z^2 + 1)^-1

sage: MacMahonOmega(mu, 1, [1 - x*mu^2, 1 - y*mu, 1 - z*mu, 1 - w/mu])
(x^2*y*z*w + x*y*z*w - x*y*w - x*z*w - y*z*w + 1) * (-x + 1)^-1 * (-y + 1)^-1 * (-z + 1)^-1 * (-w + 1)^-1

sage: MacMahonOmega(mu, 1, [1 - x*mu, 1 - y*mu, 1 - z/mu, 1 - w/mu])
(x^2*y*z*w + x*y^2*z*w - x*y*z*w - x*y*z - x*y*w + 1) * (-x + 1)^-1 * (-y + 1)^-1 * (-z + 1)^-1 * (-w + 1)^-1
```

We demonstrate the different allowed input variants:

\[
(-x*y^2 - x*y + y^2 + y + 1) * (-x + 1)^{-1} * (-x*y + 1)^{-1}
\]

```python
sage: MacMahonOmega(mu, mu^2, [1 - x*mu, 1 - y/mu])
\(-x*y^2 - x*y + y^2 + y + 1) * (-x + 1)^{-1} * (-x*y + 1)^{-1}
```

We demonstrate the different allowed input variants:

\[
(-x*z + 1)^{-1} * (-x*w + 1)^{-1} * (-y*z + 1)^{-1} * (-y*w + 1)^{-1}
\]

```python
sage: MacMahonOmega(mu, mu^-2, [1 - x*mu, 1 - y/mu])
x^2 * (-x + 1)^{-1} * (-x*y + 1)^{-1}
```

Return \(\Omega_\geq\) of the expression specified by the input.

To be more precise, calculate

\[
\Omega_\geq \mu^a / (1 - z_0 \mu^e_0)(1 - z_{n-1} \mu^{e_{n-1}})
\]

and return its numerator and a factorization of its denominator. Note that \(z_0, \ldots, z_{n-1}\) only appear in the output, but not in the input.

INPUT:

- \(a\) – an integer
- \(exponents\) – a tuple of integers

OUTPUT:

A pair representing a quotient as follows: Its first component is the numerator as a Laurent polynomial, its second component a factorization of the denominator as a tuple of Laurent polynomials, where each Laurent polynomial \(z\) represents a factor \(1 - z\).

The parents of these Laurent polynomials is always a Laurent polynomial ring in \(z_0, \ldots, z_{n-1}\) over \(\mathbb{Z}\), where \(n\) is the length of \(exponents\).

EXAMPLES:
sage: from sage.rings.polynomial.omega import Omega_ge
sage: Omega_ge(0, (1, -2))
(1, (z0, z0^2*z1))
sage: Omega_ge(0, (1, -3))
(1, (z0, z0^3*z1))
sage: Omega_ge(0, (1, -4))
(1, (z0, z0^4*z1))
sage: Omega_ge(0, (2, -1))
(z0*z1 + 1, (z0, z0*z1^2))
sage: Omega_ge(0, (3, -1))
(z0^2 + z0*z1 + 1, (z0, z0*z1^3))
sage: Omega_ge(0, (4, -1))
(z0^3 + z0^2*z1^2 + z0*z1 + 1, (z0, z0*z1^4))

sage: Omega_ge(0, (1, 1, -2))
(-z0^2*z1*z2 - z0*z1^2*z2 + z0*z1*z2 + 1, (z0, z1, z0^2*z2, z1^2*z2))
sage: Omega_ge(0, (2, -1, -1))
(z0*z1*z2 + z0*z1 + z0*z2 + 1, (z0, z0*z1^2, z0*z2^2))
sage: Omega_ge(0, (2, 1, -1))
(-z0*z1*z2^2 - z0*z1*z2 + z0*z2 + 1, (z0, z1, z0*z2^2, z1*z2))
sage: Omega_ge(0, (2, -2))
(-z0*z1 + 1, (z0, z0*z1, z0*z1))
sage: Omega_ge(0, (2, -3))
(z0^2 + z0*z1, (z0, z0^3*z1^2))
sage: Omega_ge(0, (3, 1, -3))
(-z0^3*z1^3*z2^3 + 2*z0^2*z1^3*z2^2 - z0*z1^3*z2 + z0^2*z2^2 - 2*z0*z2 + 1,
 (z0, z1, z0*z2, z0*z2, z1^3*z2))
sage: Omega_ge(0, (3, 6, -1))
(-z0*z1*z2^8 - z0*z1*z2^7 - z0*z1*z2^6 - z0*z1*z2^5 - z0*z1*z2^4 +
 z1*z2^5 - z0*z1*z2^3 + z1*z2^4 - z0*z1*z2^2 + z1*z2^3 -
 z0*z1*z2 + z0^2*z2^2 + z1*z2^2 + z0*z2 + z1*z2 + 1,
 (z0, z1, z0*z2^3, z1^3*z2^6))

sage: from sage.rings.polynomial.omega import homogenous_symmetric_function
sage: homogenous_symmetric_function(j, x)
Return a complete homogeneous symmetric polynomial (Wikipedia article Complete_homogeneous_symmetric_polynomial).

INPUT:

- j – the degree as a nonnegative integer
- x – an iterable of variables

OUTPUT:

A polynomial of the common parent of all entries of x

EXAMPLES:

sage: from sage.rings.polynomial.omega import homogenous_symmetric_function
sage: P = PolynomialRing(ZZ, 'X', 3)
sage: homogenous_symmetric_function(0, P.gens())
1
sage: homogenous_symmetric_function(1, P.gens())
(continues on next page)
sage.rings.polynomial.omega.partition(items, predicate=<class 'bool'>)

Split items into two parts by the given predicate.

**INPUT:**

- **item** – an iterator
- **predicate** – a function

**OUTPUT:**

A pair of iterators; the first contains the elements not satisfying the predicate, the second the elements satisfying the predicate.

**ALGORITHM:**

Source of the code: http://nedbatchelder.com/blog/201306/filter_a_list_into_two_parts.html

**EXAMPLES:**

```
sage: from sage.rings.polynomial.omega import partition
sage: E, O = partition(srange(10), is_odd)
sage: tuple(E), tuple(O)
((0, 2, 4, 6, 8), (1, 3, 5, 7, 9))
```
7.1 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.

INPUT:

- \( R \), the base ring. It has to be a commutative ring, and in some applications it must even be a field
- \( \text{names} \), a list of generator names. Generator names must be alpha-numeric.
- \( \text{order} \) (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:

\[
X.\text{gen}(i)[m] > X.\text{gen}(j)[n] \text{ if and only if } i<j \text{ or } (i=j \text{ and } m>n)
\]

We provide a 'dense' and a 'sparse' implementation. In the dense implementation, the Infinite Polynomial Ring carries a finite polynomial ring that comprises all variables up to the maximal index that has been used so far. This is potentially a very big ring and may also comprise many variables that are not used.

In the sparse implementation, we try to keep the underlying finite polynomial rings small, using only those variables that are really needed. By default, we use the dense implementation, since it usually is much faster.

EXAMPLES:

```sage
sage: X.<x,y> = InfinitePolynomialRing(ZZ, implementation='sparse')
sage: A.<alpha,beta> = InfinitePolynomialRing(QQ, order='deglex')
sage: f = x[5] + 2; f
```


\( x_5 + 2 \)

\texttt{sage}: \ g = 3*y[1]; \ g

\texttt{3*y_1}

It has some advantages to have an underlying ring that is not univariate. Hence, we always have at least two variables:

\texttt{sage}: \ g._p.parent()

\texttt{Multivariate Polynomial Ring in y_1, y_0 over Integer Ring}

\texttt{sage}: \ f2 = alpha[5] + 2; \ f2
\texttt{alpha_5 + 2}

\texttt{sage}: \ g2 = 3*beta[1]; \ g2
\texttt{3*beta_1}

\texttt{sage}: \ A.polynomial_ring()

\texttt{Multivariate Polynomial Ring in alpha_5, alpha_4, alpha_3, alpha_2, alpha_1, alpha_0, \ldots, beta_5, beta_4, beta_3, beta_2, beta_1, beta_0 over Rational Field}

Of course, we provide the usual polynomial arithmetic:

\texttt{sage}: \ f + g

\texttt{x_5 + 3*y_1 + 2}

\texttt{sage}: \ p = x[10]^2*(f+g); \ p
\texttt{x_10^2*x_5 + 3*x_10^2*y_1 + 2*x_10^2}

\texttt{sage}: \ p2 = alpha[10]^2*(f2+g2); \ p2
\texttt{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:

\texttt{sage}: \ P = Permutation(((10,1)))
\texttt{sage}: \ p^P

\texttt{x_5*x_1^2 + 3*x_1^2*y_10 + 2*x_1^2}

\texttt{sage}: \ p2^P

\texttt{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 \textit{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:

\texttt{sage}: \ J = A*(alpha[1]*beta[2])
\texttt{sage}: \ J.groebner_basis()

\[ \texttt{[alpha_1*beta_2, alpha_2*beta_1]} \]

For more details, see \textit{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:

\texttt{sage}: \ A.<a,b,c> = InfinitePolynomialRing(ZZ)
\texttt{sage}: \ B.<b,c,d> = InfinitePolynomialRing(A)
\texttt{sage}: \ B

\texttt{Infinite polynomial ring in a, b, c, d over Integer Ring}
This is also allowed if finite polynomial rings are involved:

\[
\begin{align*}
\text{sage: } & A.<a_3,a_1,b_1,c_2,c_0> = \mathbb{Z}\[] \\
\text{sage: } & B.<b,c,d> = \text{InfinitePolynomialRing}(A, \text{order}='\text{degrevlex}') \\
\text{sage: } & B \\
\text{Infinite polynomial ring in } b, c, d \text{ over Multivariate Polynomial Ring in } a_3, a_1 \ldots \text{over Integer Ring}
\end{align*}
\]

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 \textit{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:

\[
\begin{align*}
\text{sage: } & X.<x,y_1> = \mathbb{Z}\[] \\
\text{sage: } & Y.<x,z> = \text{InfinitePolynomialRing}(X) \\
\text{Note that } X \text{ is not merged into } Y; \text{ this is since the monomial order of } X \text{ is } \text{degrevlex}, \text{ but of } Y \text{ is } \text{lex}.
\end{align*}
\]

The variable \( x \) of \( X \) can still be interpreted in \( Y \), although the first generator of \( Y \) is called \( x \) as well:

\[
\begin{align*}
\text{sage: } & x \\
\text{sage: } & X('x') \\
\text{sage: } & \text{Y(X('x'))} \\
\text{sage: } & Y('x') \\
\text{But there is only merging if the resulting monomial order is uniquely determined. This is not the case in the following examples, and thus an error is raised:}
\end{align*}
\]

\[
\begin{align*}
\text{sage: } & X.<y_1,x> = \mathbb{Z}\[] \\
\text{sage: } & Y.<y,z> = \text{InfinitePolynomialRing}(X) \\
\text{Traceback (most recent call last):} \\
\ldots \\
\text{CoercionException: Overlapping variables } (('y', 'z'),['y_1']) \text{ are incompatible} \\
\text{sage: } & Y.<z,y> = \text{InfinitePolynomialRing}(X) \\
\text{Traceback (most recent call last):} \\
\ldots \\
\text{CoercionException: Overlapping variables } (('z', 'y'),['y_1']) \text{ are incompatible} \\
\text{sage: } & X.<x_3,y_1,y_2> = \text{PolynomialRing}(\mathbb{Z}\[], \text{order}=\text{lex}) \\
\text{sage: } & \# y_1 and y_2 would be in opposite order in an Infinite Polynomial Ring \\
\text{sage: } & Y.<y> = \text{InfinitePolynomialRing}(X) \\
\text{Traceback (most recent call last):} \\
\ldots \\
\text{CoercionException: Overlapping variables } (('y'),['y_1', 'y_2']) \text{ are incompatible}
\end{align*}
\]

If the type of monomial orderings (e.g., ‘degrevlex’ versus ‘lex’) or if the implementations don’t match, there is no simplified construction available:

\[
\begin{align*}
\text{sage: } & X.<x,y> = \text{InfinitePolynomialRing}(\mathbb{Z}\[]) \\
\text{sage: } & Y.<z> = \text{InfinitePolynomialRing}(X, \text{order}=\text{degrevlex'})
\end{align*}
\]
sage: Y
Infinite polynomial ring in z over Infinite polynomial ring in x, y over Integer Ring
sage: Y.<z> = InfinitePolynomialRing(X, implementation='sparse')
sage: Y
Infinite polynomial ring in z over Infinite polynomial ring in x, y over Integer Ring

```
class sage.rings.polynomial.infinite_polynomial_ring.GenDictWithBasering(parent, start)

    Bases: object

    A dictionary-like class that is suitable for usage in sage_eval.

    This pseudo-dictionary accepts strings as index, and then walks down a chain of base rings of (infinite) polynomial rings until it finds one ring that has the given string as variable name, which is then returned.

    EXAMPLES:

    sage: R.<a,b> = InfinitePolynomialRing(ZZ)
sage: D = R.gens_dict()  # indirect doctest
sage: D['a_15']
a_15
sage: type(_)
<class 'sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_dense'>
sage: sage_eval('3*a_3*b_5-1/2*a_7', D)
-1/2*a_7 + 3*a_3*b_5
```

```
next()

    Return a dictionary that can be used to interpret strings in the base ring of self.

    EXAMPLES:

    sage: R.<a,b> = InfinitePolynomialRing(QQ['t'])
sage: D = R.gens_dict()
sage: D
GenDict of Infinite polynomial ring in a, b over Univariate Polynomial Ring in t over Rational Field
sage: next(D)
GenDict of Univariate Polynomial Ring in t over Rational Field
sage: sage_eval('t^2', next(D))
t^2
```

```
class sage.rings.polynomial.infinite_polynomial_ring.InfiniteGenDict(Gens)

    Bases: object

    A dictionary-like class that is suitable for usage in sage_eval.

    The generators of an Infinite Polynomial Ring are not variables. Variables of an Infinite Polynomial Ring are returned by indexing a generator. The purpose of this class is to return a variable of an Infinite Polynomial Ring, given its string representation.

    EXAMPLES:

    sage: R.<a,b> = InfinitePolynomialRing(ZZ)
sage: D = R.gens_dict()  # indirect doctest
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
```

(continues on next page)
class sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialGen

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
```

create_key (R, names=('x',), order='lex', implementation='dense')

Creates a key which uniquely defines the infinite polynomial ring.

create_object (version, key)

Return the infinite polynomial ring corresponding to the key key.

class sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_dense (R, names, order)
Bases: \( \text{sage.rings.polynomial.infinite_polynomial_ring.} \)
\( \text{InfinitePolynomialRing\_sparse} \)

Dense implementation of Infinite Polynomial Rings

Compared with \( \text{InfinitePolynomialRing\_sparse} \), 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 \( \text{self} \).

**OUTPUT:**

A pair \( F, R \), where \( F \) is a construction functor and \( R \) is a ring, so that \( F(R) \) is \( \text{self} \).

**EXAMPLES:**

```python
sage: R.<x,y> = InfinitePolynomialRing(GF(5))
sage: R.construction()
[InfPoly([x,y], "lex", "dense"), Finite Field of size 5]
```

**polynomial\_ring()**

Return the underlying finite polynomial ring.

**Note:** The ring returned can change over time as more variables are used.

Since the rings are cached, we create here a ring with variable names that do not occur in other doc tests, so that we avoid side effects.

**EXAMPLES:**

```python
sage: X.<xx, yy> = InfinitePolynomialRing(ZZ)
sage: X.polynomial\_ring()
Multivariate Polynomial Ring in xx_0, yy_0 over Integer Ring
sage: a = yy[3]
sage: X.polynomial\_ring()
Multivariate Polynomial Ring in xx_3, xx_2, xx_1, xx_0, yy_3, yy_2, yy_1, yy_→0 over Integer Ring
```

**tensor\_with\_ring\((R)\)**

Return the tensor product of \( \text{self} \) with another ring.

**INPUT:**

\( R \) - a ring.

**OUTPUT:**

An infinite polynomial ring that, mathematically, can be seen as the tensor product of \( \text{self} \) with \( R \).

**NOTE:**

It is required that the underlying ring of \( \text{self} \) coerces into \( R \). Hence, the tensor product is in fact merely an extension of the base ring.

**EXAMPLES:**

```python
sage: R.<a,b> = InfinitePolynomialRing(ZZ, implementation='sparse')
sage: R.tensor\_with\_ring(QQ)
Infinite polynomial ring in a, b over Rational Field
```
The following tests against a bug that was fixed at trac ticket #10468:

```python
sage: R.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
```
```python
sage: R.tensor_with_ring(QQ) is R
True
```

```python
class sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse(R, names, order)
```

**Bases:** sage.rings.ring.CommutativeRing

Sparse implementation of Infinite Polynomial Rings.

An Infinite Polynomial Ring with generators $x_*, y_*, \ldots$ over a field $F$ is a free commutative $F$-algebra generated by $x_0, x_1, x_2, \ldots, y_0, y_1, y_2, \ldots, \ldots$ and is equipped with a permutation action on the generators, namely $x_{P(n)} = x_{P(n)}$, $y_{P(n)} = y_{P(n)}$, 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:

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: Y.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: X is Y
True
```

```python
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.

```python
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:**

```python
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.
A pair \( F, R \), where \( F \) is a construction functor and \( R \) is a ring, so that \( F(R) \) is self.

**EXAMPLES:**

```
sage: R.<x,y> = InfinitePolynomialRing(GF(5))
sage: R.construction()
[InfPoly([x,y], "lex", "dense"), Finite Field of size 5]
```

**gen**(\( i=\text{None} \))

Return the \( i^{th} \) ‘generator’ (see the description in \( \text{ngens()} \)) 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
```

**gens_dict()**

Return a dictionary-like object containing the infinitely many \{\text{var_name:variable}\} pairs.

**EXAMPLES:**

```
sage: R = InfinitePolynomialRing(ZZ, 'a')
sage: D = R.gens_dict()
sage: D
GenDict of Infinite polynomial ring in a over Integer Ring
sage: D['a_5']
a_5
```

**is_field\(**args, **kwds\)**

Return False since Infinite Polynomial Rings are never fields.

Since Infinite Polynomial Rings must have at least one generator, they have infinitely many variables and thus never are fields.

**EXAMPLES:**

```
sage: R.<x, y> = InfinitePolynomialRing(QQ)
sage: R.is_field()
False
```

**is_integral_domain\(**args, **kwds\)**

An infinite polynomial ring is an integral domain if and only if the base ring is. Arguments are passed to is_integral_domain method of base ring.

**EXAMPLES:**

```
sage: R.<x, y> = InfinitePolynomialRing(QQ)
sage: R.is_integral_domain()
True
```

**is_noetherian()**

Return False, since polynomial rings in infinitely many variables are never Noetherian rings.
Since Infinite Polynomial Rings must have at least one generator, they have infinitely many variables and are thus not noetherian, as a ring.

**Note:** Infinite Polynomial Rings over a field $F$ are noetherian as $F(G)$ modules, where $G$ is the symmetric group of the natural numbers. But this is not what the method `is_noetherian()` is answering.

### krull_dimension(*args, **kwds)

Return infinity, since polynomial rings in infinitely many variables have infinite Krull dimension.

**EXAMPLES:**

```
sage: R.<x, y> = InfinitePolynomialRing(QQ)
sage: R.krull_dimension()
+Infinity
```

### ngens()

Return the number of generators for this ring.

Since there are countably infinitely many variables in this polynomial ring, by ‘generators’ we mean the number of infinite families of variables. See `infinite_polynomial_ring` for more details.

**EXAMPLES:**

```
sage: X.<x> = InfinitePolynomialRing(ZZ)
sage: X.ngens()
1
sage: X.<x1,x2> = InfinitePolynomialRing(QQ)
sage: X.ngens()
2
```

### 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:**
\textbf{7.2 Elements of Infinite Polynomial Rings}

\textbf{AUTHORS:}

- Simon King <simon.king@nuigalway.ie>
- Mike Hansen <mhansen@gmail.com>

An Infinite Polynomial Ring has generators \(x_*, y_*, \ldots\), so that the variables are of the form \(x_0, x_1, x_2, \ldots, y_0, y_1, y_2, \ldots\). Using the generators, we can create elements as follows:
Any Infinite Polynomial Ring $X$ is equipped with a monomial ordering. We only consider monomial orderings in which:

$$X \text{.gen}(i)[m] > X \text{.gen}(j)[n] \iff i < j \text{ or } i==j \text{ and } m>n$$

Under this restriction, the monomial ordering can be lexicographic (default), degree lexicographic, or degree reverse lexicographic. Here, the ordering is lexicographic, and elements can be compared as usual:

```
sage: X._order
'lex'
sage: a > b
True
```

Note that, when a method is called that is not directly implemented for ‘InfinitePolynomial’, it is tried to call this method for the underlying 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 $c$. 

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:**
Currently, \( P \) and \( X._P \) (the underlying polynomial ring of \( X \)) both have two variables:

```python
sage: X._P
Multivariate Polynomial Ring in alpha_1, alpha_0 over Integer Ring
```

By default, a coercion from \( P \) to \( X._P \) would not be name preserving. However, this is taken care for; a name preserving conversion is impossible, and by consequence an error is raised:

```python
sage: InfinitePolynomial(X, (alpha_1+alpha_2)^2)
Traceback (most recent call last):
...  
TypeError: Could not find a mapping of the passed element to this ring.
```

When extending the underlying polynomial ring, the construction of an infinite polynomial works:

```python
sage: alpha[2]
alpha_2
sage: InfinitePolynomial(X, (alpha_1+alpha_2)^2)
alpha_2^2 + 2*alpha_2*alpha_1 + alpha_1^2
```

In the sparse implementation, it is not checked whether the polynomial really belongs to the parent:

```python
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:

```python
sage: Y(a)
alpha_2^2 + alpha_1^2
```

class sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_dense

Element of a dense Polynomial Ring with a Countably Infinite Number of Variables.

INPUT:

- \( A \) – an Infinite Polynomial Ring in dense implementation
- \( p \) – a \textit{classical} polynomial that can be interpreted in \( A \).

Of course, one should not directly invoke this class, but rather construct elements of \( A \) in the usual way.

This class inherits from \texttt{InfinitePolynomial_sparse}. See there for a description of the methods.

class sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse

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[]
B.<b,c> = InfinitePolynomialRing(A, implementation='sparse')
p = a*b[100] + 1/2*c[4]
p
```

```
a*b_100 + 1/2*c_4
```

```sage
p.parent()
```

```
Infinite polynomial ring in b, c over Univariate Polynomial Ring in a over Rational Field
```

```sage
p.polynomial().parent()
```

```
Multivariate Polynomial Ring in b_100, b_0, c_4, c_0 over Univariate Polynomial Ring in a over Rational Field
```

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)
a.coefficient(x[0])
a.coefficient(x[1])
a.coefficient(x[2])
a.coefficient(x[0]*x[1])
```

```
2*x_1
2*x_0 + 1
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:
```python
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: sorted(p.footprint().items())
[(1, [2, 3]), (30, [1, 0])]
```

**gcd**
computes the greatest common divisor

**EXAMPLES:**

```python
sage: R.<x>=InfinitePolynomialRing(QQ)
sage: p1=x[0]+x[1]**2
sage: gcd(p1,p1+3)
1
sage: gcd(p1,p1)==p1
True
```

**is_nilpotent**
Return True if self is nilpotent, i.e., some power of self is 0.

**EXAMPLES:**

```python
sage: R.<x> = InfinitePolynomialRing(QQbar)
sage: (x[0]+x[1]).is_nilpotent()
False
sage: R(0).is_nilpotent()
True
sage: _.<x> = InfinitePolynomialRing(Zmod(4))
sage: (2*x[0]).is_nilpotent()
True
sage: (2+x[4]*x[7]).is_nilpotent()
False
sage: _.<y> = InfinitePolynomialRing(Zmod(100))
sage: (5+2*y[0] + 10*(y[0]^2+y[1]^2)).is_nilpotent()
False
True
```

**is_unit**
Answer whether self is a unit.

**EXAMPLES:**

```python
sage: R1.<x,y> = InfinitePolynomialRing(ZZ)
sage: R2.<a,b> = InfinitePolynomialRing(QQ)
sage: (1+x[2]).is_unit()
False
sage: R1(1).is_unit()
True
sage: R1(2).is_unit()
False
sage: R2(2).is_unit()
True
sage: (1+a[2]).is_unit()
False
```

Check that trac ticket #22454 is fixed:
The coefficient of the leading term of self.

**EXAMPLES:**

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p.lc()
3
```

The leading monomial of self.

**EXAMPLES:**

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p.lm()
x_10*x_1^2*y_1^3
```

The leading term (= product of coefficient and monomial) of self.

**EXAMPLES:**

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p.lt()
3*x_10*x_1^2*y_1^3
```

Return the maximal index of a variable occurring in self, or -1 if self is scalar.

**EXAMPLES:**

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p.max_index()
4
sage: x[0].max_index()
0
sage: X(10).max_index()
-1
```

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 diminish the variable indices occurring in N and preserves their order, so that there is some term T ∈ X with TNP = M. If there is no such permutation, return p
3. Replace p by p − TqP and continue with step 1.

EXAMPLES:

sage: X.<x,y> = InfinitePolynomialRing(QQ)
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

(continues on next page)
Last, we demonstrate the report option:

```
sage: p.reduce(I, tailreduce=True, report=True)
: T[2]: >
 x_1^2 + y_2^2
```

The output ‘:’ means that there was one reduction of the leading monomial. ‘T[2]’ means that a tail reduction was performed on a polynomial with two terms. At ‘>’, one round of the reduction process is finished (there could only be several non-trivial rounds if \( I \) was generated by more than one polynomial).

**ring()**

The ring which \( self \) belongs to.

This is the same as \( self.parent() \).

**squeeze()**

Reduce the variable indices occurring in \( self \).

**stretch** \((k)\)

Stretch \( self \) by a given factor.

**Examples:**

```
sage: X.<x,y> = InfinitePolynomialRing(ZZ, implementation='sparse')
sage: p.reduce(I, tailreduce=True, report=True)
: T[2]: >
 x_1^2 + y_2^2
```

```
sage: X.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: p.ring()
Infinite polynomial ring in x, y over Integer Ring
```

```
sage: X.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: p = x[100]*y[1]+x[50]*y[1000]
sage: p.squeeze()
x_2*y_4 + x_1*y_3
```

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: a.stretch(2)
x_4 + x_2 + x_0
```

7.2. Elements of Infinite Polynomial Rings
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

**symmetric_cancellation_order**(other)

Comparison of leading terms by Symmetric Cancellation Order, \(<_\text{sc} \>.

**INPUT:**

self, other – two Infinite Polynomials

**ASSUMPTION:**

Both Infinite Polynomials are non-zero.

**OUTPUT:**

\((c, \sigma, w)\), 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 \(self <_\text{sc} other\) (resp. \(self >_\text{sc} other\)) or is 1 if \(self.\lt() == other.\lt()\)
- **w** is 1 or is a term so that \(w*\text{self.}\lt()^\sigma == other.\lt()\) if \(c \leq 0\), and \(w*other.\lt()^\sigma == \text{self.}\lt()\) if \(c = 1\)

**THEORY:**

If the Symmetric Cancellation Order is a well-quasi-ordering then computation of Groebner bases always terminates. This is the case, e.g., if the monomial order is lexicographic. For that reason, lexicographic order is our default order.

**EXAMPLES:**

```python
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:**

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p.tail()
2*x_10+y_30
```

**variables()**

Return the variables occurring in \(self\) (tuple of elements of some polynomial ring).

**EXAMPLES:**
7.3 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: 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., 
\[ x_P^i = x_{P(i)}, \ y_P^n = y_{P(n)}, \ldots \] 
for any permutation \( P \). Note that the variables \( x_0, y_0, \ldots \) of index zero are invariant under that action.

A **Symmetric Ideal** is an ideal in an infinite polynomial ring \( X \) that is invariant under the permutation action. In other words, if \( \mathfrak{S}_\infty \) denotes the symmetric group of \( 1, 2, \ldots \), then a Symmetric Ideal is a right \( X[\mathfrak{S}_\infty] \)-submodule of \( X \).

It is known by work of Aschenbrenner and Hillar [AB2007] that an Infinite Polynomial Ring \( X \) with a single generator \( x_* \) is Noetherian, in the sense that any Symmetric Ideal \( I \subset X \) is finitely generated modulo addition, multiplication by elements of \( X \), and permutation of variable indices (hence, it is a finitely generated right \( X[\mathfrak{S}_\infty] \)-module).

Moreover, if \( X \) is equipped with a lexicographic monomial ordering with \( x_1 < x_2 < x_3 \ldots \) then there is an algorithm of Buchberger type that computes a Groebner basis \( G \) for \( I \) that allows for computation of a unique normal form, that is zero precisely for the elements of \( I \) – see [AB2008]. See \texttt{groebner_basis()} for more details.

Our implementation allows more than one generator and also provides degree lexicographic and degree reverse lexicographic monomial orderings – we do, however, not guarantee termination of the Buchberger algorithm in these cases.

**EXAMPLES:**

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: I == loads(dumps(I))
True
sage: latex(I)
\left(x_{1} y_{2} y_{1} + 2 x_{1} y_{2}\right)\Bold{Q}[/x_{\ast}, y_{\ast} \to \mathfrak{S}_{\infty}]
```

The default ordering is lexicographic. We now compute a Groebner basis:

```python
sage: J = I.groebner_basis() ; J  # about 3 seconds
\[x_1 y_2 y_1 + 2 x_1 y_1, \ x_2 y_2 y_1 + 2 x_2 y_1, \ x_2 x_1 y_1^2 + 2 x_2 x_1 y_1, \ x_2 x_1 y_2^2 + 2 x_2 x_1 y_2\]
```

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 \( J \):

```python
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:

```python
sage: P=Permutation([2, 1])
sage: J[2]
\[x_2 x_1 y_1^2 + 2 x_2 x_1 y_1\]
sage: J[2]^P
\[x_2 x_1 y_2^2 + 2 x_2 x_1 y_2\]
sage: J[2]^P in J
False
```

However, any element of \( J \) has symmetric reduction zero even after applying a permutation. This even holds when the permutations involve higher variable indices than the ones occurring in \( J \):
Since \( I \) is not a Groebner basis, it is no surprise that it can not detect ideal membership:

\[
\text{sage: } \text{[p.reduce(I) for p in J]}
\]

\[
[0, x_2*y_2*y_1 + 2*x_2*y_1, x_2*x_1*y_1^2 + 2*x_2*x_1*y_1, x_2*x_1*y_2 - x_2*x_1*y_1]
\]

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.

\[
\text{sage: } I = X*(x[1])
\]

\[
\text{sage: } I*I
\]

Symmetric Ideal \((x_1^2, x_2*x_1)\) of Infinite polynomial ring in \(x, y\) over \(\text{Rational Field}\)

\[
\text{sage: } I^3
\]

Symmetric Ideal \((x_1^3, x_2*x_1^2, x_2^2*x_1, x_3*x_2*x_1)\) of Infinite polynomial ring in \(x, y\) over \(\text{Rational Field}\)

\[
\text{sage: } I*I == X*(x[1]^2)
\]

\[\text{False}\]

\text{groebner\textunderscore basis}(\text{tailreduce=}False, \text{reduced=}True, \text{algorithm=}None, \text{report=}None, \text{use\textunderscore full\textunderscore group=}False)

Return a symmetric Groebner basis (type Sequence) of self.

INPUT:

- \text{tailreduce} – (bool, default False) If True, use tail reduction in intermediate computations
- \text{reduced} – (bool, default True) If True, return the reduced normalised symmetric Groebner basis.
- \text{algorithm} – (string, default None) Determine the algorithm (see below for available algorithms).
- \text{report} – (object, default None) If not None, print information on the progress of computation.
- \text{use\textunderscore full\textunderscore group} – (bool, default False) If True then proceed as originally suggested by [AB2008]. Our default method should be faster; see \text{symmetrisation()} for more details.

The computation of symmetric Groebner bases also involves the computation of \textit{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:

- \texttt{autoselect} (default)
- \texttt{\texttt{\texttt{singular:groebner}}} Singular’s \texttt{groebner} command
- \texttt{\texttt{\texttt{singular:std}}} Singular’s \texttt{std} command
- \texttt{\texttt{\texttt{singular:stdhilb}}} Singular’s \texttt{stdhilb} command
- \texttt{\texttt{\texttt{singular:stdfglm}}} Singular’s \texttt{stdfglm} command
- \texttt{\texttt{\texttt{libsingular:std}}} \texttt{libSingular’s std} command
- \texttt{\texttt{\texttt{libsingular:slimgb}}} \texttt{libSingular’s slimgb} command
- \texttt{\texttt{\texttt{toy:buchberger}}} Sage’s toy/educational \texttt{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.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
```

(continues on next page)
The Aschenbrenner-Hillar algorithm is only guaranteed to work if the base ring is a field. So, we raise a `TypeError` if this is not the case:

```python
sage: R.<x,y> = InfinitePolynomialRing(ZZ)
sage: I = R*[x[1]+x[2],y[1]]
sage: I.groebner_basis()
Traceback (most recent call last):
  ...TypeError: The base ring (= Integer Ring) must be a field
```

`interreduced_basis()`

A fully symmetrically reduced generating set (type `Sequence`) of self.

This does essentially the same as `interreduction()` with the option `tailreduce`, but it returns a `Sequence` rather than a `SymmetricIdeal`.  

7.3. Symmetric Ideals of Infinite Polynomial Rings
EXAMPLES:

```python
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 increasing sorted.
- `report` – (object, default None) If not None, some information on the progress of computation is printed.
- `RStrat` – (SymmetricReductionStrategy, default None) A reduction strategy to which the polynomials resulting from the interreduction will be added. If RStrat already contains some polynomials, they will be used in the interreduction. The effect is to compute in a quotient ring.

OUTPUT:

A Symmetric Ideal J (sorted list of generators) coinciding with self as an ideal, so that any generator is symmetrically reduced w.r.t. the other generators. Note that the leading coefficients of the result are not necessarily 1.

EXAMPLES:

```python
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I = X*(x[1]+x[2], x[1]*x[2])
sage: I.interreduction()
Symmetric Ideal (-x_1^2, x_2 + x_1) of Infinite polynomial ring in x over Rational Field
```

Here, we show the `report` option:

```python
sage: I.interreduction(report=True)
Symmetric interreduction
[1/2] ->
[2/2] :
[1/2] ->
> Symmetric Ideal (-x_1^2, x_2 + x_1) of Infinite polynomial ring in x over Rational Field
```

[m/n] indicates that polynomial number m is considered and the total number of polynomials under consideration is n. ‘-> 0’ is printed if a zero reduction occurred. The rest of the report is as described in `sage.rings.polynomial.symmetric_reduction.SymmetricReductionStrategy.reduce()`.

Last, we demonstrate the use of the optional parameter `RStrat`: 

```python
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: R = SymmetricReductionStrategy(X)
sage: R
Symmetric Reduction Strategy in Infinite polynomial ring in x over Rational Field
sage: I.interreduction(RStrat=R)
Symmetric Ideal (-x_1^2, x_2 + x_1) of Infinite polynomial ring in x over Rational Field
sage: R
Symmetric Reduction Strategy in Infinite polynomial ring in x over Rational Field, modulo
   x_1^2,
   x_2 + x_1
sage: R = SymmetricReductionStrategy(X, [x[1]^2])
sage: I.interreduction(RStrat=R)
Symmetric Ideal (x_2 + x_1) of Infinite polynomial ring in x over Rational Field
```

### is_maximal()

Answers whether self is a maximal ideal.

**ASSUMPTION:**

`self` is defined by a symmetric Groebner basis.

**NOTE:**

It is not checked whether self is in fact a symmetric Groebner basis. A wrong answer can result if this assumption does not hold. A `NotImplementedError` is raised if the base ring is not a field, since symmetric Groebner bases are not implemented in this setting.

**EXAMPLES:**

```python
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.is_maximal()
False
```

The preceding answer is wrong, since it is not the case that `I` is given by a symmetric Groebner basis:

```python
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:**

7.3. Symmetric Ideals of Infinite Polynomial Rings 577
reduce (I, tailreduce=True)
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 True) If True, the non-leading terms will be reduced as well.

OUTPUT:
Symmetric reduction of self with respect to I.

THEORY:
Reduction of an element \( p \) of an Infinite Polynomial Ring \( X \) by some other element \( q \) means the following:

1. Let \( M \) and \( N \) be the leading terms of \( p \) and \( q \).
2. Test whether there is a permutation \( P \) that does not diminish the variable indices occurring in \( N \) and preserves their order, so that there is some term \( T \in X \) with \( TN^P = M \). If there is no such permutation, return \( p \).
3. Replace \( p \) by \( p - Tq^P \) and continue with step 1.

EXAMPLES:

```sage
sage: X.<x,y> = InfinitePolynomialRing(QQ)
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 preceding is correct, since any permutation that turns \( x[1]^2*y[2] \) into a factor of \( x[3]^2*y[2] \) interchanges the variable indices 1 and 2 – which is not allowed. However, reduction by \( x[2]^2*y[1] \) works, since one can change variable index 1 into 2 and 2 into 3:

```sage
sage: I.reduce([x[2]^2*y[1]])
Symmetric Ideal (y_3*y_1^2) of Infinite polynomial ring in x, y over Rational Field
```

The next example shows that tail reduction is not done, unless it is explicitly advised. The input can also be a symmetric ideal:

```sage
sage: J = (y[2])*X
sage: I.reduce(J)
Symmetric Ideal (x_3^2*y_1 + y_3*y_1^2) of Infinite polynomial ring in x, y over Rational Field
sage: I.reduce(J, tailreduce=True)
Symmetric Ideal (x_3^2*y_1) of Infinite polynomial ring in x, y over Rational Field
```

squeezed()
Reduce the variable indices occurring in self.
OUTPUT:
A Symmetric Ideal whose generators are the result of applying \texttt{squeezed()} to the generators of self.

NOTE:
The output describes the same Symmetric Ideal as self.

EXAMPLES:

\begin{verbatim}
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
\end{verbatim}

\texttt{symmetric\_basis()}
A symmetrised generating set (type \texttt{Sequence}) of self.

This does essentially the same as \texttt{symmetrisation()} with the option `tailreduce`, and it returns a Sequence rather than a \texttt{SymmetricIdeal}.

EXAMPLES:

\begin{verbatim}
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]
\end{verbatim}

\texttt{symmetrisation}(N=None, tailreduce=False, report=None, use\_full\_group=False)
Apply permutations to the generators of self and interreduce

INPUT:

- \texttt{N} – (integer, default None) Apply permutations in $\text{Sym}(N)$. If it is not given then it will be replaced by the maximal variable index occurring in the generators of self.interreduction().squeezed().
- \texttt{tailreduce} – (bool, default False) If True, perform tail reductions.
- \texttt{report} – (object, default None) If not None, report on the progress of computations.
- \texttt{use\_full\_group} (optional) – If True, apply all elements of $\text{Sym}(N)$ to the generators of self (this is what \cite{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 $\text{Sym}(N)$, and we are convinced that both methods yield the same result.

OUTPUT:
A symmetrically interreduced symmetric ideal with respect to which any $\text{Sym}(N)$-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:
7.4 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 diminish the variable indices occurring in \( N \) and preserves their order, so that there is some term \( T \in X \) with \( TNP = M \). If there is no such permutation, return \( p \).
3. Replace \( p \) by \( p - TqP \) and continue with step 1.

When reducing one polynomial \( p \) with respect to a list \( L \) of other polynomials, there usually is a choice of order on which the efficiency crucially depends. Also it helps to modify the polynomials on the list in order to simplify the basic reduction steps.

The preparation of \( L \) may be expensive. Hence, if the same list is used many times then it is reasonable to perform the preparation only once. This is the background of `SymmetricReductionStrategy`.

Our current strategy is to keep the number of terms in the polynomials as small as possible. For this, we sort \( L \) by increasing number of terms. If several elements of \( L \) allow for a reduction of \( p \), we choose the one with the smallest number of terms. Later on, it should be possible to implement further strategies for choice.

When adding a new polynomial \( q \) to \( L \), we first reduce \( q \) with respect to \( L \). Then, we test heuristically whether it is possible to reduce the number of terms of the elements of \( L \) by reduction modulo \( q \). That way, we see best chances to keep the number of terms in intermediate reduction steps relatively small.

EXAMPLES:

First, we create an infinite polynomial ring and one of its elements:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
```

We want to symmetrically reduce it by another polynomial. So, we put this other polynomial into a list and create a Symmetric Reduction Strategy object:

```
sage: 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:

x_3 + 2*x_2*y_1^2 + 3*x_1*y_2^2
x_3

However, it is possible to ask for tailreduction already when the Symmetric Reduction Strategy is created:

sage: 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
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

(continues on next page)
y_3 + 3*y_2^2*y_1 + 2*y_2*y_1^2
y_3

add_generator (p, good_input=None)
Add another polynomial to self.

INPUT:

• p – An element of the underlying infinite polynomial ring.

• good_input – (bool, default None) If True, it is assumed that p is reduced with respect to self. Otherwise, this reduction will be done first (which may cost some time).

Note: Previously added polynomials may be modified. All input is prepared in view of an efficient symmetric reduction.

EXAMPLES:

sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: S = SymmetricReductionStrategy(X)
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo x_3*y_1 + x_1*y_1 + y_3

Note that the first added polynomial will be simplified when adding a suitable second polynomial:

sage: S.add_generator(x[2]+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, 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

(continues on next page)
In the previous example, \( x^3 + x^2 \) is added without being reduced to zero.

**`gens()`**

Return the list of Infinite Polynomials modulo which self reduces.

**EXAMPLES:**

```python
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
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.
- \( \text{notail} \) – (bool, default \( \text{False} \)) If \( \text{True} \), tail reduction is avoided (but there is no guarantee that there will be no tail reduction at all).
- \( \text{report} \) – (object, default \( \text{None} \)) If not \( \text{None} \), print information on the progress of the computation.

**OUTPUT:**

Reduction of \( p \) with respect to \( \text{self} \).

**Note:** If tail reduction shall be forced, use `tailreduce()`.

**EXAMPLES:**

```python
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:
```python
    →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
:::>
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.

```python
reset()
```

Remove all polynomials from self.

**EXAMPLES:**

```python
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field, modulo
    →y_2*y_1^2,
    →y_2^2*y_1
sage: S.reset()
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field
```

```python
setgens(L)
```

Define the list of Infinite Polynomials modulo which self reduces.

**INPUT:**

L – a list of elements of the underlying infinite polynomial ring.

**Note:** It is not tested if L is a good input. That method simply assigns a copy of L to the generators of self.

**EXAMPLES:**

```python
dsage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: R = SymmetricReductionStrategy(X)
sage: R.setgens(S.gens())
sage: R
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field
```

(continues on next page)
sage: R.gens() is S.gens()
False
sage: R.gens() == S.gens()
True

tailreduce \((p, \text{report=\texttt{None}})\)

Symmetric reduction of an infinite polynomial, with forced tail reduction.

**INPUT:**

- \(p\) – an element of the underlying infinite polynomial ring.
- \(\text{report} = (\text{object, default } \texttt{None})\) If not \texttt{None}, print information on the progress of the computation.

**OUTPUT:**

Reduction (including the non-leading elements) of \(p\) with respect to \(\text{self}\).

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<x,y> = InfinitePolynomialRing(QQ)
```

```python
sage: S = SymmetricReductionStrategy(X, [y[3]])
```

```python
sage: S.reduce(y[4]*x[1] + y[1]*x[4])
\texttt{x_4*y_1 + x_1*y_4}
```

```python
sage: S.tailreduce(y[4]*x[1] + y[1]*x[4])
\texttt{x_4*y_1}
```

Last, we demonstrate the ‘report’ option:

```python
```

```python
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over \texttt{Rational Field}, modulo
\texttt{y_3 + y_2, x_2 + y_1, x_1*y_2 + y_4 + y_1^2}
```

```python
T[3]:\>>> T[3]:>
\texttt{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.
- ‘\texttt{:::>’ 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.
8.1 Boolean Polynomials

Elements of the quotient ring

\[ \mathbb{F}_2[x_1, ..., x_n]/ \langle x_1^2 + x_1, ..., x_n^2 + x_n \rangle . \]

are called boolean polynomials. Boolean polynomials arise naturally in cryptography, coding theory, formal logic, chip design and other areas. This implementation is a thin wrapper around the PolyBoRi library by Michael Brickenstein and Alexander Dreyer.

“Boolean polynomials can be modelled in a rather simple way, with both coefficients and degree per variable lying in \{0, 1\}. The ring of Boolean polynomials is, however, not a polynomial ring, but rather the quotient ring of the polynomial ring over the field with two elements modulo the field equations \( x^2 = x \) for each variable \( x \). Therefore, the usual polynomial data structures seem not to be appropriate for fast Groebner basis computations. We introduce a specialised data structure for Boolean polynomials based on zero-suppressed binary decision diagrams (ZDDs), which is capable of handling these polynomials more efficiently with respect to memory consumption and also computational speed. Furthermore, we concentrate on high-level algorithmic aspects, taking into account the new data structures as well as structural properties of Boolean polynomials.” - [BD2007]

For details on the internal representation of polynomials see

http://polybori.sourceforge.net/zdd.html

AUTHORS:

• Michael Brickenstein: PolyBoRi author
• Alexander Dreyer: PolyBoRi author
• Burcin Erocal <burcin@erocal.org>: main Sage wrapper author
• Martin Albrecht <malb@informatik.uni-bremen.de>: some contributions to the Sage wrapper
• Simon King <simon.king@uni-jena.de>: Adopt the new coercion model. Fix conversion from univariate polynomial rings. Pickling of BooleanMonomialMonoid (via UniqueRepresentation) and BooleanMonomial.
• Charles Bouillaguet <charles.bouillaguet@gmail.com>: minor changes to improve compatibility with MPolynomial and make the variety() function work on ideals of BooleanPolynomial’s.

EXAMPLES:

Consider the ideal

\[ \langle ab + cd + 1, ace + de, abc + ce, bc + cde + 1 \rangle . \]
First, we compute the lexicographical Groebner basis in the polynomial ring

\[ R = \mathbb{F}_2[a, b, c, d, e]. \]

```
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 \( \mathbb{F}_2 \) then this Groebner basis was the one to consider. If one wants solutions over \( \mathbb{F}_2 \) only then one adds the field polynomials to the ideal to force the solutions in \( \mathbb{F}_2 \).

```
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 \( \mathbb{F}_2 \) 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 \( \mathbb{F}_2 \) by considering the quotient ring. If \( I \) is an ideal in \( \mathbb{F}[x_1, \ldots, x_n] \) then the ideals in the quotient ring \( \mathbb{F}[x_1, \ldots, x_n]/I \) are in one-to-one correspondence with the ideals of \( \mathbb{F}[x_0, \ldots, x_n] \) containing \( I \) (that is, the ideals \( J \) satisfying \( I \subset J \subset P \)).

```
sage: Q = P.quotient( sage.rings.ideal.FieldIdeal(P) )
sage: I2 = ideal([Q(f) for f in I1.gens()])
sage: for f in I2.groebner_basis():
   ....: f
   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.
8.1.1 Implementation specific notes

PolyBoRi comes with a Python wrapper. However this wrapper does not match Sage’s style and is written using Boost. Thus Sage’s wrapper is a reimplementation of Python bindings to PolyBoRi’s C++ library. This interface is written in Cython like all of Sage’s C/C++ library interfaces. An interface in PolyBoRi style is also provided which is effectively a reimplementation of the official Boost wrapper in Cython. This means that some functionality of the official wrapper might be missing from this wrapper and this wrapper might have bugs not present in the official Python interface.

8.1.2 Access to the original PolyBoRi interface

The re-implementation PolyBoRi’s native wrapper is available to the user too:

```python
sage: from brial import *
sage: declare_ring([Block('x',2),Block('y',3)],globals())
Boolean PolynomialRing in x0, x1, y0, y1, y2
sage: r
Boolean PolynomialRing in x0, x1, y0, y1, y2
sage: [Variable(i, r) for i in range(r.ngens())]
x(0), x(1), y(0), y(1), y(2)
```

For details on this interface see: http://polybori.sourceforge.net/doc/tutorial/tutorial.html.

Also, the interface provides functions for compatibility with Sage accepting convenient Sage data types which are slower than their native PolyBoRi counterparts. For instance, sets of points can be represented as tuples of tuples (Sage) or as BooleSet (PolyBoRi) and naturally the second option is faster.

```python
class sage.rings.polynomial.pbori.BooleConstant
    Bases: object

    Construct a boolean constant (modulo 2) from integer value:

    INPUT:
    • i - an integer

    EXAMPLES:

    sage: from brial import BooleConstant
    sage: [BooleConstant(i) for i in range(5)]
    [0, 1, 0, 1, 0]

    deg()
    Get degree of boolean constant.

    EXAMPLES:

    sage: from brial import BooleConstant
    sage: BooleConstant(0).deg()
    -1
    sage: BooleConstant(1).deg()
    0

    has_constant_part()
    This is true for BooleConstant(1).

    EXAMPLES:
```
sage: from brial import BooleConstant
sage: BooleConstant(1).has_constant_part()
True
sage: BooleConstant(0).has_constant_part()
False

**is_constant()**
This is always true for this case.

**EXAMPLES:**

sage: from brial import BooleConstant
sage: BooleConstant(1).is_constant()
True
sage: BooleConstant(0).is_constant()
True

**is_one()**
Check whether boolean constant is one.

**EXAMPLES:**

sage: from brial import BooleConstant
sage: BooleConstant(0).is_one()
False
sage: BooleConstant(1).is_one()
True

**is_zero()**
Check whether boolean constant is zero.

**EXAMPLES:**

sage: from brial import BooleConstant
sage: BooleConstant(1).is_zero()
False
sage: BooleConstant(0).is_zero()
True

**variables()**
Get variables (return always and empty tuple).

**EXAMPLES:**

sage: from brial import BooleConstant
sage: BooleConstant(0).variables()
()sage: BooleConstant(1).variables()
()

**class sage.rings.polynomial.pbori.BooleSet**

Return a new set of boolean monomials. This data type is also implemented on the top of ZDDs and allows to see polynomials from a different angle. Also, it makes high-level set operations possible, which are in most cases faster than operations handling individual terms, because the complexity of the algorithms depends only on the structure of the diagrams.
Objects of type `BooleanPolynomial` can easily be converted to the type `BooleSet` by using the member function `BooleanPolynomial.set()`.

**INPUT:**

- `param` - either a `CCuddNavigator`, a `BooleSet` or None.
- `ring` - a boolean polynomial ring.

**EXAMPLES:**

```python
sage: from brial 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 brial import *
sage: BooleSet([Monomial(B)])
{{}}
```

Note: `BooleSet` prints as `{}` but are not Python dictionaries.

\[ \text{cartesian_product}\ (rhs) \]

Return the Cartesian product of this set and the set `rhs`.

The Cartesian product of two sets \( X \) and \( Y \) is the set of all possible ordered pairs whose first component is a member of \( X \) and whose second component is a member of \( Y \).

\[
X \times Y = \{(x, y) | x \in X \text{ and } y \in Y \}.
\]

**EXAMPLES:**

```python
sage: B = BooleanPolynomialRing(5,'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2+x2*x3
sage: s = f.set(); s
{{x1,x2}, {x2,x3}}
sage: t = g.set(); t
{{x4}, {}}
sage: s.cartesian_product(t)
{{x1,x2,x4}, {x1,x2}, {x2,x3,x4}, {x2,x3}}
```

\[ \text{change}\ (ind) \]

Swaps the presence of \( x_i \) in each entry of the set.

**EXAMPLES:**

```python
sage: P.<a,b,c> = BooleanPolynomialRing()
sage: f = a+b
sage: s = f.set(); s
{{a}, {b}}
sage: s.change(0)
```

(continues on next page)
diff (rhs)
Return the set theoretic difference of this set and the set rhs.

The difference of two sets $X$ and $Y$ is defined as:

$$X \setminus Y = \{ x \mid x \in X \text{ and } x \notin Y \}.$$ 

EXAMPLES:

```python
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0, x1, x2, x3, x4 = B.gens()
sage: f = x1*x2 + x2*x3
sage: s = f.set(); s
{{x1, x2}, {x2, x3}}
sage: g = x2*x3 + 1
sage: t = g.set(); t
{{x2, x3}, {}}
sage: s.diff(t)
{{x1, x2}}
```

divide (rhs)
Divide each element of this set by the monomial rhs and return a new set containing the result.

EXAMPLES:

```python
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing(order='lex')
sage: f = b*e + b*c*d + b
sage: s = f.set(); s
{{b, c, d}, {b, e}, {b}}
sage: s.divide(b.lm())
{{c, d}, {e}, {}}
sage: f = b*e + b*c*d + b + c
sage: s = f.set()
sage: s.divide(b.lm())
{{c, d}, {e}, {}}
```

divisors_of (m)
Return those members which are divisors of m.

INPUT:

* m - a boolean monomial

EXAMPLES:

```python
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0, x1, x2, x3, x4 = B.gens()
sage: f = x1*x2 + x2*x3
sage: s = f.set()
sage: s.divisors_of((x1*x2*x4).lead())
{{x1, x2}}
```
empty()
Return True if this set is empty.

EXAMPLES:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: BS = (a*b + c).set()
sage: BS.empty()
False
```

```
sage: BS = B(0).set()
sage: BS.empty()
True
```

include_divisors()
Extend this set to include all divisors of the elements already in this set and return the result as a new set.

EXAMPLES:

```
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: f = a*d*e + a*f + a*b*d*e + a*c*d*e + 1
sage: s = f.set(); s
{{a,d,e}, {a,f}, {b,d,e}, {c,d,e}, {}}
sage: s.include_divisors()
{{a,d,e}, {a,d}, {a,e}, {a,f}, {a}, {b,d,e}, {b,d}, {b,e}, {b}, {c,d,e}, {c,d}, {c,e}, {c}, {d,e}, {d}, {e}, {f}, {}}
```

intersect(other)
Return the set theoretic intersection of this set and the set rhs.

The union of two sets $X$ and $Y$ is defined as:

$$X \cap Y = \{x | x \in X \text{ and } x \in Y\}.$$

EXAMPLES:

```
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.

EXAMPLES:

```
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: f = a*d*e + a*f + a*b*d*e + a*c*d*e + a
sage: s = f.set(); s
{{a,b,d,e}, {a,c,d,e}, {a,d,e}, {a,f}, {a}}
sage: s.minimal_elements()
{{a}}
```
multiples_of($m$)
Return those members which are multiples of $m$.

INPUT:

- $m$ - a boolean monomial

EXAMPLES:

```
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0, x1, x2, x3, x4 = B.gens()
sage: f = x1*x2+x2*x3
sage: s = f.set()
sage: s.multiples_of(x1.lm())
{{x1, x2}}
```

n_nodes()
Return the number of nodes in the ZDD.

EXAMPLES:

```
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0, x1, x2, x3, x4 = B.gens()
sage: f = x1*x2+x2*x3
sage: s = f.set(); s
{{x1, x2}, {x2, x3}}
sage: s.n_nodes()
4
```

navigation()
Navigators provide an interface to diagram nodes, accessing their index as well as the corresponding then- and else-branches.

You should be very careful and always keep a reference to the original object, when dealing with navigators, as navigators contain only a raw pointer as data. For the same reason, it is necessary to supply the ring as argument, when constructing a set out of a navigator.

EXAMPLES:

```
sage: from brial 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.

EXAMPLES:

```sage
test = BooleanPolynomialRing(5,'x')
test.gens()
test(f)
```

set()

Return self.

EXAMPLES:

```sage
test.<a,b,c,d> = BooleanPolynomialRing(4)
test.set() == test.set()
```

size_double()

Return the size of this set as a floating point number.

EXAMPLES:

```sage
test = BooleanPolynomialRing(5,'x')
test.gens()
test(f)
test.size_double()
```

stable_hash()

A hash value which is stable across processes.

EXAMPLES:

```sage
test.<x,y> = BooleanPolynomialRing()
test.set() == test.set()
test.stable_hash() == test.stable_hash()
```

Note: This function is part of the upstream PolyBoRi interface. In Sage all hashes are stable.

subset0(i)

Return a set of those elements in this set which do not contain the variable indexed by i.

INPUT:

* i - an index

EXAMPLES:

```sage:BooleanPolynomialRing(5,'x')
```
\texttt{sage}: B.inject_variables()
Defining x0, x1, x2, x3, x4
\texttt{sage}: f = x1*x2+x2*x3
\texttt{sage}: s = f.set(); s
\{\{x1, x2\}, \{x2, x3\}\}
\texttt{sage}: s.subset0(1)
\{\{x2, x3\}\}

\texttt{subset1}(i)
Return a set of those elements in this set which do contain the variable indexed by \texttt{i} and evaluate the variable indexed by \texttt{i} to 1.

\textbf{INPUT:}
\begin{itemize}
  \item \texttt{i} - an index
\end{itemize}

\textbf{EXAMPLES:}
\begin{verbatim}
\texttt{sage}: BooleanPolynomialRing(5, 'x')
Boolean PolynomialRing in x0, x1, x2, x3, x4
\texttt{sage}: B = BooleanPolynomialRing(5, 'x')
\texttt{sage}: B.inject_variables()
Defining x0, x1, x2, x3, x4
\texttt{sage}: f = x1*x2+x2*x3
\texttt{sage}: s = f.set(); s
\{\{x1, x2\}, \{x2, x3\}\}
\texttt{sage}: s.subset1(1)
\{\{x2\}\}
\end{verbatim}

\texttt{union}(\texttt{rhs})
Return the set theoretic union of this set and the set \texttt{rhs}.

The union of two sets \(X\) and \(Y\) is defined as:
\[
X \cup Y = \{x | x \in X \text{ or } x \in Y\}.
\]

\textbf{EXAMPLES:}
\begin{verbatim}
\texttt{sage}: B = BooleanPolynomialRing(5, 'x')
\texttt{sage}: x0, x1, x2, x3, x4 = B.gens()
\texttt{sage}: f = x1*x2+x2*x3
\texttt{sage}: s = f.set(); s
\{\{x1, x2\}, \{x2, x3\}\}
\texttt{sage}: g = x2*x3 + 1
\texttt{sage}: t = g.set(); t
\{\{x2, x3\}, \{\}\}
\texttt{sage}: s.union(t)
\{\{x1, x2\}, \{x2, x3\}, \{\}\}
\end{verbatim}

\texttt{vars}()
Return the variables in this set as a monomial.

\textbf{EXAMPLES:}
\begin{verbatim}
\texttt{sage}: B.<a,b,c,d,e,f> = BooleanPolynomialRing(order='lex')
\texttt{sage}: f = a + b*e + d*f + e + 1
\texttt{sage}: s = f.set()
\texttt{sage}: s
\end{verbatim}
{{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.

class sage.rings.polynomial.pbori.BooleanMonomial
    Bases: sage.structure.element.MonoidElement
    Construct a boolean monomial.
    INPUT:
    • parent - parent monoid this element lives in
    EXAMPLES:

    sage: from brial 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 brial 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 brial import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M(x*y).degree()
2
sage: M(x*y).degree(x)
1
sage: M(x*y).degree(z)
0

**divisors()**

Return a set of boolean monomials with all divisors of this monomial.

**EXAMPLES:**

```python
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y
sage: m = f.lm()
sage: m.divisors()
{{x,y}, {x}, {y}, {}}
```

**gcd(rhs)**

Return the greatest common divisor of this boolean monomial and rhs.

**INPUT:**

- *rhs* - a boolean monomial

**EXAMPLES:**

```python
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.

**EXAMPLES:**

```python
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y
sage: m = f.lm()
sage: m.index()
0
```

**Note:** This function is part of the upstream PolyBoRi interface.

**iterindex()**

Return an iterator over the indices of the variables in self.

**EXAMPLES:**

```python
sage: from brial import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
```
\section{Boolean Polynomials}

\begin{Verbatim}
sage: M = BooleanMonomialMonoid(P)
sage: list(M(x*z).iterindex())
[0, 2]
\end{Verbatim}

**multiples** \((\text{rhs})\)

Return a set of boolean monomials with all multiples of this monomial up to the bound \(\text{rhs}\).

**INPUT:**

- \(\text{rhs}\) - a boolean monomial

**EXAMPLES:**

\begin{Verbatim}
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}}
\end{Verbatim}

**Note:** The returned set always contains \texttt{self} even if the bound \(\text{rhs}\) is smaller than \texttt{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.

**EXAMPLES:**

\begin{Verbatim}
sage: from brial import BooleSet
tsage: B = BooleanPolynomialRing(5,'x')
tsage: x0,x1,x2,x3,x4 = B.gens()
tsage: f = x1*x2+x2*x3*x4+x2*x4+x3+x4+1
tsage: m = f.lm(); m
x1*x2
sage: nav = m.navigation()
sage: BooleSet(nav, B)
{{x1,x2}}
sage: nav.value()
1
\end{Verbatim}

**reducible_by** \((\text{rhs})\)

Return True if \texttt{self} is reducible by \texttt{rhs}.

**INPUT:**

- \(\text{rhs}\) - a boolean monomial

**EXAMPLES:**
ring()  
Return the corresponding boolean ring.

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: a.lm().ring() is B
True
```

set()  
Return a boolean set of variables in this monomials.

```python
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y
sage: m = f.lm()
sage: m.set()
{x, y}
```

stable_hash()  
A hash value which is stable across processes.

```python
sage: B.<x,y> = BooleanPolynomialRing()
sage: x.lm() is x.lm()
False
sage: x.lm().stable_hash() == x.lm().stable_hash()
True
```

Note: This function is part of the upstream PolyBoRi interface. In Sage all hashes are stable.

variables()  
Return a tuple of the variables in this monomial.

```python
sage: from brial 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.
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
def from brial import BooleanMonomialMonoid
def P.<x,y> = BooleanPolynomialRing(2)
def M = BooleanMonomialMonoid(P)
def M
MonomialMonoid of Boolean PolynomialRing in x, y
def M.gens()
(x, y)
def type(M.gen(0))
<type 'sage.rings.polynomial.pbori.BooleanMonomial'>
```

Since [trac ticket #9138](https://trac.sagemath.org/ticket/9138), boolean monomial monoids are unique parents and are fit into the category framework:

```sage
def loads(dumps(M))
def M
def TestSuite(M).run()
```

**gen** (i=0)

Return the i-th generator of self.

INPUT:

• i - an integer

EXAMPLES:

```sage
def from brial import BooleanMonomialMonoid
def P.<x,y,z> = BooleanPolynomialRing(3)
def M = BooleanMonomialMonoid(P)
def M.gen(0)
x
def M.gen(2)
z
def P = BooleanPolynomialRing(1000, 'x')
def M = BooleanMonomialMonoid(P)
def M.gen(50)
x50
```
ngens()  
Return the number of variables in this monoid.

EXAMPLES:

```python
sage: from brial import BooleanMonomialMonoid
sage: P = BooleanPolynomialRing(100, 'x')
sage: M = BooleanMonomialMonoid(P)
sage: M.ngens()
100
```

class sage.rings.polynomial.pbori.BooleanMonomialVariableIterator

Bases: object

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

Note: Do not use this method to construct boolean polynomials, but use the appropriate __call__ method in the parent.

constant()  
Return True if this element is constant.

EXAMPLES:

```python
sage: B.<a,b> = BooleanPolynomialRing()
sage: a.constant()  
False
```

```python
sage: B(1).constant()  
True
```

Note: This function is part of the upstream PolyBoRi interface.

custom_coefficient()  
Return the constant coefficient of this boolean polynomial.

EXAMPLES:

```python
sage: B.<a,b> = BooleanPolynomialRing()
sage: a.constant_coefficient()  
0
```

```python
sage: (a+1).constant_coefficient()  
1
```
**deg**()

Return the degree of self. This is usually equivalent to the total degree except for weighted term orderings which are not implemented yet.

**EXAMPLES:**

```
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.

**EXAMPLES:**

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: x.elength()
1

sage: f = x*y + 1
sage: f.elength()
2
```

**REFERENCES:**

Note: This function is part of the upstream PolyBoRi interface.

first_term()
Return the first term with respect to the lexicographical term ordering.

EXAMPLES:

```
sage: B.<a,b,z> = BooleanPolynomialRing(3,order='lex')
sage: f = b*z + a + 1
sage: f.first_term()
a
```

Note: This function is part of the upstream PolyBoRi interface.

graded_part(deg)
Return graded part of this boolean polynomial of degree deg.

INPUT:
- deg - a degree

EXAMPLES:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + c*d + a*b + 1
sage: f.graded_part(2)
a*b + c*d
sage: f.graded_part(0)
1
```

has_constant_part()
Return True if this boolean polynomial has a constant part, i.e. if 1 is a term.

EXAMPLES:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + c*d + a*b + 1
sage: f.has_constant_part()
True
```

```
sage: f = a*b*c + c*d + a*b
sage: f.has_constant_part()
False
```

is_constant()
Check if self is constant.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P(1).is_constant()
True
```
Sage: P(0).is_constant()
True

Sage: x.is_constant()
False

Sage: (x+y).is_constant()
False

**is_equal** *(right)*

EXAMPLES:

Sage: B.<a,b,z> = BooleanPolynomialRing(3)
Sage: f = a*z + b + 1
Sage: g = b + z
Sage: f.is_equal(g)
False

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

8.1. Boolean Polynomials
**is_pair()**
Check if `self` has exactly two terms.

**EXAMPLES:**

```python
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:**

```python
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
```

Chapter 8. Boolean Polynomials
sage: (x + 1).is_singleton()
False

sage: (x*y + 1).is_singleton()
False

sage: (x + y + 1).is_singleton()
False

sage: ((x + 1)*(y + 1)).is_singleton()
False

is_singleton_or_pair()
Check if self has at most two terms.

EXAMPLES:

sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P(0).is_singleton_or_pair()
True

sage: x.is_singleton_or_pair()
True

sage: P(1).is_singleton_or_pair()
True

sage: (x*y).is_singleton_or_pair()
True

sage: (x + y).is_singleton_or_pair()
True

sage: (x + 1).is_singleton_or_pair()
True

sage: (x*y + 1).is_singleton_or_pair()
True

sage: (x + y + 1).is_singleton_or_pair()
False

sage: ((x + 1)*(y + 1)).is_singleton_or_pair()
False

is_unit()
Check if self is invertible in the parent ring.

Note that this condition is equivalent to being 1 for boolean polynomials.

EXAMPLES:

sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P.one().is_unit()
True

8.1. Boolean Polynomials
```python
sage: x.is_unit()
False
```

**is_univariate()**

Return True if self is a univariate polynomial, that is if self contains only one variable.

**EXAMPLES:**

```python
sage: P.<x,y,z> = 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:**

```python
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P(0).is_zero()
True
sage: x.is_zero()
False
sage: P(1).is_zero()
False
```

**lead()**

Return the leading monomial of boolean polynomial, with respect to the order of parent ring.

**EXAMPLES:**

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x+y+y+z).lead()
x
```

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: (x+y+y+z).lead()
y*z
```

**Note:** This function is part of the upstream PolyBoRi interface.

**lead_deg()**

Return the total degree of the leading monomial of self.

**EXAMPLES:**
```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: p = x + y*z
sage: p.lead_deg()
1
```

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: p = x + y*z
sage: p.lead_deg()
2
```

```python
sage: P(0).lead_deg()
0
```

**Note:** This function is part of the upstream PolyBoRi interface.

---

**lead_divisors()**

Return a BooleSet of all divisors of the leading monomial.

**EXAMPLES:**

```python
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: f = a*b + z + 1
sage: f.lead_divisors()
{{a,b}, {a}, {b}, {}}
```

**Note:** This function is part of the upstream PolyBoRi interface.

---

**lex_lead()**

Return the leading monomial of boolean polynomial, with respect to the lexicographical term ordering.

**EXAMPLES:**

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x+y+y*z).lex_lead()
x
```

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: (x+y+y*z).lex_lead()
x
```

```python
sage: P(0).lex_lead()
0
```

**Note:** This function is part of the upstream PolyBoRi interface.

---

**lex_lead_deg()**

Return degree of leading monomial with respect to the lexicographical ordering.

**EXAMPLES:**

```python
sage: B.<x,y,z> = BooleanPolynomialRing(3, order='lex')
sage: f = x + y*z
```

(continues on next page)
Note: This function is part of the upstream PolyBoRi interface.

\texttt{lm}()  
Return the leading monomial of this boolean polynomial, with respect to the order of parent ring.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x+y+y*z).lm()
x
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: (x+y+y*z).lm()
y*z
sage: P(0).lm()
0
\end{verbatim}

\texttt{lt}()  
Return the leading term of this boolean polynomial, with respect to the order of the parent ring.

Note that for boolean polynomials this is equivalent to returning leading monomials.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x+y+y*z).lt()
x
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: (x+y+y*z).lt()
y*z
\end{verbatim}

\texttt{map\_every\_x\_to\_x\_plus\_one()}  
Map every variable $x_i$ in this polynomial to $x_i + 1$.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: f = a*b + z + 1; f
a*b + z + 1
sage: f.map_every_x_to_x_plus_one()
a*b + a + b + z + 1
\end{verbatim}
monomial_coefficient (mon)

Return the coefficient of the monomial mon in self, where mon must have the same parent as self.

INPUT:

- mon - a monomial

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: x.monomial_coefficient(x)
1
sage: x.monomial_coefficient(y)
0
sage: R.<x,y,z,a,b,c>=BooleanPolynomialRing(6)
sage: f=(1-x)*(1+y); f
x*y + x + y + 1
sage: f.monomial_coefficient(1)
1
sage: f.monomial_coefficient(0)
0
```

monomials ()

Return a list of monomials appearing in self ordered largest to smallest.

EXAMPLES:

```
sage: P.<a,b,c> = BooleanPolynomialRing(3,order='lex')
sage: f = a + c*b
sage: f.monomials()
[a, b*c]
sage: P.<a,b,c> = BooleanPolynomialRing(3,order='deglex')
sage: f = a + c*b
sage: f.monomials()
[b*c, a]
sage: P.zero().monomials()
[]
```

n_nodes ()

Return the number of nodes in the ZDD implementing this polynomial.

EXAMPLES:

```
sage: B = BooleanPolynomialRing(5,'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2 + x2*x3 + 1
sage: f.n_nodes()
4
```

Note: This function is part of the upstream PolyBoRi interface.
n_vars()  
Return the number of variables used to form this boolean polynomial.

EXAMPLES:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + 1
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.

EXAMPLES:

```
sage: from brial 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.

EXAMPLES:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + 1
sage: f.nvariables()
3
```

reduce(I)
Return the normal form of self w.r.t. I, i.e. return the remainder of self with respect to the polynomials
in \( I \). If the polynomial set/list \( I \) is not a Groebner basis the result is not canonical.

**INPUT:**

- \( I \) - a list/set of polynomials in self.parent(). If \( I \) is an ideal, the generators are used.

**EXAMPLES:**

```python
sage: B.<x0,x1,x2,x3> = BooleanPolynomialRing(4)
sage: I = B.ideal((x0 + x1 + x2 + x3,  
....: x0*x1 + x1*x2 + x0*x3 + x2*x3,  
....: x0*x1*x2 + x0*x1*x3 + x0*x2*x3 + x1*x2*x3,  
....: x0*x1*x2*x3 + 1))
sage: gb = I.groebner_basis()
sage: f, g, h, i = I.gens()
sage: f.reduce(gb)
0
sage: p = f*g + x0*h + x2*i
sage: p.reduce(gb)
0
sage: p.reduce(I)
x1*x2*x3 + x2
sage: p.reduce([])
x0*x1*x2 + x0*x1*x3 + x0*x2*x3 + x2
```

**Note:** If this function is called repeatedly with the same \( I \) then it is advised to use PolyBoRi’s `GroebnerStrategy` object directly, since that will be faster. See the source code of this function for details.

**reducible_by** (*rhs*)

Return True if this boolean polynomial is reducible by the polynomial \( rhs \).

**INPUT:**

- \( rhs \) - a boolean polynomial

**EXAMPLES:**

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing(4, order='deglex')
sage: f = (a*b + 1)*(c + 1)
sage: f.reducible_by(d)
False
sage: f.reducible_by(c)
True
sage: f.reducible_by(c + 1)
True
```

**Note:** This function is part of the upstream PolyBoRi interface.

**ring()**

Return the parent of this boolean polynomial.

**EXAMPLES:**

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: a.ring() is B
True
```
set()
Return a BooleSet with all monomials appearing in this polynomial.

EXAMPLES:

```
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: (a+b+z+1).set()
{(a,b), (z), ()}
```

spoly(rhs)
Return the S-Polynomial of this boolean polynomial and the other boolean polynomial rhs.

EXAMPLES:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + c*d + a*b + 1
sage: g = c*d + b
sage: f.spoly(g)
a*b + a*c*d + c*d + 1
```

Note: This function is part of the upstream PolyBoRi interface.

stable_hash()
A hash value which is stable across processes.

EXAMPLES:

```
sage: B.<x,y> = BooleanPolynomialRing()
sage: x is B.gen(0)
False
sage: x.stable_hash() == B.gen(0).stable_hash()
True
```

Note: This function is part of the upstream PolyBoRi interface. In Sage all hashes are stable.

subs(in_dict=None, **kwds)
Fixes some given variables in a given boolean polynomial and returns the changed boolean polynomials. The polynomial itself is not affected. The variable,value pairs for fixing are to be provided as dictionary of the form {variable:value} or named parameters (see examples below).

INPUT:
• in_dict -(optional) dict with variable:value pairs
• **kwds - names parameters

EXAMPLES:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y + z + y*z + 1
sage: f.subs(x=1)
y*z + y + z + 1
sage: f.subs(x=0)
y*z + z + 1
```
This method can work fully symbolic:

```python
sage: f.subs(x=var('a'),y=var('b'),z=var('c'))
a*b + b*c + c + 1
sage: f.subs({'x':var('a'),'y':var('b'),'z':var('c')})
a*b + b*c + c + 1
```

**terms()**

Return a list of monomials appearing in self ordered largest to smallest.

**EXAMPLES:**

```python
sage: P.<a,b,c> = BooleanPolynomialRing(3,order='lex')
sage: f = a + c*b
sage: f.terms()
[a, b*c]
sage: P.<a,b,c> = BooleanPolynomialRing(3,order='deglex')
sage: f = a + c*b
sage: f.terms()
[b*c, a]
```

**total_degree()**

Return the total degree of self.

**EXAMPLES:**

```python
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: (x+y).total_degree()
1
sage: P(1).total_degree()
0
sage: (x*y + x + y + 1).total_degree()
2
```

**univariate_polynomial(R=None)**

Return a univariate polynomial associated to this multivariate polynomial.

If this polynomial is not in at most one variable, then a `ValueError` exception is raised. This is checked using the `is_univariate()` method. The new Polynomial is over GF(2) and in the variable `x` if no ring `R` is provided.
Here's an example with a constant multivariate polynomial:

```python
sage: g = R(1)
sage: h = g.univariate_polynomial(); h
1
sage: h.parent()
Univariate Polynomial Ring in x over Finite Field of size 2 (using GF2X)
```

**variable** (*i=0*)

Return the *i*-th variable occurring in `self`. The index *i* is the index in `self.variables()`.

**EXAMPLES:**

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*z + z + 1
sage: f.variables()
(x, z)
sage: f.variable(1)
z
```

**variables()**

Return a tuple of all variables appearing in `self`.

**EXAMPLES:**

```python
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:**

```python
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
```
Note: This function is part of the upstream PolyBoRi interface.

zeros_in(s)

Return a set containing all elements of s where this boolean polynomial evaluates to zero.

If s is given as a BooleSet, then the return type is also a BooleSet. If s is a set/list/tuple of tuple this function returns a tuple of tuples.

INPUT:

• s - candidate points for evaluation to zero

EXAMPLES:

```sage```
```python
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b + c + d + 1
```

Now we create a set of points:

```sage```
```python
sage: s = a*b + a*b*c + c*d + 1
sage: s = s.set(); s
{{a,b,c}, {a,b}, {c,d}, {}}
```

This encodes the points (1,1,1,0), (1,1,0,0), (0,0,1,1) and (0,0,0,0). But of these only (1,1,0,0) evaluates to zero.

```sage```
```python
sage: f.zeros_in(s)
{{a,b}}
```

```sage```
```python
sage: f.zeros_in([(1,1,1,0), (1,1,0,0), (0,0,1,1), (0,0,0,0)])
((1, 1, 0, 0),)
```

```sage```
```python```

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```
```python
sage: P.<x0, x1, x2, x3> = BooleanPolynomialRing(4)
sage: I = P.ideal(x0*x1*x2*x3 + x0*x1*x3 + x0*x1 + x0*x2 + x0)
sage: I
```

(continues on next page)
Ideal (x0*x1*x2*x3 + x0*x1*x3 + x0*x1 + x0*x2 + x0) of Boolean PolynomialRing in x0, x1, x2, x3

sage: loads(dumps(I)) == I
True

dimension()
Return the dimension of self, which is always zero.

groebner_basis(algorithm='polybori', **kwds)
Return a Groebner basis of this ideal.

INPUT:

- algorithm - either "polybori" (built-in default) or "magma" (requires Magma).
- red_tail - tail reductions in intermediate polynomials, this option affects mainly heuristics. The reducedness of the output polynomials can only be guaranteed by the option redsb (default: True)
- mnsb - return a minimal Groebner basis (default: True)
- redsb - return a minimal Groebner basis and all tails are reduced (default: True)
- deg_bound - only compute Groebner basis up to a given degree bound (default: False)
- faugere - turn off or on the linear algebra (default: False)
- linear_algebra_in_last_block - this affects the last block of block orderings and degree orderings. If it is set to True linear algebra takes effect in this block. (default: True)
- gauss_on_linear - perform Gaussian elimination on linear polynomials (default: True)
- selection_size - maximum number of polynomials for parallel reductions (default: 1000)
- heuristic - Turn off heuristic by setting heuristic=False (default: True)
- lazy - (default: True)
- invert - setting invert=True input and output get a transformation x+1 for each variable x, which 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()
We compute the same example with Magma:

```python
sage: sr = mq.SR(2,1,1,4,gf2=True, polybori=True)
sage: F,s = sr.polynomial_system()
sage: I = F.ideal()
sage: I.groebner_basis(algorithm='magma', prot='sage') # optional - magma
Leading term degree: 3. Critical pairs: 101 (all pairs of current degree eliminated by criteria).
Highest degree reached during computation: 3.
Polynomial Sequence with 35 Polynomials in 36 Variables
```

### interreduced_basis()

If this ideal is spanned by \((f_1, \ldots, f_n)\) this method returns \((g_1, \ldots, g_s)\) such that:

- \(<f_1, \ldots, f_n> = <g_1, \ldots, g_s>
- \(LT(g_i) \neq LT(g_j)\) for all \(i \neq j\)
- \(LT(g_i)\) does not divide \(m\) for all monomials \(m\) of \((g_1, \ldots, g_{i-1}, g_{i+1}, \ldots, g_s)\)

#### EXAMPLES:

```python
sage: sr = mq.SR(1, 1, 1, 4, gf2=True, polybori=True)
sage: F,s = sr.polynomial_system()
sage: I = F.ideal()
sage: I.interreduced_basis()
[k100 + 1, k101 + k001 + 1, k102, k103 + 1, x100 + k001 + 1, x101 + k001, ˓→x102, x103 + k001, w100 + 1, w101 + k001 + 1, w102 + 1, w103 + 1, s000 + ˓→k001, s001 + k001 + 1, s002, s003 + k001 + 1, k000 + 1, k002 + 1, k003 + 1]
```

### reduce(f)

Reduce an element modulo the reduced Groebner basis for this ideal. This returns 0 if and only if the element is in this ideal. In any case, this reduction is unique up to monomial orders.

#### EXAMPLES:

```python
sage: 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
```

8.1. Boolean Polynomials
variety(**kwds)

Return the variety associated to this boolean ideal.

EXAMPLES:

A simple example:

```
sage: from sage.doctest.fixtures import reproducible_repr
sage: R.<x,y,z> = BooleanPolynomialRing()
sage: I = ideal([ x*y*z + x*z + y + 1, x+y+z+1 ])
sage: print(reproducible_repr(I.variety()))
[{'x': 0, 'y': 1, 'z': 0}, {'x': 1, 'y': 1, 'z': 1}]
```

class sage.rings.polynomial.pbori.BooleanPolynomialIterator
Bases: object

Iterator over the monomials of a boolean polynomial.

class sage.rings.polynomial.pbori.BooleanPolynomialRing
Bases: sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base

Construct a boolean polynomial ring with the following parameters:

INPUT:

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

EXAMPLES:

```
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

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

sage: R.term_order()
Degree lexicographic term order
```

c change_ring(base_ring=None, names=None, order=None)

Return a new multivariate polynomial ring with base ring `base_ring`, variable names set to `names`, and term ordering given by `order`.

When `base_ring` is not specified, this function returns a `BooleanPolynomialRing` isomorphic to `self`. Otherwise, this returns an `MPolynomialRing`. Each argument above is optional.
INPUT:

- base_ring – a base ring
- names – variable names
- order – a term order

EXAMPLES:

```sage
code_snippet
```

```plaintext
clone(ordering=None, names=[], blocks=[])
Shallow copy this boolean polynomial ring, but with different ordering, names or blocks if given.
```

ring.clone(ordering=..., names=..., block=...) generates a shallow copy of ring, but with different ordering, names or blocks if given.

EXAMPLES:

```sage
code_snippet
```

```plaintext
8.1. Boolean Polynomials
```
```python
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:

```python
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 $R = \mathbb{F}_2[x_1, x_2, \ldots, x_n]$ if $x_1, x_2, \ldots, x_n$ is the ordered list of variable names of this ring. $R$ also has the same term ordering as this ring.

**EXAMPLES:**

```python
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:

```python
sage: B.cover_ring() is B.cover_ring()
True
```

**defining_ideal()**

Return $I = \langle x_i^2 + x_i \rangle \subseteq R$ where $R = \text{self.cover_ring}()$, and $x_i$ any element in the set of variables of this ring.

**EXAMPLES:**

```python
sage: B.<x,y> = BooleanPolynomialRing(2)
sage: I = B.defining_ideal(); I
Ideal (x^2 + x, y^2 + y) of Multivariate Polynomial Ring in x, y over Finite Field of size 2
```

**gen**(i=0)

Return the i-th generator of this boolean polynomial ring.

**INPUT:**

- i - an integer or a boolean monomial in one variable

**EXAMPLES:**

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.gen()
x
sage: P.gen(2)
z
sage: m = x.monomials()[0]
```

(continues on next page)
sage: P.gen(m)
x

gens()
Return the tuple of variables in this ring.

EXAMPLES:

sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.gens()
(x, y, z)

sage: P = BooleanPolynomialRing(10,'x')
sage: P.gens()
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)

get_base_order_code()
EXAMPLES:

sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: B.get_base_order_code()
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()
EXAMPLES:

sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: B.get_order_code()
0

sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing(order='deglex')
sage: B.get_order_code()
1

Note: This function which is part of the PolyBoRi upstream API works with a current global ring. This notion is avoided in Sage.

has_degree_order()
Return checks whether the order code corresponds to a degree ordering.

EXAMPLES:
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P.has_degree_order()
False

id()
Return a unique identifier for this boolean polynomial ring.

EXAMPLES:

sage: P.<x,y> = BooleanPolynomialRing(2)
sage: print("id: {}".format(P.id()))
id: ...

sage: P = BooleanPolynomialRing(10, 'x')
sage: Q = BooleanPolynomialRing(20, 'x')
sage: P.id() != Q.id()
True

ideal(*gens, **kwds)
Create an ideal in this ring.

INPUT:
- gens - list or tuple of generators
- coerce - bool (default: True) automatically coerce the given polynomials to this ring to form the ideal

EXAMPLES:

sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.ideal(x+y)
Ideal (x + y) of Boolean PolynomialRing in x, y, z

sage: P.ideal(x*y, y*z)
Ideal (x*y, y*z) of Boolean PolynomialRing in x, y, z

sage: P.ideal([x+y, z])
Ideal (x + y, z) of Boolean PolynomialRing in x, y, z

interpolation_polynomial(zeros, ones)
Return the lexicographically minimal boolean polynomial for the given sets of points.

Given two sets of points zeros - evaluating to zero - and ones - evaluating to one -, compute the lexicographically minimal boolean polynomial satisfying these points.

INPUT:
- zeros - the set of interpolation points mapped to zero
- ones - the set of interpolation points mapped to one

EXAMPLES:

First we create a random-ish boolean polynomial.

sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing(6)
sage: f = a*b*c*e + a*d*e + a*f + b + c + e + f + 1
Now we find interpolation points mapping to zero and to one.

```python
sage: zeros = set([(1, 0, 1, 0, 0, 0), (1, 0, 1, 1, 1, 1),
                 (0, 0, 0, 0, 1, 0), (0, 0, 1, 1, 1, 1),
                 (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.

```python
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:

```python
sage: B = BooleanPolynomialRing(4,"x0,x1,x2,x3")
sage: x = B.gen
sage: V=(x(0)+x(1)+x(2)+x(3)+1).set(); V
{{x0}, {x1}, {x2}, {x3}, {}}
sage: f=x(0)*x(1)+x(1)+x(2)+1
sage: z = f.zeros_in(V); z
{{x1}, {x2}}
sage: o = V.diff(z); o
{{x0}, {x3}, {}}
sage: B.interpolation_polynomial(z,o)
x1 + x2 + 1
```

ALGORITHM: Calls interpolate_smallest_lex as described in the PolyBoRi tutorial.

**n_variables()**

Return the number of variables in this boolean polynomial ring.

**EXAMPLES:**

```python
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P.n_variables()
2
sage: P = BooleanPolynomialRing(1000, 'x')
sage: P.n_variables()
1000
```

**Note:** This is part of PolyBoRi’s native interface.
**ngens()**
Return the number of variables in this boolean polynomial ring.

**EXAMPLES:**

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P.ngens()
2

sage: P = BooleanPolynomialRing(1000, 'x')
sage: P.ngens()
1000
```

**one()**

**EXAMPLES:**

```
sage: P.<x0,x1> = BooleanPolynomialRing(2)
sage: P.one()
1
```

**random_element (degree=None, terms=None, choose_degree=False, vars_set=None)**
Return a random boolean polynomial. Generated polynomial has the given number of terms, and at most
given degree.

**INPUT:**

- **degree** - maximum degree (default: 2 for len(var_set) > 1, 1 otherwise)
- **terms** – number of terms requested (default: 5). If more terms are requested than exist, then this parameter is silently reduced to the maximum number of available terms.
- **choose_degree** - choose degree of monomials randomly first, rather than monomials uniformly random
- **vars_set** - list of integer indices of generators of self to use in the generated polynomial

**EXAMPLES:**

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.random_element(degree=3, terms=4)
x*y*z + x*z + x + y*z

sage: P.random_element(degree=1, terms=2)
z + 1
```

In corner cases this function will return fewer terms by default:

```
sage: P = BooleanPolynomialRing(2,'y')
sage: P.random_element()
y0*y1 + y0

sage: P = BooleanPolynomialRing(1,'y')
sage: P.random_element()
y
```

We return uniformly random polynomials up to degree 2:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: B.random_element(terms=Infinity)
a*b + a*c + a*d + b*c + b*d + d
```
remove_var(order=None, *var)
Remove a variable or sequence of variables from this ring.

If order is not specified, then the subring inherits the term order of the original ring, if possible.

EXAMPLES:

```python
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:

```python
sage: R.remove_var(y,z,x,w)
Finite Field of size 2
```

If possible, the term order is kept:

```python
sage: R.<x,y,z,w> = BooleanPolynomialRing(order='deglex')
sage: R.remove_var(y).term_order()
Degree lexicographic term order
```

```python
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:

```python
sage: R.<x,y,z,u,v> = BooleanPolynomialRing(order='deglex(2),deglex(3)')
sage: R.remove_var(x,y,z)
Traceback (most recent call last):
  ... ValueError: impossible to use the original term order (most likely because it was a block order). Please specify the term order for the subring
sage: R.remove_var(x,y,z, order='deglex')
Boolean PolynomialRing in u, v
```

variable(i=0)
Return the i-th generator of this boolean polynomial ring.

INPUT:

- i - an integer or a boolean monomial in one variable

EXAMPLES:

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.variable()
x
sage: P.variable(2)
z
sage: m = x.monomials()[0]
sage: P.variable(m)
x
```

zero()
EXAMPLES:
.. code-block:: python

    sage: P.<x0,x1> = BooleanPolynomialRing(2)
    sage: P.zero()
    0

class sage.rings.polynomial.pbori.BooleanPolynomialVector

    Bases: object

    A vector of boolean polynomials.

    EXAMPLES:

    .. code-block:: python

        sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
        sage: from brial 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 \texttt{el} to this vector.

    EXAMPLES:

    .. code-block:: python

        sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
        sage: from brial 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*d + a*c + a*e + a*f + b*e, b*c + b*d + c*d + c + 1]

class sage.rings.polynomial.pbori.BooleanPolynomialVectorIterator

    Bases: object

class sage.rings.polynomial.pbori.CCuddNavigator

    Bases: object

constant()

derse_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.
EXAMPLES:

```python
sage: from brial 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]
```

```python
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

**EXAMPLES:**

```python
sage: from brial import *
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: gbs = GroebnerStrategy(B)
sage: gbs.add_as_you_wish(a + b)
sage: list(gbs)
[a + b]
sage: gbs.add_as_you_wish(a + c)
```

Note that nothing happened immediately but that the generator was indeed added:

```python
sage: list(gbs)
[a + b]
```

```python
sage: gbs.symmGB_F2()  
sage: list(gbs)
[a + c, b + c]
```

**add_generator**(*p*)
Add a new generator.

**INPUT:**
- *p* - a polynomial

**EXAMPLES:**

```python
sage: from brial 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]
```

(continues on next page)
Let's start by importing the necessary packages:

```python
from brial import *
```

We then define a Boolean Polynomial Ring `B` using variables `a, b, c, d, e, f`:

```python
B.<a,b,c,d,e,f> = BooleanPolynomialRing()
```

Next, we create a `GroebnerStrategy` object `gbs` for the ring `B`:

```python
gbs = GroebnerStrategy(B)
```

We add a generator to `gbs`:

```python
gbs.add_generator(a + b)
```

Then, we assign `gbs` to a variable `sage` and use it to list the current generators:

```python
list(gbs)
```

Now, we add another generator delayed:

```python
gbs.add_generator_delayed(a + c)
```

Again, we list the current generators:

```python
list(gbs)
```

Next, we list all generators:

```python
list(gbs.all_generators())
```

Then, we add another generator as we wish:

```python
gbs.add_as_you_wish(a + c)
```

Again, we list the current generators:

```python
list(gbs)
```

Finally, we list all generators:

```python
list(gbs.all_generators())
```

We can also check if a polynomial contains 1:

```python
contains_one()
```

This function returns `True` if 1 is in the generating system.

We construct an example which contains 1 in the ideal spanned by the generators but not in the set of generators:

```python
B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: from brial import GroebnerStrategy
```

(continues on next page...)

---

**add_generator**

Add a new generator to the strategy.

**add_generator_delayed**

Add a new generator but do not perform interreduction immediately.

**all_generators**

Return all current generators.

**all_spolys_in_next_degree**

Return all spolys in the next degree.

**clean_top_by_chain_criterion**

Clean the top of the ideal using the chain criterion.

**contains_one**

Return `True` if 1 is in the generating system.

**EXAMPLES**

We construct an example which contains 1 in the ideal spanned by the generators but not in the set of generators:
```python
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:

```python
sage: from brial import groebner_basis
groebner_basis(gb)
[1]
```

**faugere_step_dense** *(v)*

Reduces a vector of polynomials using linear algebra.

**INPUT:**
- *v* - a boolean polynomial vector

**EXAMPLES:**

```python
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: from brial 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 brial 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.

---

**8.1. Boolean Polynomials**
minimalize_and_tail_reduce()  
Return a vector of all polynomials with minimal leading terms and do tail reductions.

Note: Use that if strat contains a GB and you want a reduced GB.

next_spoly()  

nf(p)  
Compute the normal form of p with respect to the generating set.

INPUT:  
• p - a boolean polynomial

EXAMPLES:

```python
sage: P = PolynomialRing(GF(2),10, 'x')
sage: B = BooleanPolynomialRing(10,'x')
sage: I = sage.rings.ideal.Cyclic(P)
sage: I = B.ideal([B(f) for f in I.gens()])
sage: gb = I.groebner_basis()
sage: from brial import GroebnerStrategy
sage: strat = GroebnerStrategy(B)
sage: strat.add_generator(f)
sage: strat.add_generator(g)
sage: strat.select(f.lm())
```  

Note: The result is only canonical if the generating set is a Groebner basis.

npairs()  

reduction_strategy  
select(m)  
Return the index of the generator which can reduce the monomial m.

INPUT:  
• m - a BooleanMonomial

EXAMPLES:

```python
sage: B.<a,b,c,d,e> = BooleanPolynomialRing()
sage: f = B.random_element()
sage: g = B.random_element()
sage: from brial import GroebnerStrategy
sage: strat = GroebnerStrategy(B)
sage: strat.add_generator(f)
sage: strat.add_generator(g)
sage: strat.select(f.lm())
```  

(continues on next page)
small_spols_in_next_degree\((f,n)\)

some_spols_in_next_degree\((n)\)

suggest_plugin_variable()

\textbf{symmGB\_P2()}

Compute a Groebner basis for the generating system.

\textbf{Note:} This implementation is out of date, but it will revived at some point in time. Use the \texttt{groebner\_basis()} function instead.

top_sugar()

\textbf{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.

\textbf{INPUT:}

\begin{itemize}
  \item \(v\) - the index of a variable
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing() 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*b + b + c*e + e + 1) sage: gb.variable_has_value(0) False sage: g = gb.groebner_basis() sage: list(g) [a, b + 1, c + 1, d, e + 1, f]
\end{verbatim}

\begin{verbatim}
sage: gb = GroebnerStrategy(B) sage: _ = [gb.add_generator(f) for f in g] sage: gb.variable_has_value(0) True
\end{verbatim}

\class{\texttt{sage.rings.polynomial.pbori.MonomialConstruct}}

Bases: \texttt{object}

\textbf{Implements} PolyBoRi’s \texttt{Monomial()} constructor.

\class{\texttt{sage.rings.polynomial.pbori.MonomialFactory}}

Bases: \texttt{object}

\textbf{Implements} PolyBoRi’s \texttt{Monomial()} constructor. If a ring is given is can be used as a Monomial factory for the given ring.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from brial import * sage: B.<a,b,c> = BooleanPolynomialRing() sage: fac = MonomialFactory() sage: fac = MonomialFactory(B)
\end{verbatim}
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 the order of parent ring.

    EXAMPLES:

    sage: from brial 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 the order of parent ring.

    EXAMPLES:

    sage: from brial 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.

    EXAMPLES:

    sage: from brial import *
    sage: B.<x,y,z> = BooleanPolynomialRing()
    sage: red = ReductionStrategy(B)
    sage: red.add_generator(x)
    sage: list([f.p for f in red])
    [x]

    can_rewrite(p)

    Return True if p can be reduced by the generators of this strategy.

    EXAMPLES:

    sage: from brial 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)
    (continues on next page)
True
sage: red.can_rewrite(b + c)
False
sage: red.can_rewrite(a*d + b*c + d + 1)
True

cheap_reductions(p)
Perform ‘cheap’ reductions on p.

INPUT:
  • p - a boolean polynomial

EXAMPLES:

sage: from brial 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

EXAMPLES:

sage: from brial import *
           
sage: B.<x,y,z> = BooleanPolynomialRing()
           
sage: red = ReductionStrategy(B)
           
sage: red.opt_red_tail = True
           
sage: red.add_generator(x + y + 1)
           
sage: red.add_generator(y*z + z)
           
sage: red.head_normal_form(x + y*z)
y + z + 1
           
sage: red.nf(x + y*z)
y + z + 1

nf(p)
Compute the normal form of p w.r.t. to the generators of this reduction strategy object.

EXAMPLES:
\texttt{sage}: \texttt{red.add_generator(x + y + 1)}
\texttt{sage}: \texttt{red.add_generator(y*z + z)}
\texttt{sage}: \texttt{red.nf(x)}
\texttt{y + 1}
\texttt{sage}: \texttt{red.nf(y*z + x)}
\texttt{y + z + 1}

\textbf{reduced\_normal\_form}(p)

Compute the normal form of \( p \) with respect to the generators of this strategy and perform tail reductions.

\textbf{INPUT}:

- \( p \) - a polynomial

\textbf{EXAMPLES}:

\begin{verbatim}
\texttt{sage: from brial import *}
\texttt{sage: B.<x,y,z> = BooleanPolynomialRing()}
\texttt{sage: red = ReductionStrategy(B)}
\texttt{sage: red.add_generator(x + y + 1)}
\texttt{sage: red.add_generator(y*z + z)}
\texttt{sage: red.reduced_normal_form(x)}
\texttt{y + 1}
\texttt{sage: red.reduced_normal_form(y*z + x)}
\texttt{y + z + 1}
\end{verbatim}

\texttt{sage\_rings\_polynomial\_pbori.TermOrder\_from\_pb\_order}(n, order, blocks)

class \texttt{sage\_rings\_polynomial\_pbori.VariableBlock}

Bases: object

class \texttt{sage\_rings\_polynomial\_pbori.VariableConstruct}

Bases: object

Implements \texttt{PolyBoRi's Variable()} constructor.

class \texttt{sage\_rings\_polynomial\_pbori.VariableFactory}

Bases: object

Implements \texttt{PolyBoRi's Variable()} constructor and a variable factory for given ring

\texttt{sage\_rings\_polynomial\_pbori.add\_up\_polynomials}(v, \textit{init})

Add up all entries in the vector \( v \).

\textbf{INPUT}:

- \( v \) - a vector of boolean polynomials

\textbf{EXAMPLES}:

\begin{verbatim}
\texttt{sage: from brial import *}
\texttt{sage: B.<a,b,c,d> = BooleanPolynomialRing()}
\texttt{sage: v = BooleanPolynomialVector()}
\texttt{sage: l = [B.random_element() for _ in range(5)]}
\texttt{sage: _ = [v.append(e) for e in l]}
\texttt{sage: add\_up\_polynomials(v, B.zero())}
\texttt{a*d + b*c + b*d + c + 1}
\texttt{sage: sum(l)}
\texttt{a*d + b*c + b*d + c + 1}
\end{verbatim}
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

EXAMPLES:

```python
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: from brial 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 1 0 0 0 1 0 0 0 1 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0]
[0 1 1 1 0 0 0 0 1 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 0 0]
[0 1 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0]
sage: e = gauss_on_polys(l)
sage: E,v = Sequence(e,B).coefficient_matrix()
sage: E
[1 0 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 1 1 1 0 1 0 1 0]
[0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 1 1 0]
[0 0 1 0 0 0 0 1 1 0 1 1 0 1 1 0 1 1]
[0 0 0 1 0 0 1 0 1 1 0 1 0 0 1 0 1 0]
[0 0 0 0 0 0 1 1 0 1 0 1 0 0 0 0 0 0]
sage: A.echelon_form()
[1 0 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 1 1 1 0 1 0 1 0]
[0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 1 1 0]
[0 0 1 0 0 0 0 1 1 0 1 1 0 1 1 0 1 1]
[0 0 0 1 0 0 1 0 1 1 0 1 0 0 1 0 1 0]
[0 0 0 0 0 0 1 1 0 1 0 1 0 0 0 0 0 0]
```

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:

```python
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)
```

(continues on next page)
The opposite of navigating down a ZDD using navigators is to construct new ZDDs in the same way, namely giving their else- and then-branch as well as the index value of the new node.

**INPUT:**

- `root` - a variable
- `a` - the if branch, a `BooleSet` or a `BoolePolynomial`
- `b` - the else branch, a `BooleSet` or a `BoolePolynomial`

**EXAMPLES:**

```python
sage: from brial import if_then_else
sage: B = BooleanPolynomialRing(6, 'x')
sage: x0, x1, x2, x3, x4, x5 = B.gens()
sage: f0 = x2*x3 + x3
sage: f1 = x4
sage: if_then_else(x1, f0, f1)

{{x1, x2, x3}, {x1, x3}, {x4}}
```

```python
sage: if_then_else(x1.lm().index(), f0, f1)

{{x1, x2, x3}, {x1, x3}, {x4}}
```

```python
sage: if_then_else(x5, f0, f1)

Traceback (most recent call last):
...
IndexError: index of root must be less than the values of roots of the branches.
```

Interpolate a polynomial evaluating to zero on `zero` and to one on `ones`.

**INPUT:**

- `zero` - the set of zero
- `one` - the set of ones

**EXAMPLES:**

```python
sage: B = BooleanPolynomialRing(4, "x0,x1,x2,x3")
sage: x = B.gen
sage: from brial.interpolate import *
sage: V = (x(0) + x(1) + x(2) + x(3) + 1).set()

sage: V

{{x0}, {x1}, {x2}, {x3}, {}}
```

```python
sage: f = x(0)*x(1) + x(1) + x(2) + 1
sage: nf_lex_points(f, V)
x1 + x2 + 1
```

```python
sage: z = f.zeros_in(V)
```

(continues on next page)
sage.rings.polynomial.pbori.interpolate_smallest_lex(zero, one)
Interpolate the lexicographical smallest polynomial evaluating to zero on \texttt{zero} and to one on \texttt{ones}.

INPUT:
- \texttt{zero} - the set of zeros
- \texttt{one} - the set of ones

EXAMPLES:
Let V be a set of points in $\mathbb{F}_2^4$ and f a Boolean polynomial. V can be encoded as a BooleSet. Then we are interested in the normal form of f against the vanishing ideal of V: $I(V)$.

It turns out, that the computation of the normal form can be done by the computation of a minimal interpolation polynomial, which takes the same values as f on V:

```python
sage: B = BooleanPolynomialRing(4,"x0,x1,x2,x3")
sage: x = B.gen
sage: from brial.interpolate import *
sage: V=(x(0)+x(1)+x(2)+x(3)+1).set()
```

We take $V = \{e_0,e_1,e_2,e_3,0\}$, where $e_i$ describes the i-th unit vector. For our considerations it does not play any role, if we suppose V to be embedded in $\mathbb{F}_4^2$ or a vector space of higher dimension:

```python
sage: V
{{x0}, {x1}, {x2}, {x3}, {}}
sage: f=x(0)*x(1)+x(1)+x(2)+1
sage: nf_lex_points(f,V)
x1 + x2 + 1
```

In this case, the normal form of f w.r.t. the vanishing ideal of V consists of all terms of f with degree smaller or equal to 1.

It can be easily seen, that this polynomial forms the same function on V as f. In fact, our computation is equivalent to the direct call of the interpolation function \texttt{interpolate_smallest_lex}, which has two arguments: the set of interpolation points mapped to zero and the set of interpolation points mapped to one:

```python
sage: z=f.zeros_in(V)
sage: z
{{x1}, {x2}}
sage: o=V.diff(z)
sage: o
{{x0}, {x3}, {}}
sage: interpolate_smallest_lex(z,o)
x1 + x2 + 1
```
sage.rings.polynomial.pbori.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.

**EXAMPLES:**

```python
sage: from brial import ll_red_nf_noredsb
sage: B.<a,b,c,d> = BooleanPolynomialRing()
```

```python
sage: p = a*b + c + d + 1
sage: f, g = a + c + 1, b + d + 1
sage: reductors = f.set().union( g.set() )
```

```python
sage: ll_red_nf_noredsb(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.

**EXAMPLES:**

```python
sage: from brial import ll_red_nf_noredsb_single_recursive_call
```

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing()
```

```python
sage: p = a*b + c + d + 1
sage: f, g = a + c + 1, b + d + 1
sage: reductors = f.set().union( g.set() )
```

```python
sage: ll_red_nf_noredsb_single_recursive_call(p, reductors)
b*c + b*d + c + d + 1
```

sage.rings.polynomial.pbori.ll_red_nf_redsb \((p, reductors)\)

Redude the polynomial \(p\) by the set of \(reductors\) with linear leading terms. It is assumed that the set \(reductors\) is a reduced Groebner basis.

**INPUT:**

- \(p\) - a boolean polynomial
- \(reductors\) - a boolean set encoding a reduced Groebner basis with linear leading terms.

**EXAMPLES:**

```python
sage: from brial import ll_red_nf_redsb
```

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing()
```

```python
sage: p = a*b + c + d + 1
sage: f, g = a + c + 1, b + d + 1
sage: reductors = f.set().union( g.set() )
```

```python
sage: ll_red_nf_redsb(p, reductors)
b*c + b*d + c + d + 1
```
Map every variable $x_i$ in this polynomial to $x_i + 1$.

**EXAMPLES:**

```
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: f = a*b + z + 1; f
a*b + z + 1
sage: from brial 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
```

Return a random set of monomials with length elements with each element in the variables variables.

**EXAMPLES:**

```
sage: from brial 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}}
```

Perform tail reduction on $p$ using the generators of $s$.

**INPUT:**

- $s$ - a reduction strategy
- $p$ - a polynomial

**EXAMPLES:**

```
sage: from brial import *
sage: B.<x,y,z> = BooleanPolynomialRing()
sage: red = ReductionStrategy(B)
sage: red.add_generator(x + y + 1)
sage: red.add_generator(y*x + z)
sage: red_tail(red,x)
x
sage: red_tail(red,x*y + x)
x*y + y + 1
```

The the PolyBoRi random seed to seed
EXAMPLES:

```
sage: from brial 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.
```
```
EXAMPLES:
```
sage: B.<a,b,c> = BooleanPolynomialRing()
```
```
sage: f = a*b + c + 1
```
```
sage: from brial 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 brial 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
```
```
EXAMPLES:
```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
```
```
sage: from brial import top_index
```
```
sage: top_index(x.lm())
```
```
0
```
```
sage: top_index(y*z)
```
```
(continues on next page)
sage.rings.polynomial.pbori.unpickle_BooleanPolynomial \((ring, string)\)
Unpickle boolean polynomials

EXAMPLES:

```python
sage: T = TermOrder('deglex',2)+TermOrder('deglex',2)
sage: P.<a,b,c,d> = BooleanPolynomialRing(4, order=T)
sage: loads(dumps(a+b)) == a+b  # indirect doctest
True
```

sage.rings.polynomial.pbori.unpickle_BooleanPolynomial0 \((ring, l)\)
Unpickle boolean polynomials

EXAMPLES:

```python
sage: T = TermOrder('deglex',2)+TermOrder('deglex',2)
sage: P.<a,b,c,d> = BooleanPolynomialRing(4, order=T)
sage: loads(dumps(a+b)) == a+b  # indirect doctest
True
```

sage.rings.polynomial.pbori.unpickle_BooleanPolynomialRing \((n, names, order)\)
Unpickle boolean polynomial rings.

EXAMPLES:

```python
sage: T = TermOrder('deglex',2)+TermOrder('deglex',2)
```

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

EXAMPLES:

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b + a*c + d + b
```

Now we create a set of points:

```python
sage: s = a*b + a*b*c + c*d + b*c
sage: s = s.set(); s
{(a,b,c), (a,b), (b,c), (c,d)}
```

This encodes the points \((1,1,0,0), (1,1,0,0), (0,0,1,1)\) and \((0,1,1,0)\). But of these only \((1,1,0,0)\) evaluates to zero.:

```python
sage: from brial import zeros
sage: zeros(f,s)
{(a,b)}
```
For comparison we work with tuples:

```python
sage: f.zeros_in([(1,1,1,0), (1,1,0,0), (0,0,1,1), (0,1,1,0)])
((1, 1, 0, 0),)
```
9.1 Noncommutative Polynomials via libSINGULAR/Plural

This module provides specialized and optimized implementations for noncommutative multivariate polynomials over many coefficient rings, via the shared library interface to SINGULAR. In particular, the following coefficient rings are supported by this implementation:

- the rational numbers \(\mathbb{Q}\) and
- finite fields \(\mathbb{F}_p\) for \(p\) prime

**AUTHORS:**

The PLURAL wrapper is due to

- Burcin Erocal (2008-11 and 2010-07): initial implementation and concept
- Michael Brickenstein (2008-11 and 2010-07): initial implementation and concept
- Oleksandr Motsak (2010-07): complete overall noncommutative functionality and first release
- Alexander Dreyer (2010-07): noncommutative ring functionality and documentation
- Simon King (2011-09): left and two-sided ideals; normal forms; pickling; documentation

The underlying libSINGULAR interface was implemented by

- Martin Albrecht (2007-01): initial implementation
- Joel Mohler (2008-01): misc improvements, polishing
- Martin Albrecht (2008-08): added \(\mathbb{Q}(a)\) and \(\mathbb{Z}\) support
- Simon King (2009-04): improved coercion
- Martin Albrecht (2009-05): added \(\mathbb{Z}/n\mathbb{Z}\) support, refactoring
- Martin Albrecht (2009-06): refactored the code to allow better re-use

**Todo:** extend functionality towards those of libSINGULARs commutative part

**EXAMPLES:**

We show how to construct various noncommutative polynomial rings:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P.<x,y,z> = A.g_algebra(relations=(y*x:-x*y), order = 'lex')
```

(continues on next page)
sage: P
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-
relations: {y*x: -x*y}

sage: y*x + 1/2
-x*y + 1/2

sage: A.<x,y,z> = FreeAlgebra(GF(17), 3)
sage: P.<x,y,z> = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
sage: P
Noncommutative Multivariate Polynomial Ring in x, y, z over Finite Field of size 17,
relations: {y*x: -x*y}

sage: y*x + 7
-x*y + 7

Raw use of this class; this is not the intended use!

sage: from sage.matrix.constructor import Matrix
sage: c = Matrix(3)
sage: c[0,1] = -2
sage: c[0,2] = 1
sage: c[1,2] = 1

sage: d = Matrix(3)
sage: d[0, 1] = 17
sage: P = QQ['x','y','z']
sage: c = c.change_ring(P)
sage: d = d.change_ring(P)

sage: from sage.rings.polynomial.plural import NCPolynomialRing_plural
sage: R.<x,y,z> = NCPolynomialRing_plural(QQ, c = c, d = d, order=TermOrder('lex',3),
category=Algebras(QQ))
sage: R
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-
relations: {y*x: -2*x*y + 17}

sage: R.term_order()
Lexicographic term order

sage: a,b,c = R.gens()
sage: f = 57 * a^2*b + 43 * c + 1; f
57*x^2*y + 43*z + 1

sage.rings.polynomial.plural.ExteriorAlgebra(base_ring, names, order='degrevlex')

Return the exterior algebra on some generators

This is also known as a Grassmann algebra. This is a finite dimensional algebra, where all generators anti-
commute.

See Wikipedia article Exterior algebra

INPUT:

• base_ring – the ground ring

• names – a list of variable names

EXAMPLES:
```python
from sage.rings.polynomial.plural import ExteriorAlgebra
E = ExteriorAlgebra(QQ, ['x', 'y', 'z']) ; E #random
Quotient of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {z*x: -x*z, z*y: -y*z, y*x: -x*y} by the ideal (z^2, y^2, x^2)
sage: sorted(E.cover().domain().relations().items(), key=str)
[(y*x, -x*y), (z*x, -x*z), (z*y, -y*z)]
sage: sorted(E.cover().kernel().gens(), key=str)
[x^2, y^2, z^2]
sage: E.inject_variables()
Defining xbar, ybar, zbar
sage: x,y,z = (xbar,ybar,zbar)
sage: y*x
-x*y
sage: all(v^2==0 for v in E.gens())
True
sage: E.one()
1
```

class sage.rings.polynomial.plural.ExteriorAlgebra_plural

Bases: sage.rings.polynomial.plural.NCPolynomialRing_plural
class sage.rings.polynomial.plural.G_AlgFactory

Bases: sage.structure.factory.UniqueFactory
A factory for the creation of g-algebras as unique parents.
create_key_and_extra_args(base_ring, c, d, names=None, order=None, category=None, check=None)
Create a unique key for g-algebras.
INPUT:
- base_ring - a ring
- c, d - two matrices
- names - a tuple or list of names
- order - (optional) term order
- category - (optional) category
- check - optional bool
create_object(version, key, **extra_args)
Create a g-algebra to a given unique key.
INPUT:
- key - a 6-tuple, formed by a base ring, a tuple of names, two matrices over a polynomial ring over the base ring with the given variable names, a term order, and a category
- extra_args - a dictionary, whose only relevant key is ‘check’.

class sage.rings.polynomial.plural.NCPolynomialRing_plural

Bases: sage.rings.ring.Ring
A non-commutative polynomial ring.
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._is_category_initialized()
True
sage: H.category()
Category of algebras over Rational Field
sage: TestSuite(H).run()

Note that two variables commute if they are not part of the given relations:

sage: H.<x,y,z> = A.g_algebra({z*x:x*z+2*x, z*y:y*z-2*y})
sage: x*y == y*x
True

free_algebra()

The free algebra of which this is the quotient.

EXAMPLES:

sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
sage: B = P.free_algebra()
sage: A == B
True

gen(n=0)

Returns the n-th generator of this noncommutative polynomial ring.

INPUT:

- n – an integer >= 0

EXAMPLES:

sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
sage: P.gen(),P.gen(1)
(x, y)

Note that the generators are not cached:

sage: P.gen(1) is P.gen(1)
False

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.
- side - string (either “left”, which is the default, or “twosided”) Must be a keyword argument. Defines whether the ideal is a left ideal or a two-sided ideal. Right ideals are not implemented.

EXAMPLES:

```sage
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P.<x,y,z> = A.g_algebra(relations={y*x:-x*y}, order='lex')

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])
Left Ideal (x + 2*y + 2*z - 1, 2*x*y + 2*y*z - y, x^2 + 2*y^2 + 2*z^2 - x) of
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {y*x: -x*y}

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], side='twosided')
Twosided Ideal (x + 2*y + 2*z - 1, 2*x*y + 2*y*z - y, x^2 + 2*y^2 + 2*z^2 - x) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {y*x: -x*y}
```

\textbf{is_commutative}()

Return False.

\textbf{Todo:} Provide a mathematically correct answer.

\textbf{EXAMPLES:}

```sage
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order='lex')
sage: P.is_commutative()
False
```

\textbf{is_field}(*\texttt{args}, **\texttt{kwargs})

Return False.

\textbf{EXAMPLES:}

```sage
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order='lex')
sage: P.is_field()
False
```

\textbf{monomial_all_divisors}(t)

Return a list of all monomials that divide t.

Coefficients are ignored.

\textbf{INPUT:}

- \texttt{t} - a monomial

\textbf{OUTPUT:}

a list of monomials

\textbf{EXAMPLES:}

```sage
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order='lex')
sage: P.inject_variables()
Defining x, y, z
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]
```

\textbf{ALGORITHM:} addwithcarry idea by Toon Segers

9.1. Noncommutative Polynomials via libSINGULAR/Plural

649
**monomial_divides** \((a, b)\)

Return `False` if \(a\) does not divide \(b\) and `True` otherwise. Coefficients are ignored.

**INPUT:**
- \(a\) – monomial
- \(b\) – monomial

**EXAMPLES:**

```python
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order='lex')
sage: P.inject_variables()
Defining x, y, z
sage: P.monomial_divides(x*y*z, x^3*y^2*z^4)
True
sage: P.monomial_divides(x^3*y^2*z^4, x*y*z)
False
```

**monomial_lcm** \((f, g)\)

LCM for monomials. Coefficients are ignored.

**INPUT:**
- \(f\) – monomial
- \(g\) – monomial

**EXAMPLES:**

```python
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order='lex')
sage: P.inject_variables()
Defining x, y, z
sage: P.monomial_lcm(3/2*x*y,x)
x*y
```

**monomial_pairwise_prime** \((g, h)\)

Return `True` if \(h\) and \(g\) are pairwise prime. Both \(h\) and \(g\) are treated as monomials.

Coefficients are ignored.

**INPUT:**
- \(h\) – monomial
- \(g\) – monomial

**EXAMPLES:**

```python
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order='lex')
sage: P.inject_variables()
Defining x, y, z
sage: P.monomial_pairwise_prime(x^2*z^3, y^4)
```

(continues on next page)
**monomial_pairwise_prime** *(f, g)*

Return True if neither f nor g is divisible by the other.

**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: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order='lex')
sage: P.inject_variables()
Defining x, y, z
sage: P.monomial_quotient(3/2*x*y,x,coeff=True)
3/2*y
```

Note that **Z** behaves differently if coeff=True:

```
sage: P.monomial_quotient(2*x,3*x)
1
sage: P.monomial_quotient(2*x,3*x,coeff=True)
2/3
```

**Warning:** Assumes that the head term of \( f \) is a multiple of the head term of \( g \) and return the multiplier \( m \). If this rule is violated, funny things may happen.

**monomial_reduce** *(f, G)*

Try to find a \( g \) in \( G \) where \( g\cdot\text{lm}(f) \) divides \( f \). If found \((\text{flt}, g)\) is returned, \((0, 0)\) otherwise, where \( \text{flt} = f/g\cdot\text{lm}(f) \).

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: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order='lex')
sage: P.inject_variables()
Defining x, y, z
sage: P.monomial_reduce(2*x,3*x)
(0, 0)
sage: P.monomial_reduce(2*x,3*x,coeff=True)
(1/3, 2)
```
sage: f = x*y^2
sage: G = [ 3/2*x^3 + y^2 + 1/2, 1/4*x*y + 2/7, 1/2 ]
sage: P.monomial_reduce(f,G)
(y, 1/4*x*y + 2/7)

ngens()
Returns the number of variables in this noncommutative polynomial ring.

EXAMPLES:

sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P.<x,y,z> = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
sage: P.ngens()
3

relations(add_commutative=False)
Return the relations of this g-algebra.

INPUT:
add_commutative (optional bool, default False)

OUTPUT:
The defining relations. There are some implicit relations: Two generators commute if they are not part of any given relation. The implicit relations are not provided, unless add_commutative==True.

EXAMPLES:

sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({z*x:x*z+2*x, z*y:y*z-2*y})
sage: x*y == y*x
True
sage: H.relations()
{x*z: x*z + 2*x, z*y: y*z - 2*y}
sage: H.relations(add_commutative=True)
{y*x: x*y, z*x: x*z + 2*x, z*y: y*z - 2*y}

term_order()
Return the term ordering of the noncommutative ring.

EXAMPLES:

sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
sage: P.term_order()
Lexicographic term order

sage: P = A.g_algebra(relations={y*x:-x*y})
sage: P.term_order()
Degree reverse lexicographic term order

class sage.rings.polynomial.plural.NCPolynomial_plural
Bases: sage.structure.element.RingElement

A noncommutative multivariate polynomial implemented using libSINGULAR.

coefficient(degrees)
Return the coefficient of the variables with the degrees specified in the python dictionary degrees.
Mathematically, this is the coefficient in the base ring adjoined by the variables of this ring not listed in degrees. However, the result has the same parent as this polynomial.

This function contrasts with the function `monomial_coefficient()` which returns the coefficient in the base ring of a monomial.

INPUT:

- **degrees** - Can be any of:
  - a dictionary of degree restrictions
  - a list of degree restrictions (with None in the unrestricted variables)
  - a monomial (very fast, but not as flexible)

OUTPUT:

element of the parent of this element.

**Note:** For coefficients of specific monomials, look at `monomial_coefficient()`.

EXAMPLES:

```python
sage: A.<x,z,y> = FreeAlgebra(QQ, 3)
sage: R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: R.inject_variables()
Defining x, z, y
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})
z + y^2 + y + 1
sage: f.coefficient(x)
y^2 - y + 1
sage: f.coefficient([0,None]) # not tested
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.

```python
sage: f.coefficient(x^0) # outputs the full polynomial
x^2*y^2 + x^2*y + x^2 + x*y^2 - x*y + x + z + y^2 + y + 1
sage: A.<x,z,y> = FreeAlgebra(GF(389), 3)
sage: R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: R.inject_variables()
Defining x, z, y
sage: f=x*y+5
sage: c=f.coefficient({x:0,y:0}); c
5
sage: parent(c)
Noncommutative Multivariate Polynomial Ring in x, z, y over Finite Field of size 389, nc-relations: (y*x: -x*y + z)
```
constant_coefficient()

Return the constant coefficient of this multivariate polynomial.

EXAMPLES:

```
sage: A.<x,z,y> = FreeAlgebra(GF(389), 3)
sage: P = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: P.inject_variables()
Defining x, z, y
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
```
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
A.<x,z,y> = FreeAlgebra(GF(389), 3)
R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
R.inject_variables()
Defining x, z, y
f = (2+x*y^3+z^2 + (7)*x^2 + (3))
f.dict()
{(0, 0, 0): 3, (1, 2, 3): 2, (2, 0, 0): 7}
```

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
A.<x,z,y> = FreeAlgebra(GF(389), 3)
R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
R.inject_variables()
Defining x, z, y
f = x^3 + y + 2*z^2
f.exponents()
[(3, 0, 0), (0, 2, 0), (0, 0, 1)]
f.exponents(as_ETuples=False)
[(3, 0, 0), (0, 2, 0), (0, 0, 1)]
```

is_constant()

Return True if this polynomial is constant.

EXAMPLES:

```sage
A.<x,z,y> = FreeAlgebra(GF(389), 3)
P = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
P.inject_variables()
Defining x, z, y
x.is_constant()  # False
P(1).is_constant()  # True
```

is_homogeneous()

Return True if this polynomial is homogeneous.

EXAMPLES:
A monomial is defined to be a product of generators with coefficient 1.

EXAMPLES:

```python
sage: A.<x,z,y> = FreeAlgebra(GF(389), 3)
sage: P = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: P.inject_variables()
Defining x, z, y
sage: x.is_monomial()
True
sage: (2*x).is_monomial()
False
sage: (x*y).is_monomial()
True
sage: (x*y + x).is_monomial()
False
```

A monomial is defined to be a product of generators with coefficient 1.

EXAMPLES:

```python
sage: A.<x,z,y> = FreeAlgebra(GF(389), 3)
sage: P = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: P.inject_variables()
Defining x, z, y
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_zero()  
Return True if this polynomial is zero.

EXAMPLES:

```python
sage: A.<x,z,y> = FreeAlgebra(QQ, 3)
sage: R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: R.inject_variables()
Defining x, z, y
sage: x.is_zero()
False
sage: (x-x).is_zero()
True
```

c()  
Leading coefficient of this polynomial with respect to the term order of self.parent().

EXAMPLES:

```python
sage: A.<x,z,y> = FreeAlgebra(QQ, 3)
sage: R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: R.inject_variables()
Defining x, z, y
sage: x.is_zero()
False
sage: (x-x).is_zero()
True
```
sage: A.<x,y,z> = FreeAlgebra(GF(7), 3)
sage: R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: R.inject_variables()
Defining x, y, z

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

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: A.<x,y,z> = FreeAlgebra(GF(7), 3)
sage: R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: R.inject_variables()
Defining x, y, z

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: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: R = A.g_algebra(relations={y*x:-x*y + z}, order='deglex')
sage: R.inject_variables()

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

lt()

Leading term of this polynomial.

In Sage a term is a product of variables in some power and a coefficient.

EXAMPLES:
sage: A.<x,y,z> = FreeAlgebra(GF(7), 3)
sage: R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: R.inject_variables()
Defining x, y, z
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: A.<x,z,y> = FreeAlgebra(GF(389), 3)
sage: P = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: P.inject_variables()
Defining x, z, y
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()
Finite Field of size 389
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)
388
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: A.<x,z,y> = FreeAlgebra(GF(389), 3)
sage: P = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: P.inject_variables()
Defining x, z, y
sage: f = x + (3*2)*y*z^2 + (2+3)
sage: f.monomials()
[x, z^2*y, 1]
sage: f = P(3^2)
sage: f.monomials()
[1]
```

reduce(I)

EXAMPLES:

```
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)
The result of reduction is not the normal form, if one reduces by a list of polynomials:

```
sage: (x*z).reduce(I.gens())
x*z
```

However, if the argument is an ideal, then a normal form (reduction with respect to a two-sided Groebner basis) is returned:

```
sage: (x*z).reduce(I)
-x
```

The Groebner basis shows that the result is correct:

```
sage: I.std() #random
Left Ideal (z^2 - 1, y*z - y, x*z + x, y^2, 2*x*y - z - 1, x^2) of
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational
Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z}
sage: sorted(I.std().gens(),key=str)
[2*x*y - z - 1, x*z + x, x^2, y*z - y, y^2, z^2 - 1]
```

total_degree()

Return the total degree of self, which is the maximum degree of all monomials in self.

EXAMPLES:

```
sage: A.<x,z,y> = FreeAlgebra(QQ, 3)
sage: R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: R.inject_variables()
Defining x, z, y
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
```

(continues on next page)
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

sage.rings.polynomial.plural.SCA(base_ring, names, alt_vars, order='degrevlex')

Return a free graded-commutative algebra

This is also known as a free super-commutative algebra.

INPUT:

- base_ring – the ground field
- names – a list of variable names
- alt_vars – a list of indices of to be anti-commutative variables (odd variables)
- order – ordering to be used for the constructed algebra

EXAMPLES:

sage: from sage.rings.polynomial.plural import SCA
sage: E = SCA(QQ, ['x', 'y', 'z'], [0, 1], order = 'degrevlex')

Quotient of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {y*x: -x*y} by the ideal (y^2, x^2)

sage: E.inject_variables()
Defining xbar, ybar, zbar
sage: x,y,z = (xbar,ybar,zbar)

sage: x*y
-x*y
sage: z*x
x*z
sage: x^2
0
sage: y^2
0
sage: z^2
z^2
sage: E.one()
1

sage.rings.polynomial.plural.new_CRing(rw, base_ring)

Construct MPolynomialRing_libsingular from ringWrap, assuming the ground field to be base_ring

EXAMPLES:

sage: H.<x,y,z> = PolynomialRing(QQ, 3)
sage: from sage.libs.singular.function import singular_function

sage: ringlist = singular_function('ringlist')
sage: ring = singular_function("ring")
\begin{quote}
\begin{verbatim}
sage: L = ringlist(H, ring=H); L
[0, ['x', 'y', 'z'], [['dp', (1, 1, 1)], ['C', (0,)]], [0]]
sage: len(L)
4
sage: W = ring(L, ring=H); W
<RingWrap>
sage: from sage.rings.polynomial.plural import new_CRing
sage: R = new_CRing(W, H.base_ring())
sage: R
# indirect doctest
Multivariate Polynomial Ring in x, y, z over Rational Field
\end{verbatim}
\end{quote}
Check that trac ticket \#13145 has been resolved:
\begin{quote}
\begin{verbatim}
sage: h = hash(R.gen() + 1) # sets currRing
sage: from sage.libs.singular.ring import ring_refcount_dict, currRing_wrapper
sage: curcnt = ring_refcount_dict[currRing_wrapper()]
sage: newR = new_CRing(W, H.base_ring())
sage: ring_refcount_dict[currRing_wrapper()] - curcnt
1
\end{verbatim}
\end{quote}
sage: len(L)
6
sage: W = ring(L, ring=H); W
<noncommutative RingWrap>

sage: from sage.rings.polynomial.plural import new_NRing
sage: R = new_NRing(W, H.base_ring())
sage: R
# indirect doctest
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {y*x: x*y - 1}

sage.rings.polynomial.plural.new_Ring(ring, base_ring)
Constructs a Sage ring out of low level RingWrap, which wraps a pointer to a Singular ring.

The constructed ring is either commutative or noncommutative depending on the Singular ring.

EXAMPLES:

sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-1})
sage: H.inject_variables()
Defining x, y, z
sage: z*x
x*z
sage: z*y
y*z
sage: y*x
x*y - 1
sage: I = H.ideal([y^2, x^2, z^2-1])
sage: I._groebner_basis_libsingular()
[1]
sage: from sage.libs.singular.function import singular_function
sage: ringlist = singular_function('ringlist')
sage: ring = singular_function("ring")

sage: L = ringlist(H, ring=H); L
[
  [0 1 1]
  [0 0 1]
0, ['x', 'y', 'z'], [['dp', (1, 1, 1)], ['C', (0,)], [0], [0 0 0],
  [ 0 -1 0]
  [ 0 0 0]
]
sage: len(L)
6

sage: W = ring(L, ring=H); W
<noncommutative RingWrap>

sage: from sage.rings.polynomial.plural import new_Ring
sage: R = new_Ring(W, H.base_ring()); R
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {y*x: x*y - 1}
sage.rings.polynomial.plural.unpickle_NCPolynomial_plural($R, d$)

Auxiliary function to unpickle a non-commutative polynomial.
INDICES AND TABLES

- Index
- Module Index
- Search Page
sage.rings.fraction_field, 505
sage.rings.fraction_field_element, 510
sage.rings.fraction_field_FpT, 515
sage.rings.invariants.invariant_theory, 407
sage.rings.invariants.reconstruction, 450
sage.rings.monomials, 407
sage.rings.polynomial.convolution, 237
sage.rings.polynomial.cyclotomic, 238
sage.rings.polynomial.flatten, 404
sage.rings.polynomial.hilbert, 403
sage.rings.polynomial.ideal, 213
sage.rings.polynomial.infinite_polynomial_element, 562
sage.rings.polynomial.infinite_polynomial_ring, 553
sage.rings.polynomial.laurent_polynomial, 530
sage.rings.polynomial.laurent_polynomial_ring, 523
sage.rings.polynomial.multi_polynomial, 267
sage.rings.polynomial.multi_polynomial_element, 290
sage.rings.polynomial.multi_polynomial_ideal, 305
sage.rings.polynomial.multi_polynomial_ideal_libsingular, 391
sage.rings.polynomial.multi_polynomial_libsingular, 366
sage.rings.polynomial.multi_polynomial_ring, 287
sage.rings.polynomial.multi_polynomial_ring_base, 258
sage.rings.polynomial.multi_polynomial_sequence, 352
sage.rings.polynomial.omega, 547
sage.rings.polynomial.padics.polynomial_padic, 167
sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense, 170
sage.rings.polynomial.padics.polynomial_padic_flat, 177
sage.rings.polynomial.pbori, 587
sage.rings.polynomial.plural, 645
sage.rings.polynomial.polydict, 393
sage.rings.polynomial.polynomialCompiled, 236
sage.rings.polynomial.polynomial_element, 30
sage.rings.polynomial.polynomial_element_generic, 107
sage.rings.polynomial.polynomial_fateman, 237
sage.rings.polynomial.polynomial_gf2x, 116
sage.rings.polynomial.polynomial_integer_dense_flint, 123
sage.rings.polynomial.polynomial_integer_dense_ntl, 131
sage.rings.polynomial.polynomial_modn_dense_ntl, 154
sage.rings.polynomial.polynomial_number_field, 120
sage.rings.polynomial.polynomial_quotient_ring, 214
sage.rings.polynomial.polynomial_quotient_ring_element, 232
sage.rings.polynomial.polynomial_rational_flint, 136
sage.rings.polynomial.polynomial_real_mpfr_dense, 164
sage.rings.polynomial.polynomial_ring, 9
sage.rings.polynomial.polynomial_ring_constructor, 1
sage.rings.polynomial.polynomial_ring_homomorphism, 29
sage.rings.polynomial.polynomial_singular_interface, 166
sage.rings.polynomial.polynomial_zmod_flint, 147
sage.rings.polynomial.polynomial_zz_pex, 178
sage.rings.polynomial.real_roots, 181
sage.rings.polynomial.refine_root, 212
sage.rings.polynomial.skew_polynomial_element, 467
sage.rings.polynomial.skew_polynomial_ring, 497
sage.rings.polynomial.skew_polynomial_ring_constructor, 495
sage.rings.polynomial.symmetric_ideal, 571
sage.rings.polynomial.symmetric_reduction, 580
sage.rings.polynomial.term_order, 241
sage.rings.polynomial.toy_buchberger, 453
sage.rings.polynomial.toy_d_basis, 462
sage.rings.polynomial.toy_variety, 458
Symbols

_add_() (sage.rings.polynomial.polynomial_element.Polynomial method), 31
_add_() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 123
_add_() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl ZZ method), 159
_add_() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 136
_add_() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 149
_lmul_() (sage.rings.polynomial.polynomial_element.Polynomial method), 31
_lmul_() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 123
_lmul_() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl ZZ method), 159
_lmul_() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 136
_lmul_() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 150
_mul_() (sage.rings.polynomial.polynomial_element.Polynomial method), 32
_mul_() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 123
_mul_() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl ZZ method), 159
_mul_() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 137
_mul_() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 150
_mul_trunc_() (sage.rings.polynomial.polynomial_element.Polynomial method), 32
_mul_trunc_() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 123
_mul_trunc_() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl ZZ method), 159
_mul_trunc_() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 137
_mul_trunc_() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 150
_rmul_() (sage.rings.polynomial.polynomial_element.Polynomial method), 32
_rmul_() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 123
_rmul_() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl ZZ method), 159
_rmul_() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 137
_rmul_() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 150
_sub_() (sage.rings.polynomial.polynomial_element.Polynomial method), 31
_sub_() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 123
_sub_() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl ZZ method), 159
_sub_() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 136
_sub_() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 150

A

A_invartant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 415
abc_pd (class in sage.rings.polynomial.polynomial_compiled), 236
add_adams_operator()  (sage.rings.polynomial.polynomial_element.Polynomial method), 33
add_as_you_wish()  (sage.rings.polynomial.pbori.GroebnerStrategy method), 629
add_bigoh()  (sage.rings.polynomial.polynomial_element.Polynomial method), 33
add_generator()  (sage.rings.polynomial.pbori.GroebnerStrategy method), 629
add_generator()  (sage.rings.polynomial.symmetric_reduction.SymmetricReductionStrategy method), 582
add_generator_delayed()  (sage.rings.polynomial.pbori.GroebnerStrategy method), 630
add_m_mul_q()  (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 373
add_pd  (class in sage.rings.polynomial.pbori), 636
add_up_polynomials()  (in module sage.rings.polynomial.pbori), 636
algebraic_dependence()  (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 356
AlgebraicForm  (class in sage.rings.invariants.invariant_theory), 409
all_done()  (sage.rings.polynomial.real_roots.ocean method), 200
all_generators()  (sage.rings.polynomial.pbori.GroebnerStrategy method), 630
all_roots_in_interval()  (sage.rings.polynomial.polynomial_element.Polynomial method), 34
all_spolys_in_next_degree()  (sage.rings.polynomial.pbori.GroebnerStrategy method), 630
alpha_covariant()  (sage.rings.invariants.invariant_theory.BinaryQuintic method), 417
ambient()  (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 223
an_element()  (sage.rings.polynomial.skew_polynomial_element.SkewPolynomialBaseringInjection method), 490
any_root()  (sage.rings.polynomial.polynomial_element.Polynomial method), 34
append()  (sage.rings.polynomial.pbori.BooleanPolynomialVector method), 628
approx_bp()  (sage.rings.polynomial.real_roots.ocean method), 200
args()  (sage.rings.polynomial.multi_polynomial.MPolynomial method), 267
args()  (sage.rings.polynomial.polynomial_element.Polynomial method), 35
arithmetic_invariants()  (sage.rings.invariants.invariant_theory.BinaryQuintic method), 417
as_float()  (sage.rings.polynomial.real_roots.interval_bernstein_polynomial_integer method), 193
as_QuadraticForm()  (sage.rings.invariants.invariant_theory.QuadraticForm method), 433
associated_primes()  (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 324

B
B_invariant()  (sage.rings.invariants.invariant_theory.BinaryQuintic method), 415
base_extend()  (sage.rings.polynomial.polynomial_element.Polynomial method), 35
base_extend()  (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 20
base_field()  (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_field method), 219
base_ring()  (sage.rings.fraction_field.FractionField_generic method), 508
base_ring()  (sage.rings.polynomial.polynomial_element.Polynomial method), 35
base_ring()  (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal method), 223
base_ring()  (sage.rings.polynomial.multi_polynomial_quotient_ring.PolynomialQuotientRing_field method), 470
basis  (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal attribute), 308
basis_is_groebner()  (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 325
bateman_bound()  (in module sage.rings.polynomial.cyclotomic), 238
bernstein_down()  (in module sage.rings.polynomial.real_roots), 182
bernstein_expand()  (in module sage.rings.polynomial.real_roots), 182
bernstein_polynomial()  (sage.rings.polynomial.real_roots.bernstein_polynomial_factory_ar method), 183
bernstein_polynomial()  (sage.rings.polynomial.real_roots.bernstein_polynomial_factory_intlist method), 183
bernstein_polynomial() (sage.rings.polynomial.real_roots.bernstein_polynomial_factory_ratlist method), 184
bernstein_polynomial_factory (class in sage.rings.polynomial.real_roots), 182
bernstein_polynomial_factory_ar (class in sage.rings.polynomial.real_roots), 183
bernstein_polynomial_factory_intlist (class in sage.rings.polynomial.real_roots), 183
bernstein_polynomial_factory_ratlist (class in sage.rings.polynomial.real_roots), 184
bernstein_up() (in module sage.rings.polynomial.real_roots), 185
beta_covariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 418
binary_cubic_coefficients_from_invariants() (in module sage.rings.invariants.reconstruction), 450
binary_form_from_invariants() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 426
binary_pd (class in sage.rings.polynomial.polynomial compiled), 236
binary_quadratic() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 427
binary_quadratic_coefficients_from_invariants() (in module sage.rings.invariants.reconstruction), 450
binary_quartic() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 428
binary_quintic() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 428
binary_quintic_coefficients_from_invariants() (in module sage.rings.invariants.reconstruction), 451
BinaryQuartic (class in sage.rings.invariants.invariant_theory), 412
BinaryQuintic (class in sage.rings.invariants.invariant_theory), 414
bitsize_doctest() (in module sage.rings.polynomial.real_roots), 185
blocks() (sage.rings.polynomial.term_order.TermOrder method), 246
BooleanMonomial (class in sage.rings.polynomial.pbore), 597
BooleanMonomial_iterator (class in sage.rings.polynomial.pbore), 600
BooleanMonomial_monoid (class in sage.rings.polynomial.pbore), 600
BooleanMonomial_variable_iterator (class in sage.rings.polynomial.pbore), 602
BooleanMulAction (class in sage.rings.polynomial.pbore), 602
BooleanPolynomial (class in sage.rings.polynomial.pbore), 602
BooleanPolynomialEntry (class in sage.rings.polynomial.pbore), 617
BooleanPolynomial_ideal (class in sage.rings.polynomial.pbore), 617
BooleanPolynomialIterator (class in sage.rings.polynomial.pbore), 620
BooleanPolynomialRing (class in sage.rings.polynomial.pbore), 620
BooleanPolynomialRing_constructor () (in module sage.rings.polynomial.polynomial_ring_constructor), 1
BooleanPolynomialVector (class in sage.rings.polynomial.pbore), 628
BooleanPolynomialVector_iterator (class in sage.rings.polynomial.pbore), 628
BooleanConstant (class in sage.rings.polynomial.pbore), 589
BooleSet (class in sage.rings.polynomial.pbore), 590
BooleSet_iterator (class in sage.rings.polynomial.pbore), 597
bp_done() (sage.rings.polynomial.real_roots.island method), 196
buchberger() (in module sage.rings.polynomial.toy_buchberger), 456
buchberger_improved() (in module sage.rings.polynomial.toy_buchberger), 456

C
C_invariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 415
can_convert_to_singular() (in module sage.rings.polynomial.polynomial_singular_interface), 167
can_rewrite() (sage.rings.polynomial.pbore.ReductionStrategy method), 634
canonical_form() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 419
cardinality() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 223

cartesian_product() (sage.rings.polynomial.pbori.BooleSet method), 628

CCuddNavigator (class in sage.rings.polynomial.pbori), 591

change() (sage.rings.polynomial.pbori.BooleSet method), 591

change_ring() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial method), 259

change_ring() (sage.rings.polynomial.multi_polynomial_element.MPolynomial method), 267

change_ring() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_element method), 291

change_ring() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal method), 308

change_ring() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 259

change_ring() (sage.rings.polynomial.pbori.BooleanPolynomialRing method), 620

change_ring() (sage.rings.polynomial.polynomial_element.Polynomial method), 35

change_ring() (sage.rings.polynomial.polynomial_real_mpfr_dense.PolynomialRealDense method), 164

change_ring() (sage.rings.polynomial.pbori.BooleSet method), 591

change_var() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 471

change_variable_name() (sage.rings.polynomial.polynomial_element.Polynomial method), 36

change variable name() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 471

characteristic() (sage.rings.fraction_field.FractionField_1poly_field method), 507

characteristic() (sage.rings.invariants.invariant_theory.AlgebraicForm method), 410

characteristic() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse method), 559

characteristic() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomialRing_general method), 20

characteristic() (sage.rings.polynomial.multi_polynomial_element.MPolynomial method), 259

characteristic() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_element method), 291

characteristic() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal method), 308

characteristic() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 259

characteristic() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 224

characteristic() (sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement method), 233

cheap_reductions() (sage.rings.polynomial.pbori.ReductionStrategy method), 635

cl_maximum_root() (in module sage.rings.polynomial.real_roots), 185

cl_maximum_root_first_lambda() (in module sage.rings.polynomial.real_roots), 185

cl_maximum_root_local_max() (in module sage.rings.polynomial.real_roots), 185

class_group() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse method), 559

class_number() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomialRing_general method), 20

clebsch_invariants() (sage.rings.invariants.invariant_theory.BinomialQuintic method), 419

clean_top_by_chain_criterion() (sage.rings.polynomial.pbori.GroebnerStrategy method), 630

coefficients() (sage.rings.invariants.invariant_theory.AlgebraicForm method), 410

coefficient_matrix() (in module sage.rings.polynomial.toy_variety), 459

coefficient_matrix() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 357

coefficient() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomialSparse method), 233

coefficient() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomialRing_general method), 20

coefficient() (sage.rings.polynomial.multi_polynomial_element.MPolynomial method), 259

coefficient() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_element method), 291

coefficient() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 373

coefficient() (sage.rings.polynomial.plural.NCPolynomial_plural method), 652

coefficient() (sage.rings.polynomial.polydict.PolyDict method), 398

coefficient_matrix() (in module sage.rings.polynomial.toy_variety), 459

coefficient_matrix() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 357

coefficient() (sage.rings.invariants.invariant_theory.AlgebraicForm method), 410

672 Index
coefficients() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 532
coefficients() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 540
coefficients() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 268
coefficients() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 374
coefficients() (sage.rings.polynomial.polynomial_element.Polynomial method), 36
coefficients() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse method), 112
coefficients() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 471
coefficients() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial_generic_dense method), 490
coeffs() (sage.rings.invariants.invariant_theory.BinaryQuartic method), 412
coeffs() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 420
coeffs() (sage.rings.invariants.invariant_theory.QuadraticForm method), 433
coeffs() (sage.rings.invariants.invariant_theory.TernaryCubic method), 439
coeffs() (sage.rings.invariants.invariant_theory.TernaryQuadratic method), 441
coeffs_bitsize() (sage.rings.polynomial.real_roots.bernstein_polynomial_factory_ar method), 183
coeffs_bitsize() (sage.rings.polynomial.real_roots.bernstein_polynomial_factory_intlist method), 184
coeffs_bitsize() (sage.rings.polynomial.real_roots.bernstein_polynomial_factory_ratlist method), 184
combine_to_positives() (sage.rings.polynomial.polydict.ETuple method), 393
common_nonzero_positions() (sage.rings.polynomial.polydict.ETuple method), 393
CompiledPolynomialFunction (class in sage.rings.polynomial.polynomial_compiled), 236
complete_primary_decomposition() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 326
completion() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic method), 526
completion() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 259
complex_embeddings() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_field method), 219
complex_roots() (in module sage.rings.polynomial.complex_roots), 210
complex_roots() (sage.rings.polynomial.polynomial_element.Polynomial method), 36
compose_power() (sage.rings.polynomial.polynomial_element.Polynomial method), 36
compose_trunc() (sage.rings.polynomial.polynomial_element.Polynomial method), 37
composed_op() (sage.rings.polynomial.polynomial_element.Polynomial method), 37
conjugate() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 471
connected_components() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 358
connection_graph() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 358
constant() (sage.rings.polynomial.pbori.BooleanPolynomial method), 602
constant() (sage.rings.polynomial.pbori.CCuddNavigator method), 628
constant_coefficient() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 532
constant_coefficient() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 540
constant_coefficient() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_libdict method), 293
constant_coefficient() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_libsingular method), 374
constant_coefficient() (sage.rings.polynomial.pbori.BooleanPolynomial method), 602
constant_coefficient() (sage.rings.polynomial.plural.NCPolynomial_plural method), 654
constant_coefficient() (sage.rings.polynomial.polynomial_element.Polynomial method), 39

Index 673
constant_coefficient() (sage.rings.polynomial.polynomial_element.Polynomial_generic_dense method), 102
constant_coefficient() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 472
ConstantPolynomialSection (class in sage.rings.polynomial.polynomial_element), 30
ConstantSkewPolynomialSection (class in sage.rings.polynomial.skew_polynomial_element), 467
construction() (sage.rings.fraction_field.FractionField_generic method), 508
construction() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_dense method), 558
construction() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse method), 559
construction() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic method), 526
construction() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 260
construction() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 226
construction() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 21
contained_vars() (in module sage.rings.polynomial.pbori), 636
contains_one() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 268
contains_one() (sage.rings.polynomial.padic.poly_ring.padic.Polynomial_padic method), 167
content() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 268
content() (sage.rings.polynomial.padics.polynomial_padic.Polynomial_padic method), 124
content() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 131
content() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 124
content_ideal() (sage.rings.polynomial.multi_polynomial.Polynomial method), 269
context (class in sage.rings.polynomial.real_roots), 186
convolution() (in module sage.rings.polynomial.convolution), 237
covariant_conic() (sage.rings.invariants.invariant_theory.TernaryQuadratic method), 441
cover_ring() (sage.rings.polynomial.pbori.BooleanPolynomialRing method), 622
cover_ring() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 216
create_key() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_generic method), 557
create_key() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRingFactory method), 216
create_key_and_extra_args() (sage.rings.polynomial.plural.G_AlgFactory method), 647
create_object() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRingFactory method), 557
create_object() (sage.rings.polynomial.plural.G_AlgFactory method), 647
create_object() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRingFactory method), 216
cyclotomic_coeffs() (in module sage.rings.polynomial.cyclotomic), 238
cyclotomic_part() (sage.rings.polynomial.polynomial_element.Polynomial method), 39
cyclotomic_polynomial() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 21
cyclotomic_value() (in module sage.rings.polynomial.cyclotomic), 239

d

d_basis() (in module sage.rings.polynomial.toy_d_basis), 464
de_casteljau() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial_float method), 191
de_casteljau() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial_integer method), 193
de_casteljau_doublevec() (in module sage.rings.polynomial.real_roots), 186
de_casteljau_intvec() (in module sage.rings.polynomial.real_roots), 186
defining_ideal() (sage.rings.polynomial.pbori.BooleanPolynomialRing method), 622
deg() (sage.rings.polynomial.pbori.BooleanMonomial method), 597
deg() (sage.rings.polynomial.pbori.BooleanPolynomial method), 602
deg() (sage.rings.polynomial.pbori.BooleConstant method), 589
degree() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 533
degree() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 540
degree() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 293
degree() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 374
degree() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 170
degree() (sage.rings.polynomial.pbori.BooleanMonomial method), 597
degree() (sage.rings.polynomial.pbori.BooleanPolynomial method), 603
degree() (sage.rings.polynomial.plural.NCPolynomial_plural method), 654
degree() (sage.rings.polynomial.polynomial_dict.PolyDict method), 398
degree() (sage.rings.polynomial.polynomial_element.Polynomial method), 40
degree() (sage.rings.polynomial.polynomial_element.Polynomial_generic_dense method), 102
degree() (sage.rings.polynomial.polynomial_element.Polynomial_generic_dense_inexact method), 104
degree() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse method), 112
degree() (sage.rings.polynomial.polynomial_gf2x.Polynomial_template method), 118
degree() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 124
degree() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 124
degree() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 132
degree() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 154
degree() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 157
degree() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 159
degree() (sage.rings.polynomial.polynomial_multivariate.PolynomialQuotientRing_generic method), 226
degree() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 137
degree() (sage.rings.polynomial.polynomial_real_mpfr_dense.PolynomialRealDense method), 164
degree() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_template method), 148
degree() (sage.rings.polynomial.real_roots.PolynomialRealRoots method), 179
degree() (sage.rings.polynomial.real_roots.PolynomialRealRoots method), 179
degree() (sage.rings.polynomial.real_roots.PolynomialRealRoots method), 179
degree() (sage.rings.polynomial.real_roots.PolynomialRealRoots method), 179
degree() (sage.rings.polynomial.real_roots.PolynomialRealRoots method), 179
Delta_invariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 420
Delta_invariant() (sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics method), 443
Delta_invariant() (sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics method), 446
Delta_prime_invariant() (sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics method), 443
Delta_prime_invariant() (sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics method), 446
denom() (sage.rings.fraction_field_FpT.FpTElement method), 515
denominator() (sage.rings.fraction_field_FpT.FpTElement method), 511
denominator() (sage.rings.fraction_field_FpT.FpTElement method), 516
denominator() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 269
denominator() (sage.rings.polynomial.polynomial_element.Polynomial method), 40
denominator() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 138
derivative() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_laurent_polynomial method), 533
derivative() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 540
derivative() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 270
derivative() (sage.rings.polynomial.multi_polynomial_univariate.Polynomial method), 41
dict() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_laurent_polynomial method), 531
dict() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_laurent_polynomial method), 533
dict() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 541
dict() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 295

dict() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 375

dict() (sage.rings.polynomial.plural.NCPolynomial_plural method), 655

dict() (sage.rings.polynomial.polydict.PolyDict method), 398

dict() (sage.rings.polynomial.polynomial_element.Polynomial method), 42

dict() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse method), 112

diff() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 533

diff() (sage.rings.polynomial.pbori.BooleSet method), 592

differentiate() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 534

differentiate() (sage.rings.polynomial.polynomial_element.Polynomial method), 43

dimension() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 328

dimension() (sage.rings.polynomial.pbori.BooleanPolynomialIdeal method), 618

disc() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 171

disc() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 124

discriminant() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 138

discriminant() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 271

discriminant() (sage.rings.polynomial.polynomial_element.Polynomial method), 43

discriminant() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 125

discriminant() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 132

discriminant() (sage.rings.polynomial.polynomial_modn_dense_flint.Polynomial_dense_mod_p method), 156

discriminant() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 226

discriminant() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 139

dispersion() (sage.rings.polynomial.multi_polynomial_element.Polynomial method), 44

dispersion_set() (sage.rings.polynomial.multi_polynomial_element.Polynomial method), 44

divide() (sage.rings.polynomial.pbori.BooleSet method), 592

divided_difference() (sage.rings.polynomial.polynomial_ring.PolynomialRing_field method), 17

divides() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 375

divides() (sage.rings.polynomial.polynomial_element.Polynomial method), 45

divisors() (sage.rings.polynomial.pbori.BooleMonomial method), 598

divisors_of() (sage.rings.polynomial.pbori.BooleSet method), 592

done() (sage.rings.polynomial.real_roots.island method), 197

dotprod() (sage.rings.polynomial.polydict.ETuple method), 394

down_degree() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial_integer method), 194

down_degree_iter() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial_integer method), 194

downscale() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial_integer method), 194

dprod_imatrow_vec() (in module sage.rings.polynomial.real_roots), 187

dual() (sage.rings.invariants.invariant_theory.QuadraticForm method), 434

dummy_pd (class in sage.rings.polynomial.polynomial_compiled), 236

E

eadd() (sage.rings.polynomial.polydict.ETuple method), 394

eadd_p() (sage.rings.polynomial.polydict.ETuple method), 394

easy_linear_factors() (in module sage.rings.polynomial.pbori), 637
EisensteinD() (sage.rings.invariants.invariant_theory.BinaryQuartic method), 412
EisensteinE() (sage.rings.invariants.invariant_theory.BinaryQuartic method), 412
Element (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_mpair attribute), 529
Element (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_univariate attribute), 530
Element (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular attribute), 369
Element (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic attribute), 220
element() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_element method), 291
elength() (sage.rings.polynomial.pbori.BooleanPolynomial method), 603
elim_pol() (in module sage.rings.polynomial.toy_variety), 459
eliminate_linear_variables() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_gf2 method), 363
elimination_ideal() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 329
elem() (sage.rings.polynomial.pbori.CCuddNavigator method), 628
emax() (sage.rings.polynomial.polydict.ETuple method), 395
emin() (sage.rings.polynomial.polydict.ETuple method), 395
empty() (sage.rings.polynomial.pbori.BooleSet method), 592
emul() (sage.rings.polynomial.polydict.ETuple method), 395
escalar_div() (sage.rings.polynomial.polydict.ETuple method), 395
esub() (sage.rings.polynomial.polydict.ETuple method), 396
ETuple (class in sage.rings.polynomial.polydict), 393
euclidean_degree() (sage.rings.polynomial.polynomial_element.Polynomial method), 45
exponents() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 534
exponents() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 541
exponents() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 295
exponents() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 376
exponents() (sage.rings.polynomial.plural.NCPolynomial_plural method), 655
exponents() (sage.rings.polynomial.polydict.PolyDict method), 398
exponents() (sage.rings.polynomial.polynomial_element.Polynomial method), 46
exponents() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse method), 112
exponents() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 473
extend_variables() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 21
ExteriorAlgebra () (in module sage.rings.polynomial.plural), 646
ExteriorAlgebra_plural (class in sage.rings.polynomial.plural), 647

F
F_covariant () (sage.rings.invariants.invariant_theory.TwoTernaryQuadratics method), 447
factor() (sage.rings.fraction_field_FpT.FpTElement method), 516
factor() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 534
factor() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 541
factor() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 295
factor() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 376
factor() (sage.rings.polynomial.padics.polynomial_padic.Polynomial_padic method), 168
factor() (sage.rings.polynomial.polynomial_element.Polynomial method), 46
factor() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 125
factor() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 132
factor() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 151
factor_mod() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 171

Index
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>factor_mod()</td>
<td>(sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 125</td>
</tr>
<tr>
<td>factor_mod()</td>
<td>(sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 133</td>
</tr>
<tr>
<td>factor_mod()</td>
<td>(sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 139</td>
</tr>
<tr>
<td>factor_of_slope()</td>
<td>(sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdv method), 107</td>
</tr>
<tr>
<td>factor_padic()</td>
<td>(sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 126</td>
</tr>
<tr>
<td>factor_padic()</td>
<td>(sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 133</td>
</tr>
<tr>
<td>factor_padic()</td>
<td>(sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 140</td>
</tr>
<tr>
<td>faugere_step_dense()</td>
<td>(sage.rings.polynomial.pbori.GroebnerStrategy method), 631</td>
</tr>
<tr>
<td>fcp()</td>
<td>(sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement method), 233</td>
</tr>
<tr>
<td>FGLMStrategy</td>
<td>(class in sage.rings.polynomial.pbori), 628</td>
</tr>
<tr>
<td>field_extension()</td>
<td>(sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingDomain method), 217</td>
</tr>
<tr>
<td>field_extension()</td>
<td>(sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement method), 233</td>
</tr>
<tr>
<td>find_roots()</td>
<td>(sage.rings.polynomial.real_roots.ocean method), 200</td>
</tr>
<tr>
<td>first()</td>
<td>(sage.rings.invariants.invariant_theory.TwoAlgebraicForms method), 442</td>
</tr>
<tr>
<td>first_hilbert_series()</td>
<td>(in module sage.rings.polynomial.hilbert), 403</td>
</tr>
<tr>
<td>first_term()</td>
<td>(sage.rings.polynomial.pbori.BooleanPolynomial method), 604</td>
</tr>
<tr>
<td>flattening_morphism()</td>
<td>(sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 260</td>
</tr>
<tr>
<td>flattening_morphism()</td>
<td>(sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 21</td>
</tr>
<tr>
<td>FlatteningMorphism</td>
<td>(class in sage.rings.polynomial.flatten), 405</td>
</tr>
<tr>
<td>footprint()</td>
<td>(sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse method), 565</td>
</tr>
<tr>
<td>form()</td>
<td>(sage.rings.invariants.invariant_theory.AlgebraicForm method), 410</td>
</tr>
<tr>
<td>FormsBase</td>
<td>(class in sage.rings.invariants.invariant_theory), 424</td>
</tr>
<tr>
<td>Fp_FpT_coerce</td>
<td>(class in sage.rings.fraction_field_FpT), 520</td>
</tr>
<tr>
<td>FpT</td>
<td>(class in sage.rings.fraction_field_FpT), 515</td>
</tr>
<tr>
<td>FpT_Fp_section</td>
<td>(class in sage.rings.fraction_field_FpT), 518</td>
</tr>
<tr>
<td>FpT_iter</td>
<td>(class in sage.rings.fraction_field_FpT), 519</td>
</tr>
<tr>
<td>FpT_Polyring_section</td>
<td>(class in sage.rings.fraction_field_FpT), 519</td>
</tr>
<tr>
<td>FpTElement</td>
<td>(class in sage.rings.fraction_field_FpT), 515</td>
</tr>
<tr>
<td>fraction_field()</td>
<td>(sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic method), 526</td>
</tr>
<tr>
<td>FractionField()</td>
<td>(in module sage.rings.fraction_field), 505</td>
</tr>
<tr>
<td>FractionField_1poly_field</td>
<td>(class in sage.rings.fraction_field), 507</td>
</tr>
<tr>
<td>FractionField_generic</td>
<td>(class in sage.rings.fraction_field), 508</td>
</tr>
<tr>
<td>FractionFieldElement</td>
<td>(class in sage.rings.fraction_field_element), 510</td>
</tr>
<tr>
<td>FractionFieldElement_1poly_field</td>
<td>(class in sage.rings.fraction_field_element), 513</td>
</tr>
<tr>
<td>FractionFieldEmbedding</td>
<td>(class in sage.rings.fraction_field), 506</td>
</tr>
<tr>
<td>FractionFieldEmbeddingSection</td>
<td>(class in sage.rings.fraction_field), 507</td>
</tr>
<tr>
<td>FractionSpecializationMorphism</td>
<td>(class in sage.rings.polynomial.flatten), 406</td>
</tr>
<tr>
<td>free_algebra()</td>
<td>(sage.rings.polynomial.plural.NCPolynomialRing_plural method), 648</td>
</tr>
<tr>
<td>from_invariants()</td>
<td>(sage.rings.invariants.invariant_theory.BinaryQuintic class method), 420</td>
</tr>
<tr>
<td>from_invariants()</td>
<td>(sage.rings.invariants.invariant_theory.QuadraticForm class method), 434</td>
</tr>
<tr>
<td>from_ocean()</td>
<td>(sage.rings.polynomial.real_roots.linear_map method), 197</td>
</tr>
</tbody>
</table>
from_ocean() (sage.rings.polynomial.real_roots.warp_map method), 209
function_field() (sage.rings.fraction_field.FractionField_Lpoly_field method), 507

G

G_AlgFactory (class in sage.rings.polynomial.plural), 647
g_covariant() (sage.rings.invariants.invariant_theory.BinaryQuartic method), 413
galois_group() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 141
gamma_covariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 421
gauss_on_polys() (in module sage.rings.polynomial.polynomial_pbtor), 637
gcd() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse method), 566
gcd() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 541
gcd() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 271
gcd() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 378
gcd() (sage.rings.polynomial.pbtor[BooleanMonomial method), 598
gcd() (sage.rings.polynomial.polynomial_element.Polynomial method), 49
gcd() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse method), 112
gcd() (sage.rings.polynomial.polynomial_gf2x.Polynomial_template method), 118
gcd() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 126
gcd() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 133
gcd() (sage.rings.polynomial.polynomial_monomial_number_field.Polynomial_monomial_number_field method), 156
gcd() (sage.rings.polynomial.polynomial_monomial_number_field.Polynomial_monomial_number_field method), 121
gcd() (sage.rings.polynomial.polynomial_number_field.Polynomial_absolute_number_field_dense method), 122
gcd() (sage.rings.polynomial.polynomial_number_field.Polynomial_relative_number_field_dense method), 123
gcd() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 227
gen() (sage.rings.fraction_field.FractionField_generic method), 508
gen() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_ring method), 560
gen() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial method), 526
gen() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 369
gen() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 260
gen() (sage.rings.polynomial.pbtor[BooleanMonomialMonoid method), 601
gen() (sage.rings.polynomial.pbtor[BooleanPolynomialRing method), 622
gen() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 648
gen() (sage.rings.polynomial.polynomial_ideal.PolynomialQuotientRing_singular method), 227
gen() (sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_general method), 499

GenDictWithBasing (class in sage.rings.polynomial.infinite_polynomial_ring), 556
generic_power_trunc() (in module sage.rings.polynomial.polynomial_element), 106
gen() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal method), 310
gen() (sage.rings.polynomial.pbtor[BooleanMonomialMonoid method), 601
gen() (sage.rings.polynomial.pbtor[BooleanPolynomialRing method), 623
gen() (sage.rings.polynomial.symmetric_reduction.SymmetricReductionStrategy method), 583
gen() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse method), 560
gen() (sage.rings.polynomial.polynomial_ideal.PolynomialRing_general method), 22
gen() (sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_general method), 499
genus() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 329
get_base_order_code() (sage.rings.polynomial.pbtor[BooleanPolynomialRing method), 623
get_be_log() (sage.rings.polynomial.real_roots.context method), 186
get_cparent() (sage.rings.polynomial.polynomial_gf2x.Polynomial_template method), 119
get_cparent()

get_cparent()

get_dc_log()

get_form()

get_msb_bit()

get_msb_bit()

get_order_code()

get_realfield_rndu()

get_var_mapping()

GF2X_BuildIrred_list()

GF2X_BuildRandomIrred_list()

GF2X_BuildSparseIrred_list()

gpol()

graded_part()

gradient()

gradient()

gradient()

gradient()

greater_tuple

greater_tuple_block()

greater_tuple_deglex()

greater_tuple_degrevlex()

greater_tuple_invlex()

greater_tuple_lex()

greater_tuple_matrix()

greater_tuple_negdeglex()

greater_tuple_negdegrevlex()

greater_tuple_negdeglex()

greater_tuple_negwdeglex()

greater_tuple_negwdegrevlex()

greater_tuple_wdeglex()

groebner_basis()

groebner_basis()

groebner_fan()

GroebnerStrategy

H

h_covariant()
has_any_inverse() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 534
has_constant_part() (sage.rings.polynomial.pbori.BooleanPolynomial method), 604
has_constant_part() (sage.rings.polynomial.pbori.BooleConstant method), 589
has_cyclotomic_factor() (sage.rings.polynomial.polynomial_element.Polynomial method), 51
has_degree_order() (sage.rings.polynomial.pbori.BooleanPolynomialRing method), 623
has_inverse_of() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 535
has_root() (sage.rings.polynomial.real_roots.island method), 197
head_normal_form() (sage.rings.polynomial.pbori.ReductionStrategy method), 635
hensel_lift() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdv method), 108
hensel_lift() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 142
Hessian() (sage.rings.invariants.invariant_theory.TernaryCubic method), 438
hilbert_numerator() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 331
hilbert_poincare_series() (in module sage.rings.polynomial.hilbert), 404
hilbert_polynomial() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 332
hilbert_series() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 333
homogenize() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 272
homogenize() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal method), 316
homogenize() (sage.rings.polynomial.polydict.PolyDict method), 398
homogenized() (sage.rings.invariants.invariant_theory.AlgebraicForm method), 410
homogenized() (sage.rings.invariants.invariant_theory.SeveralAlgebraicForms method), 437
homogeneous_symmetric_function() (in module sage.rings.polynomial.omega), 550

i_covariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 421
id() (sage.rings.polynomial.pbori.BooleanPolynomialRing method), 624
ideal() (sage.rings.polynomial.laurent_polynomial.laurent_polynomial.LaurentPolynomialRing_generic method), 527
ideal() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular method), 370
ideal() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 359
ideal() (sage.rings.polynomial.pbori.BooleanPolynomialRing method), 624
ideal() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 648
Ideal_npoly_field (class in sage.rings.polynomial.ideal), 213
if_then_else() (in module sage.rings.polynomial.pbori), 638
implications() (sage.rings.polynomial.pbori.GroebnerStrategy method), 631
include_divisors() (sage.rings.polynomial.pbori.BooleSet method), 593
increase_precision() (sage.rings.polynomial.real_roots.ocean method), 200
index() (sage.rings.polynomial.pbori.BooleanMonomial method), 598
InfiniteGenDict (class in sage.rings.polynomial.infinite_polynomial_ring), 556
InfinitePolynomial() (in module sage.rings.polynomial.infinite_polynomial_element), 563
InfinitePolynomial_dense (class in sage.rings.polynomial.infinite_polynomial_element), 564
InfinitePolynomial_sparse (class in sage.rings.polynomial.infinite_polynomial_element), 564
InfinitePolynomialGen (class in sage.rings.polynomial.infinite_polynomial_ring), 557
InfinitePolynomialRing_dense (class in sage.rings.polynomial.infinite_polynomial_ring), 557
InfinitePolynomialRing_sparse (class in sage.rings.polynomial.infinite_polynomial_ring), 559
InfinitePolynomialRingFactory (class in sage.rings.polynomial.infinite_polynomial_ring), 557
inhomogeneous_quadratic_form() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 481

Index
int_list() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n method), 155
int_list() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_nntl_zz method), 160
integral() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 541
integral() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 295
integral() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 379
integral() (sage.rings.polynomial.polynomial_element.Polynomial method), 52
integral() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse method), 113
integral() (sage.rings.polynomial.polynomial_real_mpfr_dense.PolynomialRealDense method), 164
integral_closure() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 333
inter_reduction() (in module sage.rings.polynomial.toy_buchberger), 457
interpolate() (in module sage.rings.polynomial.pbori), 638
interpolate_smallest_lex() (in module sage.rings.polynomial.pbori), 639
interpolation_polynomial() (sage.rings.polynomial.pbori.BooleanPolynomialRing method), 624
interreduced_basis() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 334
interreduced_basis() (sage.rings.polynomial.pbori.BooleanPolynomialIdeal method), 619
interreduced_basis() (sage.rings.polynomial.symmetric_ideal.SymmetricIdeal method), 576
intersect() (sage.rings.polynomial.real_roots.BolleSet method), 593
intersection() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 335
interval_bernstein_polynomial (class in sage.rings.polynomial.real_roots), 188
interval_bernstein_polynomial_float (class in sage.rings.polynomial.real_roots), 191
interval_bernstein_polynomial_integer (class in sage.rings.polynomial.real_roots), 192
interval_roots() (in module sage.rings.polynomial.complex_roots), 212
intervals_disjoint() (in module sage.rings.polynomial.complex_roots), 212
intvec_to_doublevec() (in module sage.rings.polynomial.real_roots), 195
invariants() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 422
invariants() (sage.rings.invariants.invariant_theory.QuadraticForm method), 435
InvariantTheoryFactory (class in sage.rings.invariants.invariant_theory), 425
inverse_mod() (sage.rings.polynomial.multi_polynomial_element.MPolynomial method), 274
inverse_mod() (sage.rings.polynomial.polynomial_element.Polynomial method), 54
inverse_of_unit() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 542
inverse_of_unit() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 296
inverse_of_unit() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 380
inverse_of_unit() (sage.rings.polynomial.polynomial_element.Polynomial method), 55
inverse_series_trunc() (sage.rings.polynomial.polynomial_element.Polynomial method), 55
inverse_series_trunc() (sage.rings.polynomial.integer_dense_flint.Polynomial_integer_dense_flint method), 126
inverse_series_trunc() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 143
irreducible_element() (sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_finite_field method), 12
irreducible_element() (sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_mod_p method), 14
irrelevant_ideal() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 260
is_block_order() (sage.rings.polynomial.term_order.TermOrder method), 251
is_commutative() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 649
is_commutative() (sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_general method), 500
is_constant() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 535
is_constant() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 542
is_constant() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 296
is_constant() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 380
is_constant() (sage.rings.polynomial.pbori.BooleanPolynomial method), 604
is_constant() (sage.rings.polynomial.pbori.BooleConstant method), 590
is_constant() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 655
is_constant() (sage.rings.polynomial.polydict.ETuple method), 396
is_constant() (sage.rings.polynomial.multi_polynomial_element.Polynomial method), 56
is_cyclotomic() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 473
is_cyclotomic_product() (sage.rings.polynomial.polynomial_element.Polynomial method), 56
is_eisenstein() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 171
is_equal() (sage.rings.polynomial.pbori.BooleanPolynomial method), 605
is_exact() (sage.rings.fract_field.FractionField_generic method), 509
is_exact() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_general method), 527
is_exact() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 261
is_exact() (sage.rings.polynomial.laurent_polynomial_ring.PolynomialRing_general method), 22
is_exact() (sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_general method), 500
is_field() (sage.rings.fract_field.FractionField_generic method), 509
is_field() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse method), 560
is_field() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_general method), 527
is_field() (sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict_domain method), 290
is_field() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 261
is_field() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 649
is_field() (sage.rings.polynomial.multi_polynomial_ring_quotient.PolynomialQuotientRing_generic method), 227
is_field() (sage.rings.polynomial.laurent_polynomial_ring.PolynomialRing_general method), 22
is_finite() (sage.rings.fract_field.FractionField_generic method), 509
is_finite() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_general method), 527
is_finite() (sage.rings.polynomial.multi_polynomial_ring_quotient.PolynomialQuotientRing_generic method), 227
is_finite() (sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_general method), 500
is_FractionField() (in module sage.rings.fract_field), 510
is_FractionFieldElement() (in module sage.rings.fract_field_element), 514
is_gen() (sage.rings.polynomial.polynomial_element.Polynomial method), 58
is_gen() (sage.rings.polynomial.polynomial_gf2x.Polynomial_template method), 119
is_gen() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_ZZ method), 157
is_gen() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_ZZ method), 160
is_gen() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_template method), 148
is_gen() (sage.rings.polynomial.polynomial_zz_pex.Polynomial_template method), 180
is_generator() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_multipolydict method), 274
is_generator() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 296
is_global() (sage.rings.polynomial.term_order.TermOrder method), 251
is_groebner() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 359
is_homogeneous() (sage.rings.invariants.invariant_theory.FormsBase method), 424
is_homogeneous() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 297
is_homogeneous() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 317
is_homogeneous() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 380
is_homogeneous() (sage.rings.polynomial.pbori.BooleanPolynomial method), 605
is_homogeneous() (sage.rings.polynomial.plural.NCPolynomial_plural method), 655
is_homogeneous() (sage.rings.polynomial.polydict.PolyDict method), 399
is_homogeneous() (sage.rings.polynomial.polynomial_element.Polynomial method), 58
is_injective() (sage.rings.fraction_field.FractionFieldEmbedding method), 506
is_injective() (sage.rings.polynomial.polynomial_element.PolynomialBaseringInjection method), 102
is_injective() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_coercion method), 216
is_injective() (sage.rings.polynomial.polynomial_ring_homomorphism.PolynomialRingHomomorphism_from_base method), 30
is_integral() (sage.rings.fraction_field_element.FractionFieldElement_1poly_field method), 513
is_integral_domain() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse method), 560
is_integral_domain() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic method), 527
is_integral_domain() (sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict_domain method), 290
is_integral_domain() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 261
is_integral_domain() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 23
is_irreducible() (sage.rings.polynomial.polynomial_element.Polynomial method), 58
is_irreducible() (sage.rings.polynomial.polynomial_gf2x.Polynomial_GF2X method), 117
is_irreducible() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 143
is_irreducible() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 151
is_irreducible() (sage.rings.polynomial.polynomial_zz_pex.Polynomial_ZZ_pEX method), 178
is_LaurentPolynomialRing() (in module sage.rings.polynomial.laurent_polynomial_ring), 530
is_left_divisible_by() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 473
is_linearly_dependent() (in module sage.rings.polynomial.toy_variety), 460
is_local() (sage.rings.polynomial.term_order.TermOrder method), 252
is_maximal() (sage.rings.polynomial.symmetric_ideal.SymmetricIdeal method), 577
is_monic() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 474
is_monomial() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 535
is_monomial() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 543
is_monomial() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_libsingular method), 297
is_monomial() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 381
is_monomial() (sage.rings.polynomial.multi_polynomial.NCPolynomial_plural method), 656
is_monomial() (sage.rings.polynomial.polynomial_element.Polynomial method), 59
is_monomial() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 474
is_MPolynomial() (in module sage.rings.polynomial.multi_polynomial), 287
is_MPolynomial() (in module sage.rings.polynomial.multi_polynomial_element), 304
is_MPolynomialIdeal() (in module sage.rings.polynomial.multi_polynomial_ideal), 351
is_MPolynomialRing() (in module sage.rings.polynomial.multi_polynomial_ring_base), 267
is_multiple_of() (sage.rings.polynomial.polydict.ETuple method), 396
is_nilpotent() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomialSparse method), 566
is_nilpotent() (sage.rings.polynomial.multi_polynomial_element.MPolynomial method), 275
is_nilpotent() (sage.rings.polynomial.polynomial_element.MPolynomial method), 60
is_nilpotent() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 475
is_noetherian() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse method), 560
is_noetherian() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic method), 527
is_noetherian() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 261
is_noetherian() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomialRing_generic method), 527
is_one() (sage.rings.fraction_field_element.FractionFieldElement method), 511
is_one() (sage.rings.polynomial.pboration.BooleConstant method), 590
is_one() (sage.rings.polynomial.polynomial_element.Polynomial method), 60
is_one() (sage.rings.polynomial.polynomial_element.Polynomial_template method), 119
is_one() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 127
is_one() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 143
is_one() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_template method), 148
is_one() (sage.rings.polynomial.polynomial_zz_pex.Polynomial_template method), 180
is_pair() (sage.rings.polynomial.pboration.BooleanPolynomial method), 605
is_Polynomial() (in module sage.rings.polynomial.polynomial_element), 106
is_PolynomialQuotientRing() (in module sage.rings.polynomial.polynomial_quotient_ring), 232
is_PolynomialRing() (in module sage.rings.polynomial.polynomial_ring), 28
is_PolynomialSequence() (in module sage.rings.polynomial.multi_polynomial_sequence), 366
is_prime() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_element_Polynomial method), 335
is_primitive() (sage.rings.polynomial.polynomial_element.Polynomial method), 60
is_real_rooted() (sage.rings.polynomial.polynomial_element.Polynomial method), 62
is_right_divisible_by() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 475
is_singleton() (sage.rings.polynomial.pboration.BooleanPolynomial method), 606
is_singleton_or_pair() (sage.rings.polynomial.pboration.BooleanPolynomial method), 607
is_sparse() (sage.rings.polynomial.polynomial_element.Polynomial method), 23
is_sparse() (sage.rings.polynomial.skew_polynomial.SkewPolynomial method), 500
is_square() (sage.rings.fraction_field_element.FractionFieldElement method), 511
is_square() (sage.rings.polynomial.polynomial_element.Polynomial method), 516
is_square() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial method), 535
is_square() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 543
is_square() (sage.rings.polynomial.multi_polynomial_element.MPolynomial method), 275
is_square() (sage.rings.polynomial.polynomial_element.Polynomial method), 62
is_squarefree() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 381
is_squarefree() (sage.rings.polynomial.polynomial_element.Polynomial method), 63
is_surjective() (sage.rings.fraction_field.FractionFieldEmbedding method), 506
is_surjective() (sage.rings.polynomial.polynomial_element.PolynomialBaseringInjection method), 102
is_surjective() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_coercion method), 217
is_surjective() (sage.rings.polynomial.polynomial_ring_homomorphism.PolynomialRingHomomorphism_from_base method), 30
is_term() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 297
is_term() (sage.rings.polynomial.polynomial_element.Polynomial method), 64
is_term() (sage.rings.polynomial.polynomial_element.Polynomial_generic_dense method), 103
is_term() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 476
is_triangular() (in module sage.rings.polynomial.toy_variety), 460
is_unique_factorization_domain() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 23
is_unit() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomialSparse method), 566
is_unit() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 536
is_unit() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 544
is_unit() (sage.rings.polynomial.multi_polynomial.LaurentPolynomial method), 275
is_unit() (sage.rings.polynomial.pbori.BooleanPolynomial method), 607
is_unit() (sage.rings.polynomial.polynomial_element.Polynomial method), 65
is_unit() (sage.rings.polynomial.polynomial_element_generic.PolynomialGenericDomain method), 110
is_unit() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 477
is_univariate() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 536
is_univariate() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 298
is_univariate() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 381
is_univariate() (sage.rings.polynomial.pbori.BooleanPolynomial method), 608
is_zero() (sage.rings.fraction_field_element.FractionFieldElement method), 512
is_zero() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 544
is_zero() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 381
is_zero() (sage.rings.polynomial.pbori.BooleanPolynomial method), 608
is_zero() (sage.rings.polynomial.pbori.BooleConstant method), 590
is_zero() (sage.rings.polynomial.plural.NCPolynomial_plural method), 656
is_zero() (sage.rings.polynomial.polynomial_element.Polynomial method), 66
is_zero() (sage.rings.polynomial.polynomial_gf2x.Polynomial_template method), 119
is_zero() (sage.rings.polynomial.polynomial_integer_dense_flint.PolynomialIntegerDenseFlint method), 127
is_zero() (sage.rings.polynomial.polynomial_rational_flint.PolynomialRationalFlint method), 143
is_zero() (sage.rings.polynomial.polynomial_zmod_flint.PolynomialTemplate method), 180
is_zero() (sage.rings.polynomial.polynomial_zz_pex.PolynomialTemplate method), 180
is_zero() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 477
island (class in sage.rings.polynomial.real_roots), 195
iter() (sage.rings.fraction_field_FpT.FpT method), 515
iterindex() (sage.rings.polynomial.pbori.BooleanMonomial method), 598

J
j_covariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 422
J_covariant() (sage.rings.invariants.invariant_theory.TernaryCubic method), 438
J_covariant() (sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics method), 443
jacobian_ideal() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 276

K
karatsuba_threshold() (sage.rings.polynomial.polynomial_ring.PolynomialRingGeneral method), 23
kbase_libsingular() (in module sage.rings.polynomial.multi_polynomial_ideal_libsingular), 392
krull_dimension() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRingSparse method), 561
krull_dimension() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRingSparse method), 528
krull_dimension() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRingBase method), 261
krull_dimension() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRingGeneral method), 228
krull_dimension() (sage.rings.polynomial.polynomial_ring.PolynomialRingGeneral method), 23
lagrange_polynomial() (sage.rings.polynomial.polynomial_ring.PolynomialRing_field method), 18
lagrange_polynomial() (sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_general method), 501
latex() (sage.rings.polynomial.polydict.PolyDict method), 399
LaurentPolynomial (class in sage.rings.polynomial.laurent_polynomial), 530
LaurentPolynomial_mpair (class in sage.rings.polynomial.laurent_polynomial), 531
LaurentPolynomial_univariate (class in sage.rings.polynomial.laurent_polynomial), 539
LaurentPolynomialRing() (in module sage.rings.polynomial.laurent_polynomial_ring), 523
LaurentPolynomialRing_generic (class in sage.rings.polynomial.laurent_polynomial_ring), 525
LaurentPolynomialRing_mpair (class in sage.rings.polynomial.laurent_polynomial_ring), 529
LaurentPolynomialRing_univariate (class in sage.rings.polynomial.laurent_polynomial_ring), 530
LC() (in module sage.rings.polynomial.toy_d_basis), 463
lc() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse method), 567
c() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 298
c() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 381
c() (sage.rings.polynomial.plural.NCPolynomial_plural method), 656
c() (sage.rings.polynomial.polynomial_element.Polynomial method), 66
LCM() (in module sage.rings.polynomial.toy_buchberger), 456
lcm() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 382
lcm() (sage.rings.polynomial.polynomial_element.Polynomial method), 66
lcm() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 127
lcm() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 134
lcm() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 144
lcmt() (sage.rings.polynomial.polydict.PolyDict method), 399
lead() (sage.rings.polynomial.pbori.BooleanPolynomial method), 608
lead() (sage.rings.polynomial.pbori.PolynomialConstruct method), 634
lead() (sage.rings.polynomial.pbori.PolynomialFactory method), 634
lead_deg() (sage.rings.polynomial.pbori.BooleanPolynomial method), 608
lead_divisors() (sage.rings.polynomial.pbori.BooleanPolynomial method), 609
leading_coefficient() (sage.rings.polynomial.polynomial_element.Polynomial method), 66
leading_coefficient() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 477
left_divides() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 477
left_gcd() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 478
left_lcm() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 479
left_mod() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 480
left_monic() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 480
left_power_mod() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial_generic_dense method), 491
left_quo_rem() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial_generic_dense method), 492
left_xgcd() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 481
less_bits() (sage.rings.polynomial.real_roots.island method), 197
lex_lead() (sage.rings.polynomial.pbori.BooleanPolynomial method), 609
lex_lead_deg() (sage.rings.polynomial.pbori.BooleanPolynomial method), 609
lift() (sage.rings.polynomial.multi_polynomial_element.MPolynomial method), 276
lift() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 298
lift() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 382
lift() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 171

Index 687
lift () (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 228
lift () (sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement method), 235
linear_map (class in sage.rings.polynomial.real_roots), 197
linear_representation() (in module sage.rings.polynomial.toy_variety), 461
list () (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 172
list () (sage.rings.polynomial.polydict.PolyDict method), 399
list () (sage.rings.polynomial.polynomial_element.Polynomial method), 66
list () (sage.rings.polynomial.polynomial_element.Polynomial_generic_dense method), 103
list () (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse method), 113
list () (sage.rings.polynomial.polynomial_gf2x.Polynomial_template method), 119
list () (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 127
list () (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 134
list () (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n method), 155
list () (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n_ZZ method), 158
list () (sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement method), 235
list () (sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement method), 235
list () (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 144
list () (sage.rings.polynomial.polynomial_real_mpfr_dense.PolynomialRealDense method), 164
list () (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_template method), 148
list () (sage.rings.polynomial.polynomial_zz_pex.Polynomial_ZZ_pEX method), 178
list () (sage.rings.polynomial.polynomial_element.Polynomial method), 67
list () (sage.rings.polynomial.plural.NCPolynomial_plural method), 657
ll_red_nf_noredsb() (in module sage.rings.polynomial.pbori), 639
ll_red_nf_noredsb_single_recursive_call() (in module sage.rings.polynomial.pbori), 640
ll_red_nf_redsb() (in module sage.rings.polynomial.pbori), 640
ll_reduce_all() (sage.rings.polynomial.pbori.GroebnerStrategy method), 631
LM() (in module sage.rings.polynomial.toy_buchberger), 456
LM() (in module sage.rings.polynomial.toy_d_basis), 464
lm () (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse method), 567
lm () (sage.rings.polynomial.multi_polynomial_element.MPolynomial_libsingular method), 383
lm () (sage.rings.polynomial.pbori.BooleanPolynomial method), 610
lm () (sage.rings.polynomial.plural.NCPolynomial_plural method), 657
lm () (sage.rings.polynomial.polynomial_element.Polynomial method), 67
lshift_coeffs() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 172
lsign () (sage.rings.polynomial.real_roots.bernstein_polynomial_factory method), 183
LT () (in module sage.rings.polynomial.toy_buchberger), 456
lt () (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse method), 567
lt () (sage.rings.polynomial.multi_polynomial_element.MPolynomial_libsingular method), 383
lt () (sage.rings.polynomial.pbori.BooleanPolynomial method), 610
lt () (sage.rings.polynomial.plural.NCPolynomial_plural method), 657
lt () (sage.rings.polynomial.polynomial_element.Polynomial method), 67
macaulay2_str() (sage.rings.polynomial.term_order.TermOrder method), 252
macaulay_resultant () (sage.rings.polynomial.multi_polynomial.MPolynomial method), 276
macaulay_resultant () (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 261
modular_composition() (sage.rings.polynomial.polynomial_gf2x.Polynomial_GF2X method), 117
modulus() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 228
monic() (sage.rings.polynomial.polynomial_element.Polynomial method), 68
monic() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 151
monics() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 24
monomial() (sage.rings.polynomial.polynomial_quotient_ring_base.MPolynomialRing_quotient_base method), 264
monomial() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 650
monomial() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular method), 370
monomial() (sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict method), 288
monomial() (sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict method), 289
monomial() (sage.rings.polynomial.multi_polynomial_ring.NCPolynomialRing_plural method), 650
monomial() (sage.rings.polynomial.multi_polynomial_ring.NCPolynomialRing_plural method), 651
monomial() (sage.rings.polynomial.pbori.BooleanPolynomial method), 611
monomial() (sage.rings.polynomial.pbori.BooleanPolynomial method), 633
monomial() (sage.rings.polynomial.pbori.BooleanPolynomial method), 634
monomial() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 649
monomial() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 650
monomial() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 651
monomial() (sage.rings.polynomial.polydict.PolyDict method), 400
monomial_pairwise_prime() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular method), 370
monomial_pairwise_prime() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular method), 371
monomial_pairwise_prime() (sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict method), 288
monomial_pairwise_prime() (sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict method), 289
monomial_pairwise_prime() (sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict method), 299
monomial_pairwise_prime() (sage.rings.polynomial.multi_polynomial_ring.NCPolynomialRing_plural method), 652
monomial_quotient() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular method), 370
monomial_quotient() (sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict method), 289
monomial_quotient() (sage.rings.polynomial.multi_polynomial_ring.NCPolynomialRing_plural method), 651
monomial_reduce() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular method), 372
monomial_reduce() (sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict method), 288
monomial_reduce() (sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict method), 289
monomial() (in module sage.rings.polynomial), 407
monomials() (sage.rings.invariants.invariant_theory.BinaryQuartic method), 414
monomials() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 423
monomials() (sage.rings.invariants.invariant_theory.QuadraticForm method), 435
monomials() (sage.rings.invariants.invariant_theory.TernaryCubic method), 439
MonomialConstruct (class in sage.rings.polynomial.pbori), 633
MonomialFactory (class in sage.rings.polynomial.pbori), 633
monomials() (in module sage.rings.polynomial), 407
monomials() (sage.rings.invariants.invariant_theory.TernaryQuadratic method), 441
monomials() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 537
monomials() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 300
monomials() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 384
monomials() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 360
monomials() (sage.rings.polynomial.pbori.BoolePolynomial method), 611
monomials() (sage.rings.polynomial.plural.NCPolynomial_plural method), 658
monomials() (sage.rings.polynomial.polynomial_element.Polynomial method), 70
more_bits() (sage.rings.polynomial.real_roots.island method), 197
MPolynomial (class in sage.rings.polynomial.multi_polynomial), 267
MPolynomial_element (class in sage.rings.polynomial.multi_polynomial_element), 291
MPolynomial_libsingular (class in sage.rings.polynomial.multi_polynomial_libsingular), 372
MPolynomial_polydict (class in sage.rings.polynomial.multi_polynomial_element), 292
MPolynomialRing_base (class in sage.rings.polynomial.multi_polynomial_ring_base), 258
MPolynomialRing_libsingular (class in sage.rings.polynomial.multi_polynomial_libsingular), 368
MPolynomialRing_macaulay2_repr (class in sage.rings.polynomial.multi_polynomial_ideal), 287
MPolynomialRing_polydict (class in sage.rings.polynomial.multi_polynomial_ring), 290
mul_pd (class in sage.rings.polynomial.polynomial_ring_polynomial), 237
mult_fact_sim_C() (in module sage.rings.polynomial.pbori), 641
multi_point_evaluation() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 482
multiples() (sage.rings.polynomial.pbori.BooleMonomial method), 599
multiples_of() (sage.rings.polynomial.pbori.BooleSet method), 593
multiplication_trunc() (sage.rings.polynomial.polynomial_element.Polynomial method), 70

N
n_forms() (sage.rings.invariants.invariant_theory.SeveralAlgebraicForms method), 437
n_nodes() (sage.rings.polynomial.pbori.BoolePolynomial method), 611
n_nodes() (sage.rings.polynomial.pbori.BooleSet method), 594
n_variables() (sage.rings.polynomial.pbori.BoolePolynomialRing method), 625
n_vars() (sage.rings.polynomial.pbori.BoolePolynomial method), 611
name() (sage.rings.polynomial.term_order.TermOrder method), 253
navigation() (sage.rings.polynomial.pbori.BooleMonomial method), 599
navigation() (sage.rings.polynomial.pbori.BoolePolynomial method), 612
navigation() (sage.rings.polynomial.pbori.BooleSet method), 594
NCPolynomial_plural (class in sage.rings.polynomial.plural), 652
NCPolynomialIdeal (class in sage.rings.polynomial.multi_polynomial_ideal), 347
NCPolynomialRing_plural (class in sage.rings.polynomial.plural), 647
new_CRing() (in module sage.rings.polynomial.plural), 660
new_NRing() (in module sage.rings.polynomial.plural), 661
new_Ring() (in module sage.rings.polynomial.plural), 662
newton_polygon() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_sparse.Polynomial_padic_capped_relative_sparse method), 172
newton_polygon() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdv method), 108
newton_polytope() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 279
newton_raphson() (sage.rings.polynomial.polynomial_element.Polynomial method), 70
newton_slopes() (sage.rings.polynomial.polynomial_element_generic.Polynomial_cdv method), 173
newton_slopes() (sage.rings.polynomial.polynomial_element.Polynomial method), 71
next() (sage.rings.fraction_field_FpT.FpTElement method), 516
next() (sage.rings.polynomial.infinite_polynomial_ring.GenDictWithBasering method), 556
next_spoly() (sage.rings.polynomial.pbori.GroebnerStrategy method), 632
nf() (sage.rings.polynomial.pbori.GroebnerStrategy method), 632
nf() (sage.rings.polynomial.pbori.ReductionStrategy method), 635
nf3() (in module sage.rings.polynomial.pbori), 641
ngens() (sage.rings.fraction_field.FractionField_generic method), 509
ngens() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse method), 561
ngens() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic method), 528
ngens() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 336
ngens() (sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_generic method), 502
nmonomials() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 360
Node (class in sage.rings.polynomial.hilbert), 403
nonzero_positions() (sage.rings.polynomial.polydict.ETuple method), 396
nonzero_values() (sage.rings.polynomial.polydict.ETuple method), 396
norm() (sage.rings.polynomial.polynomial_element.Polynomial method), 71
norm() (sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement method), 235
normalisation() (sage.rings.polynomial.symmetric_ideal.SymmetricIdeal method), 577
npairs() (sage.rings.polynomial.pbori.GroebnerStrategy method), 632
npairs() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 360
nth_root() (sage.rings.fraction_field_element.FractionFieldElement method), 512
nth_root() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 279
nth_root() (sage.rings.polynomial.polynomial_element.Polynomial method), 72
ntl_set_directly() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n method), 155
ntl_set_directly() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n method), 160
ntl_ZZ_pX() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n method), 155
number_field() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 229
number_of_real_roots() (sage.rings.polynomial.polynomial_element.Polynomial method), 73
number_of_roots_in_interval() (sage.rings.polynomial.polynomial_element.Polynomial method), 74
number_of_terms() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial method), 531
number_of_terms() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_element method), 537
number_of_terms() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 544
number_of_terms() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_element method), 292
number_of_terms() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 384

Index
number_of_terms() (sage.rings.polynomial.polynomial_element.Polynomial method), 74
number_of_terms() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse method), 113
number_of_terms() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 482
numerator() (sage.rings.fraction_field_FpT.FpTElement method), 517
numerator() (sage.rings.fraction_field_element.FractionFieldElement method), 512
numerator() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 280
numerator() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 385
numerator() (sage.rings.polynomial.polynomial_element.Polynomial method), 75
numerator() (sage.rings.polynomial.polynomial_ring_element.Polynomial method), 208
numerator() (sage.rings.polynomial.polynomial_quotient_ring.QPolynomialQuotientRing_generic method), 350
numerator() (sage.rings.polynomial.polydict.PolyDict method), 400
numerator() (sage.rings.polynomial.real_roots.FpTElement method), 517
padded_list() (sage.rings.polynomial.polynomial_element.Polynomial method), 76
parallel_reduce() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 483
parameter() (sage.rings.polynomial.infinite_polynomial_element.Polynomial method), 25
parameter() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 502
part() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 360
partition() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse method), 561
parts() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 360
Phi_invariant() (sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics method), 444
plot() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_element method), 76
plot() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_libsingular method), 386
polar_conic() (sage.rings.invariants.invariant_theory.TernaryCubic method), 439
poly_repr() (sage.rings.polynomial.polydict.PolyDict method), 400
PolyDict (class in sage.rings.polynomial.polydict), 397
polygen() (sage.rings.polynomial.multi_polynomial_ring_element), 28
polygens() (sage.rings.polynomial.multi_polynomial_element), 29
Polynomial (class in sage.rings.polynomial.polynomial_element), 31
polynomial() (sage.rings.invariants.invariant_theory.AlgebraicForm method), 411
polynomial() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomialSparse method), 567
polynomial() (sage.rings.polynomial.multi_polynomial_element.MPolynomial method), 280
polynomial() (sage.rings.polynomial.polynomial_element.Polynomial method), 77
Polynomial_absolute_number_field_dense (class in sage.rings.polynomial.polynomial_number_field), 121
polynomial_coefficient() (sage.rings.polynomial.polydict.PolyDict method), 401
polynomial_construction() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate
method), 545
polynomial_default_category() (in module sage.rings.polynomial.polynomial_ring_constructor), 7
Polynomial_dense_mod_n (class in sage.rings.polynomial.polynomial_modn_dense_ntl), 154
Polynomial_dense_mod_p (class in sage.rings.polynomial.polynomial_modn_dense_ntl), 156
Polynomial_dense_modn_ntl_ZZ (class in sage.rings.polynomial.polynomial_modn_dense_ntl), 157
Polynomial_dense_modn_ntl_zz (class in sage.rings.polynomial.polynomial_modn_dense_ntl), 159
Polynomial_generic_cdv (class in sage.rings.polynomial.polynomial_element_generic), 107
Polynomial_generic_cdvf (class in sage.rings.polynomial.polynomial_element_generic), 110
Polynomial_generic_cdv (class in sage.rings.polynomial.polynomial_element_generic), 110
Polynomial_generic_dense (class in sage.rings.polynomial.polynomial_element), 102
Polynomial_generic_dense_cdv (class in sage.rings.polynomial.polynomial_element_generic), 110
Polynomial_generic_dense_cdvf (class in sage.rings.polynomial.polynomial_element_generic), 110
Polynomial_generic_dense_cdvr (class in sage.rings.polynomial.polynomial_element_generic), 110
Polynomial_generic_dense_field (class in sage.rings.polynomial.polynomial_element_generic), 110
Polynomial_generic_dense_inexact (class in sage.rings.polynomial.polynomial_element), 104
Polynomial_generic_domain (class in sage.rings.polynomial.polynomial_element_generic), 110
Polynomial_generic_field (class in sage.rings.polynomial.polynomial_element_generic), 110
Polynomial_generic_sparse (class in sage.rings.polynomial.polynomial_element_generic), 111
Polynomial_generic_sparse_cdv (class in sage.rings.polynomial.polynomial_element_generic), 115
Polynomial_generic_sparse_cdvf (class in sage.rings.polynomial.polynomial_element_generic), 115
Polynomial_generic_sparse_cdvr (class in sage.rings.polynomial.polynomial_element_generic), 116
Polynomial_generic_sparse_field (class in sage.rings.polynomial.polynomial_element_generic), 116
Polynomial_GF2X (class in sage.rings.polynomial.polynomial_gf2x), 117
Polynomial_integer_dense_flint (class in sage.rings.polynomial.polynomial_integer_dense_flint), 123
Polynomial_integer_dense_ntl (class in sage.rings.polynomial.polynomial_integer_dense_flint), 131
Polynomial_padic (class in sage.rings.polynomial.padics.polynomial_padic), 167
Polynomial_padic_capped_relative_dense (class in sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense), 170
Polynomial_padic_flat (class in sage.rings.polynomial.padics.polynomial_padic_flat), 177
Polynomial_rational_flint (class in sage.rings.polynomial.polynomial_rational_flint), 136
Polynomial_relative_number_field_dense (class in sage.rings.polynomial.polynomial_number_field), 122
polynomial_ring() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_dense
method), 558
polynomial_ring() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomialRing_generic
method), 528
polynomial_ring() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic
method), 229
Polynomial_singular_repr (class in sage.rings.polynomial.polynomial_singular_interface), 167
Polynomial_template (class in sage.rings.polynomial.polynomial_gf2x), 118
Polynomial_template (class in sage.rings.polynomial.polynomial_zmod_flint), 147
Polynomial_template (class in sage.rings.polynomial.polynomial_zz_pex), 179
Polynomial_zmod_flint (class in sage.rings.polynomial.polynomial_zmod_flint), 149
Polynomial_ZZ_pEX (class in sage.rings.polynomial.polynomial_zz_pex), 178
Polynomial_ZZ_pX (class in sage.rings.polynomial.polynomial_zz_pex), 179
PolynomialBaseringInjection (class in sage.rings.polynomial.polynomial_element), 101
PolynomialConstruct (class in sage.rings.polynomial.pbori), 633
PolynomialFactory (class in sage.rings.polynomial.pbori), 634
PolynomialQuotientRing_coercion (class in sage.rings.polynomial.polynomial_quotient_ring), 216
PolynomialQuotientRing_domain (class in sage.rings.polynomial.polynomial_quotient_ring), 217
PolynomialQuotientRing_field (class in sage.rings.polynomial.polynomial_quotient_ring), 219
PolynomialQuotientRing_generic (class in sage.rings.polynomial.polynomial_quotient_ring), 220
PolynomialQuotientRingElement (class in sage.rings.polynomial.polynomial_quotient_ring_element), 233
PolynomialQuotientRingFactory (class in sage.rings.polynomial.polynomial_quotient_ring), 214
PolynomialRing (class in module sage.rings.polynomial.polynomial_ring_constructor), 2
PolynomialRing_cdvf (class in sage.rings.polynomial.polynomial_ring), 11
PolynomialRing_cdvr (class in sage.rings.polynomial.polynomial_ring), 11
PolynomialRing_commutative (class in sage.rings.polynomial.polynomial_ring), 11
PolynomialRing_dense_finite_field (class in sage.rings.polynomial.polynomial_ring), 12
PolynomialRing_dense_mod_n (class in sage.rings.polynomial.polynomial_ring), 13
PolynomialRing_dense_mod_p (class in sage.rings.polynomial.polynomial_ring), 14
PolynomialRing_dense_padic_field_capped_relative (class in sage.rings.polynomial.polynomial_ring), 15
PolynomialRing_dense_padic_field_generic (class in sage.rings.polynomial.polynomial_ring), 15
PolynomialRing_dense_padic_ring_capped_absolute (class in sage.rings.polynomial.polynomial_ring), 16
PolynomialRing_dense_padic_ring_capped_relative (class in sage.rings.polynomial.polynomial_ring), 16
PolynomialRing_dense_padic_ring_fixed_mod (class in sage.rings.polynomial.polynomial_ring), 16
PolynomialRing_dense_padic_ring_generic (class in sage.rings.polynomial.polynomial_ring), 16
PolynomialRing_field (class in sage.rings.polynomial.polynomial_ring), 16
PolynomialRing_general (class in sage.rings.polynomial.polynomial_ring), 19
PolynomialRing_integral_domain (class in sage.rings.polynomial.polynomial_ring), 27
PolynomialRing_singular_repr (class in sage.rings.polynomial.polynomial_singular_interface), 167
PolynomialRingHomomorphism_from_base (class in sage.rings.polynomial.polynomial_ring_homomorphism), 29
polynomials () (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 25
PolynomialSequence () (in module sage.rings.polynomial.multi_polynomial_sequence), 355
PolynomialSequence_generic (class in sage.rings.polynomial.multi_polynomial_sequence), 356
PolynomialSequence_gf2 (class in sage.rings.polynomial.multi_polynomial_sequence), 363
PolynomialSequence_gf2e (class in sage.rings.polynomial.multi_polynomial_sequence), 365
Polyring_FpT_coerce (class in sage.rings.fraction_field_FpT), 521
pow_pd (class in sage.rings.polynomial.polynomial_compiled), 237
power_trunc () (sage.rings.polynomial.polynomial_element.Polynomial method), 77
prec () (sage.rings.polynomial.polynomial_element.Polynomial method), 78
prec () (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 484
prec_degree () (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 173
prec_degree () (sage.rings.polynomial.polynomial_element.Polynomial_generic_dense_inexact method), 105
precision_absolute () (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 174
precision_relative () (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 174
PrecisionError, 182
precompute_degree_reduction_cache () (in module sage.rings.polynomial.real_roots), 202
primary_decomposition() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 337
primary_decomposition_complete() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 338
pseudo_divrem() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 128
pseudo_quo_rem() (sage.rings.polynomial.polynomial_element.Polynomial method), 78
pseudooinverse() (in module sage.rings.polynomial.real_roots), 202

Q
quadratic_form() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 429
QuadraticForm (class in sage.rings.invariants.invariant_theory), 433
quaternary_biquadratic() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 430
quaternary_quadratic() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 430
quo_rem() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 545
quo_rem() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_libsingular method), 386
quo_rem() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_field method), 114
quo_rem() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse method), 114
quo_rem() (sage.rings.polynomial.polynomial_gf2x.Polynomial_template method), 119
quo_rem() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 128
quo_rem() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_nmod method), 134
quo_rem() (sage.rings.polynomial.polynomial_modn_dense_flint.Polynomial_modn_dense_flint method), 155
quo_rem() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_nmod method), 158
quo_rem() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_nmod_nZZ method), 160
quo_rem() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_nmod_nntl method), 164
quo_rem() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_nmod_zz method), 168
quo_rem() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_nmod_zz_n method), 172
quo_rem() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_nmod_zz_p method), 176

R
R_invariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 416
radical() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 340
radical() (sage.rings.polynomial.multi_polynomial_element.Polynomial method), 78
random_element() (sage.rings.fraction_field.FractionField_generic method), 509
random_element() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic method), 528
random_element() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal method), 318
random_element() (sage.rings.polynomial.multi_polynomial_quotient_ring.PolynomialQuotientRing_generic method), 229
random_element() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 26
random_element() (sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_general method), 503
random_set() (in module sage.rings.polynomial.pbori), 641
rational_reconstruct() (sage.rings.polynomial.polynomial_element.Polynomial method), 79
rational_reconstruct() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 151
rational_root_bounds() (in module sage.rings.polynomial.real_roots), 202
real_root_intervals() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 129
real_root_intervals() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 135
real_root_intervals() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 145
real_roots() (in module sage.rings.polynomial.real_roots), 202
real_root_intervals() (sage.rings.polynomial.polynomial_element.Polynomial method), 82
reciprocal_transform() (sage.rings.polynomial.polynomial_element.Polynomial method), 82
recursively_insert() (in module sage.rings.polynomial.pbori), 641
red_tail() (in module sage.rings.polynomial.pbori), 641
reduce() (sage.rings.fraction_field_element.FractionFieldElement method), 513
reduce() (sage.rings.fraction_field_element.FractionFieldElement_lpoly_field method), 514
reduce() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse method), 568
reduce() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 301
reduce() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomial_ideal method), 319
reduce() (sage.rings.polynomial.multi_polynomial_ideal.NCPolynomialIdeal method), 348
reduce() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 386
reduce() (sage.rings.polynomial.pbori.BooleanPolynomial method), 612
reduce() (sage.rings.polynomial.pbori.BooleanPolynomialIdeal method), 619
reduce() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 659
reduce() (sage.rings.polynomial.symmetric_ideal.SymmetricIdeal method), 578
reduce() (sage.rings.polynomial.symmetric_reduction.SymmetricReductionStrategy method), 583
reduce() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 361
reduce() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_gf2 method), 364
reduce_form() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 281
reduce_form() (sage.rings.polynomial.pbori.ReductionStrategy method), 636
reducible_by() (sage.rings.polynomial.pbori.BooleanMonomial method), 599
reducible_by() (sage.rings.polynomial.pbori.BooleanPolynomial method), 613
reduction_strategy (sage.rings.polynomial.pbori.GroebnerStrategy attribute), 632
ReductionStrategy (class in sage.rings.polynomial.pbori), 634
refine() (sage.rings.polynomial.real_roots.island method), 197
refine_all() (sage.rings.polynomial.real_roots.ocean method), 201
refine_recurse() (sage.rings.polynomial.real_roots.island method), 197
refine_root() (in module sage.rings.polynomial.refine_root), 212
region() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 190
region() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 207
region_width() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 190
relations() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 652
relative_bounds() (in module sage.rings.polynomial.real_roots), 206
remove_var() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic method), 528
remove_var() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 265
remove_var() (sage.rings.polynomial.pbori.BooleanPolynomialRing method), 626
repr_long() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 266
require_field (in module sage.rings.polynomial.multi_polynomial_ideal), 352
RequireField (class in sage.rings.polynomial.multi_polynomial_ideal), 351
res() (sage.rings.polynomial.multi_polynomial_ideal.NCPolynomialIdeal method), 348
rescale() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 175
reset() (sage.rings.polynomial.symmetric_reduction.SymmetricReductionStrategy method), 584
reset_root_width() (sage.rings.polynomial.real_roots.island method), 197
reset_root_width() (sage.rings.polynomial.real_roots.ocean method), 201
residue() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial method), 545
residue_class_degree() (sage.rings.polynomial.ideal.Ideal_1poly_field method), 213
residue_field() (sage.rings.polynomial.ideal.Ideal_1poly_field method), 214
residue_field() (sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_mod_n method), 13
resultant() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 302
resultant() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 387
resultant() (sage.rings.polynomial.polynomial_element.Polynomial method), 82
resultant() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 129
resultant() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 135
resultant() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 157
resultant() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 145
resultant() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 152
resultant() (sage.rings.polynomial.polynomial_zz_pex.Polynomial_ZZ_pex method), 179
retract() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 230
reverse() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 175
reverse() (sage.rings.polynomial.polynomial_element.Polynomial method), 83
reverse() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse method), 114
reverse() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 129
reverse() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 158
reverse() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl_ZZ method), 160
revert_series() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 493
revert_series() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial_generic_dense method), 493
revert_series() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial_generic_dense method), 493
revert_series() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 488
revert_series() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 487
revert_series() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 486
revert_series() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 485
revert_series() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 484
right_divides() (sage.rings.polynomial.polydict.PolyDict method), 401
right_divides() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 484
right_gcd() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 485
right_lcm() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 486
right_mod() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 487
right_monic() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 487
right_power_mod() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 493
right_quo_rem() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 493
right_xgcd() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 493
ring() (sage.rings.fraction_field.FractionField_generic method), 510
ring() (sage.rings.invariants.invariant_theory.FormsBase method), 425
ring() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse method), 569
ring() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 362
ring() (sage.rings.polynomial.pbori.BooleanMonomial method), 600
ring() (sage.rings.polynomial.pbori.BooleanPolynomial method), 613
ring() (sage.rings.polynomial.pbori.BooleSet method), 594
ring() (sage.rings.fraction_field.FractionField_1poly_field method), 507
root_bounds() (in module sage.rings.polynomial.real_roots), 206
root_field() (sage.rings.polynomial.padics.polynomial_padic.Polynomial_padic method), 170
roots() (sage.rings.polynomial.polynomial_element.Polynomial method), 84
roots() (sage.rings.polynomial.polynomial_element.Polynomial method), 84
roots() (sage.rings.polynomial.real_roots.ocean method), 201
rr_gap (class in sage.rings.polynomial.real_roots), 207
rshift_coeffs() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 175

S
S_class_group() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 220
S_invariant() (sage.rings.invariants.invariant_theory.TernaryCubic method), 438
S_units() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 222
sage.rings.fraction_field (module), 505
sage.rings.fraction_field_element (module), 510
sage.rings.fraction_field_FpT (module), 515
sage.rings.invariants.invariant_theory (module), 407
sage.rings.invariants.reconstruction (module), 450
sage.rings.monomials (module), 407
sage.rings.polynomial.complex_roots (module), 210
sage.rings.polynomial.convolution (module), 237
sage.rings.polynomial.cyclotomic (module), 238
sage.rings.polynomial.flatten (module), 404
sage.rings.polynomial.hilbert (module), 403
sage.rings.polynomial.ideal (module), 213
sage.rings.polynomial.infinite_polynomial_element (module), 562
sage.rings.polynomial.infinite_polynomial_ring (module), 553
sage.rings.polynomial.laurent_polynomial (module), 530
sage.rings.polynomial.laurent_polynomial_element (module), 523
sage.rings.polynomial.multi_polynomial (module), 267
sage.rings.polynomial.multi_polynomial_element (module), 290
sage.rings.polynomial.multi_polynomial_ideal (module), 305
sage.rings.polynomial.multi_polynomial_ideal_libsingular (module), 391
sage.rings.polynomial.multi_polynomial_libsingular (module), 366
sage.rings.polynomial.multi_polynomial_ring (module), 287
sage.rings.polynomial.multi_polynomial_ring_base (module), 258
sage.rings.polynomial.multi_polynomial_sequence (module), 352
sage.rings.polynomial.omega (module), 547
sage.rings.polynomial.padics.polynomial_padic (module), 167
sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense (module), 170
sage.rings.polynomial.padics.polynomial_padic_flat (module), 177
sage.rings.polynomial.pbori (module), 587
sage.rings.polynomial.plural (module), 645
sage.rings.polynomial.polydict (module), 393
sage.rings.polynomial.polynomial_compiled (module), 236
sage.rings.polynomial.polynomial_element (module), 30
sage.rings.polynomial.polynomial_element_generic (module), 107
sage.rings.polynomial.polynomial_fateman (module), 237
sage.rings.polynomial.polynomial_gf2x (module), 116
sage.rings.polynomial.polynomial_integer_dense_flint (module), 123
sage.rings.polynomial.polynomial_integer_dense_ntl (module), 131
sage.rings.polynomial.polynomial_modn_dense_ntl (module), 154
sage.rings.polynomial.polynomial_number_field (module), 120
sage.rings.polynomial.polynomial_quotient_ring (module), 214
sage.rings.polynomial.polynomial_quotient_ring_element (module), 232
sage.rings.polynomial.polynomial_real_mpfr_dense (module), 164
sage.rings.polynomial.polynomial_ring (module), 9
sage.rings.polynomial.polynomial_ring_constructor (module), 1
sage.rings.polynomial.polynomial_ring_homomorphism (module), 29
sage.rings.polynomial.polynomial_singular_interface (module), 166
sage.rings.polynomial.polynomial_zmod_flint (module), 147
sage.rings.polynomial.polynomial_zz_pex (module), 178
sage.rings.polynomial.real_roots (module), 181
sage.rings.polynomial.refine_root (module), 212
sage.rings.polynomial.skew_polynomial_element (module), 467
sage.rings.polynomial.skew_polynomial_ring (module), 497
sage.rings.polynomial.skew_polynomial_ring_constructor (module), 495
sage.rings.polynomial.symmetric_ideal (module), 571
sage.rings.polynomial.symmetric_reduction (module), 580
sage.rings.polynomial.term_order (module), 241
sage.rings.polynomial.toy_buchberger (module), 453
sage.rings.polynomial.toy_d_basis (module), 462
sage.rings.polynomial.toy_variety (module), 458
saturation() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 341
SCA() (in module sage.rings.polynomial.plural), 660
scalar_lmult() (sage.rings.polynomial.polydict.PolyDict method), 401
scalar_rmult() (sage.rings.polynomial.polydict.PolyDict method), 402
scale_intvec_var() (in module sage.rings.polynomial.real_roots), 207
scaled_coeffs() (sage.rings.invariants.invariant_theory.BinaryQuartic method), 414
scaled_coeffs() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 423
scaled_coeffs() (sage.rings.invariants.invariant_theory.QuadraticForm method), 436
scaled_coeffs() (sage.rings.invariants.invariant_theory.TernaryCubic method), 440
scaled_coeffs() (sage.rings.invariants.invariant_theory.TernaryQuadratic method), 442
second() (sage.rings.invariants.invariant_theory.TwoAlgebraicForms method), 442
section() (sage.rings.fraction_field.FractionFieldEmbedding method), 506
section() (sage.rings.fraction_field_FpT.Fp_FpT_coerce method), 520
section() (sage.rings.fraction_field_FpT.Polyring_FpT_coerce method), 521
section() (sage.rings.fraction_field_FpT.ZZ_FpT_coerce method), 522
section() (sage.rings.polynomial.flatten.FlatteningMorphism method), 405
section() (sage.rings.polynomial.polynomial_element.PolynomialBaseringInjection method), 102
section() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomialBaseringInjection method), 490
select() (in module sage.rings.polynomial.toy_buchberger), 457
select() (in module sage.rings.polynomial.toy_d_basis), 464
select() (sage.rings.polynomial.pboring.GroebnerStrategy method), 632
selmer_group() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 230
set() (sage.rings.polynomial.pboring.BooleanMonomial method), 600
set() (sage.rings.polynomial.pboring.BooleanPolynomial method), 613
set() (sage.rings.polynomial.pboring.BooleSet method), 595
set_karatsuba_threshold() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 27
set_random_seed() (in module sage.rings.polynomial.pboring), 641
setgens() (sage.rings.polynomial.symmetric_reduction.SymmetricReductionStrategy method), 584
SeveralAlgebraicForms (class in sage.rings.invariants.invariant_theory), 436
shift() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 545
shift() (sage.rings.polynomial.polynomial_element.Polynomial method), 93
shift() (sage.rings.polynomial.polynomial_element.Polynomial_generic_dense method), 104
shift() (sage.rings.polynomial.polynomial_element.Generic.Polynomial_generic_sparse method), 114
shift() (sage.rings.polynomial.polynomial_element.gf2x.Polynomial_template method), 119
shift() (sage.rings.polynomial.polynomial_element.modn_dense_ntl.Polynomial_dens_mod_n method), 155
shift() (sage.rings.polynomial.polynomial_element_modn_dense_ntl.Polynomial_dense_mod_n method), 158
shift() (sage.rings.polynomial.polynomial_element_modn_dense_ntl.Polynomial_dense_mod_n ZZ method), 161
shift() (sage.rings.polynomial.polynomial_element_real_mpfr_dense.PolynomialRealDense method), 165
shift() (sage.rings.polynomial.polynomial_element_zmod_flint.Polynomial_template method), 149
shift() (sage.rings.polynomial.polynomial_element_zz_pex.Polynomial_template method), 181
shift() (sage.rings.polynomial.polynomial_element_zz_pex.Polynomial_ZZ_pEX method), 179
shrink_bp() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial_float method), 197
singular_moreblocks() (sage.rings.polynomial.term_order.TermOrder method), 253
singular_str() (sage.rings.polynomial.term_order.TermOrder method), 253
size_double() (sage.rings.polynomial.pboring.BooleSet method), 595
SkewPolynomial (class in sage.rings.polynomial.skew_polynomial_element), 467
SkewPolynomial_generic_dense (class in sage.rings.polynomial.skew_polynomial_element), 490
SkewPolynomialBaseringInjection (class in sage.rings.polynomial.skew_polynomial_element), 490
SkewPolynomialRing() (in module sage.rings.polynomial.skew_polynomial_ring_constructor), 495
SkewPolynomialRing_general (class in sage.rings.polynomial.skew_polynomial_ring), 497
slimgb_libsingular() (in module sage.rings.polynomial.multi_polynomial_ideal_libsingular), 392
slope_factorization() (sage.rings.polynomial.polynomial_element.Generic.Polynomial_generic_cdv method), 109
slope_range() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial_float method), 192
slope_range() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial_integer method), 195
small_roots() (in module sage.rings.polynomial.polynomial_modn_dense_ntl), 161
small_roots() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n method), 156
small_roots() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 153
small_spols_in_next_degree() (sage.rings.polynomial.pboring.GroebnerStrategy method), 633
solve() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_gf2 method), 364
some_elements() (sage.rings.fraction_field.FractionField_generic method), 510
some_elements() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 27
some_spols_in_next_degree() (sage.rings.polynomial.pboring.GroebnerStrategy method), 633
sortkey (sage.rings.polynomial.term_order.TermOrder method), 254
sortkey_block() (sage.rings.polynomial.term_order.TermOrder method), 254

Index 701
sortkey_deglex() (sage.rings.polynomial.term_order.TermOrder method), 254
sortkey_degneglex() (sage.rings.polynomial.term_order.TermOrder method), 254
sortkey_degrevlex() (sage.rings.polynomial.term_order.TermOrder method), 255
sortkey_invlex() (sage.rings.polynomial.term_order.TermOrder method), 255
sortkey_matrix() (sage.rings.polynomial.term_order.TermOrder method), 255
sortkey_negdeglex() (sage.rings.polynomial.term_order.TermOrder method), 256
sortkey_negdegrevlex() (sage.rings.polynomial.term_order.TermOrder method), 256
sortkey_neglex() (sage.rings.polynomial.term_order.TermOrder method), 256
sortkey_negwdeglex() (sage.rings.polynomial.term_order.TermOrder method), 256
sortkey_negwdegrevlex() (sage.rings.polynomial.term_order.TermOrder method), 257
sortkey_wdeglex() (sage.rings.polynomial.term_order.TermOrder method), 257
sortkey_wdegrevlex() (sage.rings.polynomial.term_orderTERMOrder method), 257
sparse_iter() (sage.rings.polynomial.polydict.ETuple method), 397
specialization() (sage.rings.fraction_field_element.FractionFieldElement method), 513
specialization() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 284
specialization() (sage.rings.polynomial.polynomial_element.Polynomial method), 93
SpecializationMorphism (class in sage.rings.polynomial.flatten), 406
split_for_targets() (in module sage.rings.polynomial.real_roots), 207
splitting_field() (sage.rings.polynomial.polynomial_element.Polynomial method), 94
spol() (in module sage.rings.polynomial.toy_buchberger), 458
spol() (in module sage.rings.polynomial.toy_d_basis), 465
spoly() (sage.rings.polynomial.pbori.BooleanPolynomial method), 614
sqr_pd (class in sage.rings.polynomial.polynomial_compiled), 237
sqrt() (sage.rings.fraction_field_element.FpT.FpTElement method), 517
square() (sage.rings.polynomial.polynomial_element.Polynomial method), 95
square() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 489
squarefree_decomposition() (sage.rings.polynomial.polynomial_element.Polynomial method), 96
squarefree_decomposition() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 130
squarefree_decomposition() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 135
squarefree_decomposition() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 154
squeezed() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomialElement.InfinitePolynomial_sparse method), 569
squeezed() (sage.rings.polynomial.symmetric_ideal.SymmetricIdeal method), 578
stable_hash() (sage.rings.polynomial.pbori.BooleanMonomial method), 600
stable_hash() (sage.rings.polynomial.pbori.BooleanPolynomial method), 614
stable_hash() (sage.rings.polynomial.pbori.BooleSet method), 595
std() (sage.rings.polynomial.multi_polynomial_element.NCPolynomialIdeal method), 349
std_libsingular() (in module sage.rings.polynomial.libsingular), 392
stretch() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_element.InfinitePolynomial_sparse method), 569
sub_m_mul_q() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 388
subs() (sage.rings.fraction_field_element.FpT.FpTElement method), 518
subs() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 538
subs() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 302
subs() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_idealIdeal method), 319
subs() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 388
subs() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 362
subs() (sage.rings.polynomial.pbori.BooleanPolynomial method), 614
subs() (sage.rings.polynomial.polynomial_element.Polynomial method), 96
subsample_vec_doctest() (in module sage.rings.polynomial.real_root), 208
subset0() (sage.rings.polynomial.pbori.BooleSet method), 595
subset1() (sage.rings.polynomial.pbori.BooleSet method), 596
substitute() (sage.rings.polynomial.polynomial_element.Polynomial method), 96
substitute_variables() (module sage.rings.polynomial.pbori), 642
suggest_plugin_variable() (sage.rings.polynomial.pbori.GroebnerStrategy method), 633
support() (sage.rings.fraction_field_element.FractionFieldElement_poly_field method), 514
sylvester_matrix() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 285
sylvester_matrix() (sage.rings.polynomial.polynomial_element.Polynomial method), 97
symmetric_basis() (sage.rings.polynomial.symmetricIdeal.SymmetricIdeal method), 579
symmetric_cancellation_order() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomialSparse method), 570
symmetric_power() (sage.rings.polynomial.polynomial_element.Polynomial method), 98
SymmetricIdeal (class in sage.rings.polynomial.symmetricIdeal), 571
SymmetricReductionStrategy (class in sage.rings.polynomial.symmetric_reduction), 581
symmetrisation() (sage.rings.polynomial.symmetricIdeal.SymmetricIdeal method), 579
symmGB_F2() (sage.rings.polynomial.pbori.GroebnerStrategy method), 633
syzygy() (sage.rings.invariants.invariant_theory.TernaryCubic method), 440
syzygy() (sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics method), 445
syzygy() (sage.rings.invariants.invariant_theory.TwoTernaryQuadratics method), 448
syzygy_module() (sage.rings.polynomial.multi_polynomialIdeal.MPolynomialIdeal_singular_base_repr method), 323
syzygy_module() (sage.rings.polynomial.multi_polynomialIdeal.MPolynomialIdeal_singular_repr method), 341
syzygy_module() (sage.rings.polynomial.multi_polynomialIdeal.NCPolynomialIdeal method), 350
T
t_c covariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 416
t_covariant() (sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics method), 444
t_invariant() (sage.rings.invariants.invariant_theory.TernaryCubic method), 438
t_prime covariant() (sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics method), 444
tail() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse method), 570
tailreduce() (sage.rings.polynomial.symmetric_reduction.SymmetricReductionStrategy method), 585
tau_covariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 423
taylor_shift1_intvec() (module sage.rings.polynomial.real_root), 208
tensor_with_ring() (sage.rings.polynomial.infinite_polynomial_ring_infinite_polynomial_element, 558
TensorWithRing() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_dense method), 561
term_lmult() (sage.rings.polynomial.polydict.PolyDict method), 402
term_order() (sage.rings.polynomial.laurent_polynomial_element.LaurentPolynomialRing_generic method), 528
term_order() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 266
term_order() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 652
term_rmult() (sage.rings.polynomial.polydict.PolyDict method), 402
terminal one() (sage.rings.polynomial.pbori.CCuddNavigator method), 628
termOrd (class in sage.rings.polynomial.term_order), 246
TermOrder() (module sage.rings.polynomial.pbori.CCuddNavigator method), 636
termorder_from_singular() (module sage.rings.polynomial.term_order), 258
terms() (sage.rings.polynomial.pbori.BooleanPolynomial method), 615
ternary_biquadratic() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 431
ternary_cubic() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 432
ternary_quadratic() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 432
ternary Cubic (class in sage.rings.invariants.invariant_theory), 437
TernaryQuadratic (class in sage.rings.invariants.invariant_theory), 441
then_branch() (sage.rings.polynomial.pbori.CCuddNavigator method), 628
theta_covariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 424
Theta_covariant() (sage.rings.invariants.invariant_theory.TernaryCubic method), 439
Theta_invariant() (sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics method), 445
Theta_invariant() (sage.rings.invariants.invariant_theory.TwoTernaryQuadratics method), 447
Theta_prime_invariant() (sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics method), 445
Theta_prime_invariant() (sage.rings.invariants.invariant_theory.TwoTernaryQuadratics method), 447
to_bernstein() (in module sage.rings.polynomial.real_roots), 208
to_bernstein_warp() (in module sage.rings.polynomial.real_roots), 209
to_ocean() (sage.rings.polynomial.real_roots.linear_map method), 197
to_ocean() (sage.rings.polynomial.real_roots.warp_map method), 209
top_index() (in module sage.rings.polynomial.pbori), 642
top_sugar() (sage.rings.polynomial.pbori.GroebnerStrategy method), 633
total_degree() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_libdict method), 303
total_degree() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 389
total_degree() (sage.rings.polynomial.pbori.BooleanPolynomial method), 615
total_degree() (sage.rings.polynomial.plural.NCPolynomial_plural method), 659
total_degree() (sage.rings.polynomial.polydict.PolyDict method), 403
trace() (sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement method), 236
trace_polynomial() (sage.rings.polynomial.polynomial_element.Polynomial method), 98
transformed() (sage.rings.invariants.invariant_theory.AlgebraicForm method), 411
transformed_basis() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 341
transvectant() (in module sage.rings.invariants.invariant_theory), 448
triangular_decomposition() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 342
triangular_factorization() (in module sage.rings.polynomial.toy_variety), 461
truncate() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 546
truncate() (sage.rings.polynomial.multi_polynomial_element.MPolynomial method), 286
truncate() (sage.rings.polynomial.polynomial_element.Polynomial method), 99
truncate() (sage.rings.polynomial.polynomial_generic_dense method), 104
truncate() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse method), 115
truncate() (sage.rings.polynomial.polynomial_gf2x.Polynomial_template method), 120
truncate() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_ZZ method), 158
truncate() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_ZZ method), 161
truncate() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 146
truncate() (sage.rings.polynomial.polynomial_real_mpfr_dense.PolynomialRealDense method), 166
truncate() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_template method), 149
truncate() (sage.rings.polynomial.polynomial_zz_pex.Polynomial_template method), 181
truncate() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial_generic_dense method), 494
truncate_abs() (sage.rings.polynomial.skew_polynomial_element.PolynomialRealDense method), 166
try_rand_split() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 190
try_split() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 190
tuple_weight() (sage.rings.polynomial.term_order.TermOrder method), 257
twist_map() (sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_general method), 504
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Source</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwoAlgebraicForms</td>
<td><code>class in sage.rings.invariants.invariant_theory</code></td>
<td>442</td>
</tr>
<tr>
<td>TwoQuaternaryQuadratics</td>
<td><code>class in sage.rings.invariants.invariant_theory</code></td>
<td>443</td>
</tr>
<tr>
<td>twostd()</td>
<td><code>sage.rings.polynomial.multi_polynomial_ideal.NCPolynomialIdeal method</code></td>
<td>351</td>
</tr>
<tr>
<td>TwoTernaryQuadratics</td>
<td><code>class in sage.rings.invariants.invariant_theory</code></td>
<td>446</td>
</tr>
<tr>
<td>unary_pd</td>
<td><code>class in sage.rings.polynomial.polynomial_compiled</code></td>
<td>237</td>
</tr>
<tr>
<td>UnflatteningMorphism</td>
<td><code>class in sage.rings.polynomial.flatten</code></td>
<td>406</td>
</tr>
<tr>
<td>union()</td>
<td><code>sage.rings.polynomial.pbori.BooleSet method</code></td>
<td>596</td>
</tr>
<tr>
<td>units()</td>
<td><code>class in sage.rings.polynomial.multi_polynomial_quotient_ring.PolynomialQuotientRing_generic method</code></td>
<td>230</td>
</tr>
<tr>
<td>univar_pd</td>
<td><code>class in sage.rings.polynomial.multi_polynomial_compiled</code></td>
<td>237</td>
</tr>
<tr>
<td>univariate_polynomial()</td>
<td><code>sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method</code>, 358</td>
<td></td>
</tr>
<tr>
<td><code>sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method</code></td>
<td><code>sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method</code>, 390</td>
<td></td>
</tr>
<tr>
<td>universal_discriminant</td>
<td><code>(in module sage.rings.polynomial.polynomial_element)</code></td>
<td>106</td>
</tr>
<tr>
<td>universe()</td>
<td><code>(in module sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method)</code></td>
<td>362</td>
</tr>
<tr>
<td>unpickle_BooleanPolynomial()</td>
<td><code>(in module sage.rings.polynomial.pbori)</code></td>
<td>643</td>
</tr>
<tr>
<td>unpickle_BooleanPolynomialRing()</td>
<td><code>(in module sage.rings.polynomial.pbori)</code></td>
<td>643</td>
</tr>
<tr>
<td>unpickle_FpT_element</td>
<td><code>(in module sage.rings.fraction_field_FpT)</code></td>
<td>522</td>
</tr>
<tr>
<td>unpickle_MPolynomial_libsingular()</td>
<td><code>(in module sage.rings.polynomial.multi_polynomial_libsingular)</code></td>
<td>391</td>
</tr>
<tr>
<td>unpickle_MPolynomialRing_generic()</td>
<td><code>(in module sage.rings.polynomial.multi_polynomial_ring_base)</code></td>
<td>267</td>
</tr>
<tr>
<td><code>sage.rings.polynomial.multi_polynomial_ring_base</code></td>
<td><code>.MPolynomialRing_generic_v1()</code></td>
<td>267</td>
</tr>
<tr>
<td>unpickle_MPolynomialRing_libsingular()</td>
<td><code>(in module sage.rings.polynomial.multi_polynomial_libsingular)</code></td>
<td>391</td>
</tr>
<tr>
<td>unpickle_BooleanPolynomial()</td>
<td><code>(in module sage.rings.polynomial.pbori)</code></td>
<td>615</td>
</tr>
<tr>
<td>unpickle_BooleanPolynomialRing()</td>
<td><code>(in module sage.rings.polynomial.pbori)</code></td>
<td>643</td>
</tr>
<tr>
<td>universal_discriminant</td>
<td><code>(in module sage.rings.polynomial.polynomial_element)</code></td>
<td>106</td>
</tr>
<tr>
<td>universal_discriminant</td>
<td><code>(in module sage.rings.polynomial.polynomial_element)</code></td>
<td>106</td>
</tr>
<tr>
<td>valuation()</td>
<td><code>(in module sage.rings.polynomial.polydict.ETuple Method)</code></td>
<td>397</td>
</tr>
<tr>
<td>update()</td>
<td><code>(in module sage.rings.polynomial.toy_buchberger)</code></td>
<td>458</td>
</tr>
<tr>
<td>valuation()</td>
<td><code>(in module sage.rings.polynomial.toy_d_basis)</code></td>
<td>465</td>
</tr>
<tr>
<td>usign()</td>
<td><code>(sage.rings.polynomial.real_roots.bernstein_polynomial_factory method)</code></td>
<td>183</td>
</tr>
</tbody>
</table>

**Index 705**
valuation() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial_generic_dense method), 494
valuation_of_coefficient() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 176
value() (sage.rings.polynomial.pborel.CCuddNavigator method), 628
var_pd (class in sage.rings.polynomial.polynomial_compiled), 237
variable() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 303
variable() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 390
variable() (sage.rings.polynomial.pborel.BooleanPolynomial method), 616
variable() (sage.rings.polynomial.pborel.BooleanPolynomialRing method), 627
variable_has_value() (sage.rings.polynomial.pborel.GroebnerStrategy method), 633
variable_names_recursive() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic method), 528
variable_names_recursive() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 267
variable_names_recursive() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 27
VariableBlock (class in sage.rings.polynomial.pborel), 636
VariableConstruct (class in sage.rings.polynomial.pborel), 636
VariableFactory (class in sage.rings.polynomial.pborel), 636
variables() (sage.rings.invariants.invariant_theory.FormsBase method), 425
variables() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse method), 570
variables() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair method), 539
variables() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 546
variables() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 304
variables() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 390
variables() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 362
variables() (sage.rings.polynomial.pborel.BooleanMonomial method), 600
variables() (sage.rings.polynomial.pborel.BooleanPolynomial method), 616
variables() (sage.rings.polynomial.pborel.BooleConstant method), 590
variables() (sage.rings.polynomial.polynomial_element.Polynomial method), 100
variations() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 190
variety() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 344
variety() (sage.rings.polynomial.pborel.BooleanPolynomialIdeal method), 619
varname_key() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse method), 562
vars() (sage.rings.polynomial.pborel.BooleSet method), 596
vars_as_monomial() (sage.rings.polynomial.pborel.BooleanPolynomial method), 616
vector_space_dimension() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 346

W
warp_map (class in sage.rings.polynomial.real_roots), 209
weighted_degree() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 286
weights() (sage.rings.polynomial.term_order.TermOrder method), 258
weil_restriction() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal method), 320
weil_restriction() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_GF2E method), 366
weyl_algebra() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 267
weyl_algebra() (sage.rings.polynomial.polynomial_ring.PolynomialRing_commutative method), 12

wordsize_rational() (in module sage.rings.polynomial.real_roots), 209

X

xgcd() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 176

xgcd() (sage.rings.polynomial.polynomial_gf2x.Polynomial method), 100

xgcd() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 130

xgcd() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 135

xgcd() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_p method), 157

xgcd() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 146

xgcd() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_template method), 149

xgcd() (sage.rings.polynomial.polynomial_zz_pex.Polynomial_template method), 181

Z

zero() (sage.rings.polynomial.pbori.BooleanPolynomialRing method), 627

zeros() (in module sage.rings.polynomial.pbori), 643

zeros_in() (sage.rings.polynomial.pbori.BooleanPolynomial method), 617

ZZ_FpT_coerce (class in sage.rings.fraction_field_FpT), 521