Noncommutative Polynomials

Release 10.4

The Sage Development Team

Jul 23, 2024
CONTENTS

1 Univariate Ore Polynomials .................................................. 1
2 Noncommutative Multivariate Polynomials ...................... 129
3 Indices and Tables ............................................................... 163
Python Module Index ............................................................. 165
Index ................................................................................. 167
UNIVARIATE ORE POLYNOMIALS

1.1 Univariate Ore polynomial rings

This module provides the OrePolynomialRing, which constructs a general dense univariate Ore polynomial ring over a commutative base with equipped with an endomorphism and/or a derivation.

AUTHOR:
- Xavier Caruso (2020-04)

class sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing(base_ring, twisting_map, names, sparse, category=None)

Bases: UniqueRepresentation, Parent

Construct and return the globally unique Ore polynomial ring with the given properties and variable names.

Given a ring $R$ and a ring automorphism $\sigma$ of $R$ and a $\sigma$-derivation $\partial$, the ring of Ore polynomials $R[X; \sigma, \partial]$ is the usual abelian group polynomial $R[X]$ equipped with the modification multiplication deduced from the rule $Xa = \sigma(a)X + \partial(a)$. We refer to [Ore1933] for more material on Ore polynomials.

INPUT:
- base_ring – a commutative ring
- twisting_map – either an endomorphism of the base ring, or a (twisted) derivation of it
- names – a string or a list of strings
- sparse – a boolean (default: False); currently not supported

EXAMPLES:

The case of a twisting endomorphism

We create the Ore ring $\mathbb{F}_5[x, \text{Frob}]$ where Frob is the Frobenius endomorphism:

```sage
# needs sage.rings.finite_rings
sage: k.<a> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S = OrePolynomialRing(k, Frob, 'x'); S
Ore Polynomial Ring in x over Finite Field in a of size 5^3 twisted by a |--> a^5
```
>>> from sage.all import *

# needs sage.rings.finite_rings

k = GF(Integer(5)**Integer(3), names=('a',)); (a,) = k._first_ngens(1)
Frob = k.frobenius_endomorphism()
S = OrePolynomialRing(k, Frob, 'x'); S
Ore Polynomial Ring in x over Finite Field in a of size 5^3 twisted by a |---> a^5

In particular, observe that it is not needed to create and pass in the twisting derivation (which is 0 in our example).

As a shortcut, we can use the square brackets notation as follow:

sage: # needs sage.rings.finite_rings
sage: T.<x> = k[x, Frob]; T
Ore Polynomial Ring in x over Finite Field in a of size 5^3 twisted by a |---> a^5
sage: T is S
True

We emphasize that it is necessary to repeat the name of the variable in the right hand side. Indeed, the following fails (it is interpreted by Sage as a classical polynomial ring with variable name Frob):

sage: T.<x> = k[Frob]
^ needs sage.rings.finite_rings
Traceback (most recent call last):
  ... ValueError: variable name 'Frobenius endomorphism a |---> a^5 on Finite Field in a of size 5^3' is not alphanumeric

Note moreover that, similarly to the classical case, using the brackets notation also sets the variable:

sage: x.parent() is S
^ needs sage.rings.finite_rings
True

We are now ready to carry on computations in the Ore ring:

sage: x*a
^ needs sage.rings.finite_rings
(continues on next page)
The case of a twisting derivation

We can similarly create the Ore ring of differential operators over \( \mathbb{Q}[t] \), namely \( \mathbb{Q}[t,\frac{d}{dt}] \):

\[
\begin{align*}
sage: \quad \text{# needs sage.rings.finite_rings} \\
sage: \quad R.<t> = \mathbb{Q}[t] \\
sage: \quad \text{der} = R.derivation(); \text{der} \\
sage: \quad \frac{d}{dt} \\
sage: \quad A = \text{OrePolynomialRing}(R, \text{der}, 'd'); A \\
\text{Ore Polynomial Ring in } d \text{ over Univariate Polynomial Ring in } t \\
\text{over Rational Field twisted by } \frac{d}{dt}
\end{align*}
\]

Again, the brackets notation is available:

\[
\begin{align*}
sage: \quad B.<d> = R['d', \text{der}] \\
\quad \text{# needs sage.rings.finite_rings} \\
sage: \quad A \text{ is } B \\
\quad \text{# needs sage.rings.finite_rings} \\
\text{True}
\end{align*}
\]

and computations can be carried out:

\[
\begin{align*}
sage: \quad d*t \\
\quad \text{# needs sage.rings.finite_rings} \\
t*d + 1
\end{align*}
\]

1.1. Univariate Ore polynomial rings
The combined case

Ore polynomial rings involving at the same time a twisting morphism \( \sigma \) and a twisting \( \sigma \)-derivation can be created as well as follows:

```
sage: # needs sage.rings.padics
sage: F.<u> = Qq(3^2)
sage: sigma = F.frobenius_endomorphism(); sigma
Frobenius endomorphism on 3-adic Unramified Extension Field in u
defined by \( x^2 + 2*x + 2 \) lifting \( u \mapsto u^3 \) on the residue field
sage: der = F.derivation(3, twist=sigma); der
(3 + O(3^21))*(\text{[Frob] - id})
sage: M.<X> = F[X, der]; M
Ore Polynomial Ring in X over 3-adic Unramified Extension Field in u
defined by \( x^2 + 2*x + 2 \) twisted by Frob and \( (3 + O(3^21))*(\text{[Frob] - id}) \)
```

We emphasize that we only need to pass in the twisted derivation as it already contains in it the datum of the twisting endomorphism. Actually, passing in both twisting maps results in an error:

```
sage: F[X, sigma, der]
#...
sage: Needs sage.rings.padics
Traceback (most recent call last):
...
ValueError: variable name 'Frobenius endomorphism ...' is not alphanumeric
```

```
sage: F['X', sigma, der]
#...
Needs sage.rings.padics
Traceback (most recent call last):
...
ValueError: variable name 'Frobenius endomorphism ...' is not alphanumeric
```

4 Chapter 1. Univariate Ore Polynomials
Examples of variable name context

Consider the following:

```
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = SkewPolynomialRing(R, sigma); S
Ore 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 Ore polynomial ring is unique in Sage and other objects in Sage could have pointers to that Ore polynomial ring.

However, the variable can be changed within the scope of a `with` block using the `localvars` context:

```
sage: with localvars(S, ['y']):
    print(S)
Ore Polynomial Ring in y over Univariate Polynomial Ring in t over Integer Ring twisted by t |--> t + 1
```

Uniqueness and immutability

In Sage, there is exactly one Ore polynomial ring for each quadruple (base ring, twisting morphism, twisting derivation, name of the variable):

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(7^3)
sage: Frob = k.frobenius_endomorphism()
sage: S = k['x', Frob]
sage: T = k['x', Frob]
```

(continues on next page)
Noncommutative Polynomials, Release 10.4

(continued from previous page)

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

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(7)**Integer(3), names=('a',)); (a,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x', Frob]
>>> T = k['x', Frob]
>>> S is T
True
```

Rings with different variables names are different:

```python
sage: S is k['y', Frob]
#→ needs sage.rings.finite_rings
False
```

```python
>>> from sage.all import *
>>> S is k['y', Frob]
#→ needs sage.rings.finite_rings
False
```

Similarly, varying the twisting morphisms yields to different Ore rings (expect when the morphism coincide):

```python
sage: S is k['x', Frob^2]
#→ needs sage.rings.finite_rings
False
sage: S is k['x', Frob^3]
#→ needs sage.rings.finite_rings
False
sage: S is k['x', Frob^4]
#→ needs sage.rings.finite_rings
True
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> S is k['x', Frob^2]
False
>>> S is k['x', Frob^3]
False
>>> S is k['x', Frob^4]
True
```

Todo:
- Sparse Ore Polynomial Ring
- Multivariate Ore Polynomial Ring

```python
Element = None
```

6 Chapter 1. Univariate Ore Polynomials
change_var(var)

Return the Ore polynomial ring in variable var with the same base ring, twisting morphism and twisting derivation as self.

INPUT:

• var – a string representing the name of the new variable

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: R.<x> = OrePolynomialRing(k,Frob); R
Ore 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
Ore 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:

```python
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: R['x',sigma].characteristic()
0
sage: # needs sage.rings.finite_rings
sage: k.<u> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: k['y',Frob].characteristic()
5
```

(continues on next page)
fraction_field()

Return the fraction field of this skew ring.

EXAMPLES:

```sage
# needs sage.rings.finite_rings
k.<a> = GF(5^3)  
k.Frob = k.frobenius_endomorphism()  
S.<x> = k[x, Frob]  
K = S.fraction_field(); K  
Ore Function Field in x over Finite Field in a of size 5^3 twisted by a |--> a^5

f = 1/(x + a); f  
(x + a)^(-1)

f.parent() is K
True
```

Below is another example with differentiel operators:

```sage
R.<t> = QQ[]  
der = R.derivation()  
A.<d> = R[d, der]  
A.fraction_field()  
Ore Function Field in d over Fraction Field of Univariate Polynomial Ring in t over Rational Field twisted by d/dt

f = t/d; f  
(d - 1/t)^(-1) * t

f*d  
t
```

(continues on next page)
\[(d - 1/t)^{-1} * t\]

>>> f.d

See also:

\textit{sage.rings.polynomial.ore_function_field}

\textbf{\textit{gen}(n=0)}

Return the indeterminate generator of this Ore polynomial ring.

**INPUT:**

- \textit{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
Ore 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: S.gen(0)
x
```

This is also known as the parameter:

```
sage: S.parameter() is S.gen()
True
```

\textbf{\textit{gens}()}  

Return the tuple of generators of \texttt{self}.

**EXAMPLES:**
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]; S
Ore Polynomial Ring in x over Univariate Polynomial Ring in t
over Rational Field twisted by t |--> t + 1
sage: S.gens()
(x,)

>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1); S
Ore Polynomial Ring in x over Univariate Polynomial Ring in t
over Rational Field twisted by t |--> t + 1
>>> S.gens()
(x,)

gens_dict()

Return a {name: variable} dictionary of the generators of this Ore polynomial ring.

EXAMPLES:

sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = SkewPolynomialRing(R,sigma)
sage: S.gens_dict()
{'x': x}

>>> from sage.all import *
>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = SkewPolynomialRing(R,sigma, names=('x',)); (x,) = S._first_ngens(1)
>>> S.gens_dict()
{'x': x}

is_commutative()

Return True if this Ore polynomial ring is commutative.

This holds if the twisting morphism is the identity and the twisting derivation vanishes.

EXAMPLES:

sage: # needs sage.rings.finite_rings
sage: k.<a> = 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
sage: R.<t> = GF(5)[]
sage: der = R.derivation()
sage: A.<d> = R['d', der]
sage: A.is_commutative()
False
Noncommutative Polynomials, Release 10.4

sage: B.<b> = R['b', 5*der]
sage: B.is_commutative()
True

>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('a',)); (a,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x', Frob]; (x,) = S._first_ngens(1)
>>> S.is_commutative()
False

T = k['y', Frob**Integer(3)]; (y,) = T._first_ngens(1)
>>> T.is_commutative()
True

>>> R = GF(Integer(5))['t']; (t,) = R._first_ngens(1)
>>> der = R.derivation()
>>> A = R['d', der]; (d,) = A._first_ngens(1)
>>> A.is_commutative()
False

B = R['b', Integer(5)*der]; (b,) = B._first_ngens(1)
>>> B.is_commutative()
True

is_exact()
Return True if elements of this Ore polynomial ring are exact.
This happens if and only if elements of the base ring are exact.

EXAMPLES:

sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)

sage: Frob = k.frobenius_endomorphism()

sage: S.<x> = k['x', Frob]

sage: S.is_exact()
True

sage: S.base_ring().is_exact()
True

sage: R.<u> = k[[[]]

sage: sigma = R.hom([u + u^2])

sage: T.<y> = R['y', sigma]

sage: T.is_exact()
False

sage: T.base_ring().is_exact()
False

>>> from sage.all import *
>>> # needs sage.rings.finite_rings

k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)

Frob = k.frobenius_endomorphism()

S = k['x', Frob]; (x,) = S._first_ngens(1)

S.is_exact()
True

S.base_ring().is_exact()
True

1.1. Univariate Ore polynomial rings

11
is_field (proof=False)

Return always False since Ore polynomial rings are never fields.

EXAMPLES:

```plaintext
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x, Frob]
sage: S.is_field()
False
```

is_finite()

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

EXAMPLES:

```plaintext
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: k.is_finite()
True
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 Ore polynomial ring are sparsely represented.
**Warning:** Since sparse Ore polynomials are not yet implemented, this function always returns `False`.

**EXAMPLES:**

```python
sage: R.<t> = RR[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x', sigma]
sage: S.is_sparse()
False
```

```python
>>> from sage.all import *
>>> R = RR['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x', sigma]; (x,) = S._first_ngens(1)
>>> S.is_sparse()
False
```

**ngens()**

Return the number of generators of this Ore polynomial ring.

This is 1.

**EXAMPLES:**

```python
sage: R.<t> = RR[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x', sigma]
sage: S.ngens()
1
```

```python
>>> from sage.all import *
>>> R = RR['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x', sigma]; (x,) = S._first_ngens(1)
>>> S.ngens()
1
```

**parameter (n=0)**

Return the indeterminate generator of this Ore polynomial ring.

**INPUT:**

- `n` – index of generator to return (default: 0); exists for compatibility with other polynomial rings

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x', sigma]; S
Ore 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: S.gen(0)
```

1.1. Univariate Ore polynomial rings 13
Noncommutative Polynomials, Release 10.4

```python
>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1); S
Ore Polynomial Ring in x over Univariate Polynomial Ring in t
over Rational Field twisted by t |--> t + 1
>>> y = S.gen(); y
x
>>> y == x
True
>>> S.gen(Integer(0))
x
```

This is also known as the parameter:

```python
sage: S.parameter() is S.gen()
True
```

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

Return a random Ore polynomial in this ring.

**INPUT:**

- **degree** — (default: (-1, 2)) integer with degree or a tuple of integers with minimum and maximum degrees
- **monic** — (default: False) if True, return a monic Ore polynomial
- ***args, **kwds** — passed on to the random_element method for the base ring

**OUTPUT:**

Ore 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:**

```python
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
```

```python
sage: Frob = k.frobenius_endomorphism()
```

```python
sage: S.<x> = k['x', Frob]
```

```python
sage: S.random_element()  # random
(2*t^2 + 3)*x^2 + (4*t^2 + t + 4)*x + 2*t^2 + 2
```

```python
sage: S.random_element(monic=True)  # random
x^2 + (2*t^2 + t + 1)*x + 3*t^2 + 3*t + 2
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
```

```python
>>> Frob = k.frobenius_endomorphism()
```

```python
>>> S = k['x', Frob]; (x,) = S._first_ngens(1)
```

```python
>>> S.random_element()  # random
(2*t^2 + 3)*x^2 + (4*t^2 + t + 4)*x + 2*t^2 + 2
```

```python
>>> S.random_element(monic=True)  # random
x^2 + (2*t^2 + t + 1)*x + 3*t^2 + 3*t + 2
```

14 Chapter 1. Univariate Ore Polynomials
Use `degree` to obtain polynomials of higher degree:

```python
sage: # needs sage.rings.finite_rings
sage: p = S.random_element(degree=5)  # random
(t^2 + 3*t)*x^5 + (4*t + 4)*x^3 + (4*t^2 + 4*t)*x^2 + (2*t^2 + 1)*x + 3
sage: p.degree()  # random
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:

```python
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
```

```python
random_irreducible(degree=2, monic=True, *args, **kwds)
```

Return a random irreducible Ore polynomial.

**Warning:** Elements of this Ore polynomial ring need to have a method `is_irreducible()`. Currently, this method is implemented only when the base ring is a finite field.

**INPUT:**

- `degree` – Integer with degree (default: 2) or a tuple of integers with minimum and maximum degrees
- `monic` – if True, returns a monic Ore polynomial (default: True)
- `*args, **kwds` – passed in to the `random_element` method for the base ring

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x,Frob]
sage: A = S.random_irreducible()
sage: A.is_irreducible()
True
sage: B = S.random_irreducible(degree=3, monic=False)
sage: B.is_irreducible()
True
```
twisting_derivation()  
Return the twisting derivation defining this Ore polynomial ring or None if this Ore polynomial ring is not twisted by a derivation.

EXAMPLES:

```python
sage: R.<t> = QQ[]
sage: der = R.derivation(); der
d/dt
g1 including:
    sage: A,<d> = R[d, der]
g2 including:
    sage: A.twisting_derivation()
d/dt
g3 including:
    sage: k.<a> = GF(5^3)
    sage: Frob = k.frobenius_endomorphism()
    sage: S.<x> = k[x, Frob]
    sage: S.twisting_derivation()
```

See also:

```
twisting_morphism()
```

**twisting_morphism(n=n)**

Return the twisting endomorphism defining this Ore polynomial ring iterated n times or None if this Ore polynomial ring is not twisted by an endomorphism.

INPUT:

- n – an integer (default: 1)

EXAMPLES:
If $n$ in negative, Sage tries to compute the inverse of the twisting morphism:

```python
sage: k.<a> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: T.<y> = k[y,Frob]
sage: T.twisting_morphism(-1)
Frobenius endomorphism a |---> a^(5^2) on Finite Field in a of size 5^3
```

Sometimes it fails, even if the twisting morphism is actually invertible:

```python
sage: K = R.fraction_field()
sage: phi = K.hom([(t+Integer(1))/(t-Integer(1))])
sage: T.<y> = K[y, phi]
sage: T.twisting_morphism(-1)
Traceback (most recent call last):
  ...
NotImplementedError: inverse not implemented for morphisms of
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

(continues on next page)
When the Ore polynomial ring is only twisted by a derivation, this method returns nothing:

```
sage: der = R.derivation()
sage: A.<d> = R['x', der]
sage: A
Ore Polynomial Ring in x over Univariate Polynomial Ring in t over Rational Field twisted by d/dt
sage: A.twisting_morphism()
```

Here is an example where the twisting morphism is automatically inferred from the derivation:

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: der = k.derivation(1, twist=Frob)
sage: der
[a |---> a^5] - id
sage: S.<x> = k['x', der]
sage: S.twisting_morphism()
Frobenius endomorphism a |---> a^5 on Finite Field in a of size 5^3
```

See also:

` twisting_derivation()`
1.2 Univariate Ore polynomials

This module provides the OrePolynomial, which constructs a single univariate Ore polynomial over a commutative base equipped with an endomorphism and/or a derivation. It provides generic implementation of standard arithmetical operations on Ore polynomials as addition, multiplication, gcd, lcm, etc.

The generic implementation of dense Ore polynomials is OrePolynomial_generic_dense. The classes ConstantOrePolynomialSection and OrePolynomialBaseringInjection handle conversion from an Ore polynomial ring to its base ring and vice versa.

AUTHORS:
- Xavier Caruso (2020-05)

```python
class sage.rings.polynomial.ore_polynomial_element.ConstantOrePolynomialSection:
    Representation of the canonical homomorphism from the constants of an Ore polynomial ring to the base ring.
    This class is necessary for automatic coercion from zero-degree Ore polynomial ring into the base ring.

EXAMPLES:

```python
sage: from sage.rings.polynomial.ore_polynomial_element import *

sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R[x,sigma]
sage: m = ConstantOrePolynomialSection(S, R); m
Generic map:
    From: Ore 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

```python
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
**Definition**

Let $R$ be a commutative ring equipped with an automorphism $\sigma$ and a $\sigma$-derivation $\partial$. An Ore 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 Ore polynomials is defined by the usual addition operation and the modified multiplication is defined by the rule $Xa = \sigma(a)X + \partial(a)$ for all $a$ in $R$. Ore 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 Ore polynomials in the same ring $S$. The right (resp. left) 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 twisting morphism defining the Ore polynomial ring is invertible.

**EXAMPLES:**

We illustrate some functionalities implemented in this class.

We create the Ore polynomial ring (here the derivation is zero):

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R[x,sigma]; S
Ore Polynomial Ring in x over Univariate Polynomial Ring in t over Integer Ring
  twisted by t |--> t + 1
```

```python
>>> from sage.all import *
>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1); S
Ore Polynomial Ring in x over Univariate Polynomial Ring in t over Integer Ring
  twisted by t |--> t + 1
```

and some elements in it:

```python
sage: a = t + x + 1; a
x + t + 1
sage: b = S([t^2,t+1,1]); b
x^2 + (t + 1)*x + t^2
sage: c = S.random_element(degree=3,monic=True)
sage: c.parent() is S
True
```
Noncommutative Polynomials, Release 10.4

Ring operations are supported:

```python
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 Ore polynomial rings. You will find below a short panorama:

```python
sage: q, r = c.right_quo_rem(b)
sage: c == q*b + r
True

>>> from sage.all import *

>>> q, r = c.right_quo_rem(b)
```
The operators \( \div \) and \( \mod \) give respectively the quotient and the remainder of the right Euclidean division:

\[
\begin{align*}
\text{sage: } q &\ = c \div b \\
\text{sage: } r &\ = c \mod b
\end{align*}
\]

Here we can see the effect of the operator evaluation compared to the usual polynomial evaluation:

\[
\begin{align*}
\text{sage: } a &\ = x^2 \\
\text{sage: } a(t)
\end{align*}
\]

Here is another example over a finite field:

\[
\begin{align*}
\text{sage: } &\ None
\end{align*}
\]
Once we have Euclidean divisions, we have for free gcd and lcm (at least if the base ring is a field):

```sage
# needs sage.rings.finite_rings
a = (x + t) * (x + t^2)^2
b = (x + t) * (t*x + t + 1) * (x + t^2)
print(a.right_gcd(b))
x + t^2
print(a.left_gcd(b))
x + t
```

The left lcm has the following meaning: given Ore polynomials $a$ and $b$, their left lcm is the least degree polynomial $c = ua = vb$ for some Ore polynomials $u, v$. Such a $c$ always exist if the base ring is a field:

```sage
# needs sage.rings.finite_rings
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 + 4
print(c.is_right_divisible_by(a))
True
print(c.is_right_divisible_by(b))
True
```

(continues on next page)
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
˓→ needs sage.rings.finite_rings
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 + 3*t)*x + 4*t + 4
```

```python
sage: d.is_left_divisible_by(a)  #...
˓→ needs sage.rings.finite_rings
True
```

```python
sage: d.is_left_divisible_by(b)  #...
˓→ needs sage.rings.finite_rings
True
```

See also:

- `sage.rings.polynomial.ore_polynomial_ring`

**base_ring()**

Return the base ring of `self`.

**EXAMPLES:**

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R[sigma]
sage: a = S.random_element()
sage: a.base_ring()
Univariate Polynomial Ring in t over Integer Ring
```

```python
>>> from sage.all import *
>>> d = a.right_lcm(b); d  #...
˓→ needs sage.rings.finite_rings
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 + 3*t)*x + 4*t + 4
>>> d.is_left_divisible_by(a)  #...
˓→ needs sage.rings.finite_rings
True
>>> d.is_left_divisible_by(b)  #...
˓→ needs sage.rings.finite_rings
True
```

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+Integer(1)])
sage: S.<x> = R['x',sigma]
sage: a = S.random_element()
sage: a.base_ring()
Univariate Polynomial Ring in t over Integer Ring
```

```python
>>> from sage.all import *
>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> a = S.random_element()
>>> a.base_ring()
Univariate Polynomial Ring in t over Integer Ring
```

```python
>>> a.base_ring() is R
True
```
change_variable_name(var)

Change the name of the variable of self.

This will create the Ore polynomial ring with the new name but same base ring, twisting morphism and twisting derivation. The returned Ore polynomial will be an element of that Ore polynomial ring.

INPUT:

• var – the name of the new variable

EXAMPLES:

```
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:

```
sage: b.parent()
Ore 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:

```
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]

(continues on next page)```
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 Ore polynomial has degree -1.

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 + 1
sage: a.degree()
3
sage: S.zero().degree()
-1
sage: S(5).degree()
0
```
Noncommutative Polynomials, Release 10.4

(continued from previous page)

```python
>>> a = x**Integer(2) + t*x**Integer(3) + t**Integer(2)*x + Integer(1)
>>> a.degree()
3
>>> S.zero().degree()
-1
>>> S(Integer(5)).degree()
0
```

**exponents()**

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

**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.exponents()
[0, 2, 4]
```

```python
>>> from sage.all import *
```

```python
R = QQ['t']; (t,) = R._first_ngens(1)
sigma = R.hom([t+Integer(1)])
S = R['x',sigma]; (x,) = S._first_ngens(1)
a = Integer(1) + x**Integer(4) + (t+Integer(1))*x**Integer(2) +
   t**Integer(2)
>>> 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:**

```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.number_of_terms()
3
```

```python
>>> from sage.all import *
```

```python
R = QQ['t']; (t,) = R._first_ngens(1)
sigma = R.hom([t+Integer(1)])
S = R['x',sigma]; (x,) = S._first_ngens(1)
a = Integer(1) + x**Integer(4) + (t+Integer(1))*x**Integer(2) +
   t**Integer(2)
>>> a.number_of_terms()
3
```

This is also an alias for `hamming_weight`:

```python
sage: a.hamming_weight()
3
```

1.2. Univariate Ore polynomials 27
is_constant()  
Return whether self is a constant polynomial.

EXAMPLES:

```sage
t.<t> = QQ[]
sigma = R.hom([t+1])
S.<x> = R[x,sigma]
R(2).is_constant()  True
(x + 1).is_constant()  False
```

is_left_divisible_by(other)
Check if self is divisible by other on the left.

INPUT:

- other – an Ore polynomial in the same ring as self

OUTPUT:

Return True or False.

EXAMPLES:

```sage
# needs sage.rings.finite_rings
k.<t> = GF(5^3)
Frob = k.frobenius_endomorphism()
S.<x> = k['x',Frob]
a = x^2 + t*x + t^2 + 3
b = x^3 + (t + 1)*x^2 + 1
c = a*b
c.is_left_divisible_by(a)  True
c.is_left_divisible_by(b)  False
```

(continues on next page)
Divisibility by 0 does not make sense:

```python
sage: c.is_left_divisible_by(S(0))  # needs sage.rings.finite_rings
Traceback (most recent call last):
  ... ZeroDivisionError: division by zero is not valid
```

```
>>> from sage.all import *

>>> c.is_left_divisible_by(S(Integer(0)))  # needs sage.rings.finite_rings
Traceback (most recent call last):
  ... ZeroDivisionError: division by zero is not valid
```

**is_monic()**

Return True if this Ore 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()  # needs sage.rings.finite_rings
True
sage: a = 0*x
sage: a.is_monic()  # needs sage.rings.finite_rings
False
sage: a = t*x^3 + x^4 + (t+1)*x^2
sage: a.is_monic()  # needs sage.rings.finite_rings
True
sage: a = (t^2 + 2*t)*x^2 + x^3 + t^10*x^5
sage: a.is_monic()  # needs sage.rings.finite_rings
False
```

```python
>>> from sage.all import *

>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x'],sigma; (x,) = S._first_ngens(1)
>>> a = x + t
>>> a.is_monic()  # needs sage.rings.finite_rings
True
>>> a = Integer(0)*x
>>> a.is_monic()  # needs sage.rings.finite_rings
False
>>> a = t*x**Integer(3) + x**Integer(4) + (t+Integer(1))*x**Integer(2)
>>> a.is_monic()  # needs sage.rings.finite_rings
```

(continues on next page)
is_monomial()

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

EXAMPLES:

```
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
sage: (x^2).is_monomial()  # True
sage: S(1).is_monomial()  # True
```

The coefficient must be 1:

```
sage: (2*x^5).is_monomial()  # False
sage: S(t).is_monomial()  # False
```

```
>>> from sage.all import *

>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> x.is_monomial()  # True
>>> (x+Integer(1)).is_monomial()  # False
>>> (x**Integer(1)).is_monomial()  # True
>>> S(Integer(1)).is_monomial()  # True
```

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

```
sage: (2*x^5).is_term()  # True
sage: S(t).is_term()  # True
```
is_nilpotent()
Check if self is nilpotent.

Note: The paper “Nilpotents and units in skew polynomial rings over commutative rings” by M. Rimmer and K.R. Pearson describes a 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.

EXAMPLES:

```python
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R[x,sigma]
sage: x.is_nilpotent()
Traceback (most recent call last):
... NotImplementedError
```

```python
>>> from sage.all import *
>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+1])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> x.is_nilpotent()
Traceback (most recent call last):
... NotImplementedError
```

is_one()
Test whether this polynomial is 1.

EXAMPLES:

```python
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: R(1).is_one()
True
sage: (x + 3).is_one()
False
```

```python
>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+1])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> R(Integer(1)).is_one()
True
>>> (x + Integer(3)).is_one()
False
```
is_right_divisible_by\(\text{(other)}\)
Check if \(\text{self}\) is divisible by \(\text{other}\) on the right.

INPUT:

- \text{other} – an Ore polynomial in the same ring as \text{self}

OUTPUT:

Return True or False.

EXAMPLES:

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

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

EXAMPLES:

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

>>> from sage.all import *
>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> x.is_term()
True
>>> R(Integer(1)).is_term()
True
>>> (Integer(3)*x**Integer(5)).is_term()
True
>>> (Integer(1)+Integer(3)*x**Integer(5)).is_term()
False

If you want to test that self also has leading coefficient 1, use is_monomial() instead:

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

Return True if this Ore polynomial is a unit.

When the base ring $R$ is an integral domain, then an Ore 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+Integer(1)])
sage: S.<x> = R[x,sigma]
sage: a = x + (t+Integer(1))*x^5 + t^2*Integer(3) - x^5
sage: a.is_zero()
False
```
**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
```

By convention, the leading coefficient to the zero polynomial is zero:

```python
sage: S(0).leading_coefficient()
0
```

**left_divides(other)**

Check if self divides other on the left.

**INPUT:**

- other – an Ore polynomial in the same ring as self

**OUTPUT:**

Return True or False.

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
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.left_divides(c)
True
sage: b.left_divides(c)
False
```

(continues on next page)
Divisibility by 0 does not make sense:

```python
sage: S(0).left_divides(c)  # needs sage.rings.finite_rings
ZeroDivisionError: division by zero is not valid
```

---

**left_gcd**(other, monic=True)

Return the left gcd of self and other.

**INPUT:**

- other – an Ore 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 an Ore polynomial \( g \) with the following property: any Ore 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 following two conditions are fulfilled (otherwise left gcd do not exist in general): 1) the base ring is a field and 2) the twisting morphism is bijective.

**EXAMPLES:**

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

```python
sage: a.left_gcd(b, monic=False)
# needs sage.rings.finite_rings
2*t*x + 4*t + 2
```

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) * (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: the base ring must be a field
```

And the twisting morphism 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: the base ring must be a field
```
NotImplementedError: inversion of the twisting morphism Ring endomorphism of Fraction Field of Univariate Polynomial Ring in t over Rational Field
    Defn: t |--> t^2

left_lcm(other, monic=True)

Return the left lcm of self and other.

INPUT:

- other – an Ore 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 an Ore polynomial l with the following property: any Ore polynomial is a left multiple of l (right divisible by l) iff it is a left multiple of both self and other (right divisible by self and other). If monic is True, l 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:

sage: # needs sage.rings.finite_rings
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 + 4 + 3*t^2 + 3*t + 2)*x^2 + (t^2 + 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 \texttt{monic=False}, we can get a nonmonic \texttt{lcm}:

```
sage: a.left_lcm(b, monic=False)  
(3*t^2 + 4)*x^5 + (4*t^2 + 4*t + 1)*x^4 + (t + 1)*x^3 + (t^2 + 2)*x^2 + (3*t + 4)*x
```

The base ring needs to be a field:

```
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
```

```
\texttt{left\_mod(}other\texttt{)}
\begin{flushleft}
Return the remainder of left division of \texttt{self} by \texttt{other}.
\end{flushleft}

\textbf{EXAMPLES:}

1.2. Univariate Ore polynomials
**left_monic()**

Return the unique monic Ore polynomial $m$ which divides this polynomial on the left and has the same degree.

Given an Ore polynomial $P$ of degree $n$, its left monic is given by $P \cdot \sigma^{-n}(1/k)$, where $k$ is the leading coefficient of $P$ and $\sigma$ is the twisting morphism.

**EXAMPLES:**

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

```sage
sage: b.degree() == a.degree()  # True
sage: a.is_left_divisible_by(b)  # True
sage: twist = S.twisting_morphism(-a.degree())
```

(continues on next page)
Noncommutative Polynomials, Release 10.4

sage: a == b * twist(a.leading_coefficient())
True

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> b.degree() == a.degree()
True
>>> a.is_left_divisible_by(b)
True
>>> twist = S.twisting_morphism(-a.degree())
>>> a == b * twist(a.leading_coefficient())
True
```

Note that \( b \) does not divide \( a \) on the right:

```python
sage: a.is_right_divisible_by(b)  # needs sage.rings.finite_rings
False

>>> from sage.all import *
>>> a.is_right_divisible_by(b)  # needs sage.rings.finite_rings
False
```

This function does not work if the leading coefficient is not a unit:

```python
sage: R.<t> = QQ[]
sage: der = R.derivation()
sage: S.<x> = R[der]
sage: a = t*x
sage: a.left_monic()
Traceback (most recent call last):
  ... Not Implemented Error: the leading coefficient is not a unit
```

```python
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> der = R.derivation()
>>> S = R['x', der]; (x,) = S._first_ngens(1)
>>> a = t*x
>>> a.left_monic()
Traceback (most recent call last):
  ... Not Implemented Error: the leading coefficient is not a unit
```

left_quo_rem\( (other) \)
Return the quotient and remainder of the left Euclidean division of \( self \) by \( other \).

**INPUT:**
- \( other \) – an Ore polynomial in the same ring as \( self \)

**OUTPUT:**
- the quotient and the remainder of the left Euclidean division of this Ore polynomial by \( other \)
Note: This will fail if the leading coefficient of \texttt{other} is not a unit or if Sage can’t invert the twisting morphism.

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x,Frob]
```

```python
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
```

```python
sage: b = (3*t^2 + 4*t + 2)*x^2 + (2*t^2 + 4*t + 3)*x + 2*t^2 + t + 1
```

```python
sage: q,r = a.left_quo_rem(b)
```

```python
sage: a == b*q + r
True
```

>>> from sage.all import *

```python
>>> # needs sage.rings.finite_rings
```

```python
>>> k = GF(Integer(5)**Integer(3), names=(t,)); (t,) = k._first_ngens(1)
```

```python
>>> Frob = k.frobenius_endomorphism()
```

```python
>>> S = k[x,Frob]; (x,) = S._first_ngens(1)
```

```python
>>> a = (Integer(3)*t**Integer(2) + Integer(3)*t + Integer(2))*x**Integer(3) + (Integer(2)*t**Integer(2) + Integer(3))*x**Integer(2) + (Integer(4)*t**Integer(2) + t + Integer(4))*x + Integer(2)*t**Integer(2) + Integer(2)
```

```python
>>> b = (Integer(3)*t**Integer(2) + Integer(4)*t + Integer(2))*x**Integer(2) + (Integer(2)*t**Integer(2) + Integer(4))*x + Integer(2)*t**Integer(2) + Integer(1)
```

```python
>>> q,r = a.left_quo_rem(b)
```

```python
>>> a == b*q + r
True
```

In the following example, Sage does not know the inverse of the twisting morphism:

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

```python
sage: K = R.fraction_field()
```

```python
sage: sigma = K.hom([[(t+1)/(t-1)]])
```

```python
sage: S.<x> = K[x,sigma]
```

```python
sage: a = (-2*t^2 - t + 1)*x^3 + (-t^2 + t)*x^2 + (-12*t - 2)*x - t^2 - 95*t + 1
```

```python
sage: b = x^2 + (5*t - 6)*x - 4*t^2 + 4*t - 1
```

```python
sage: a.left_quo_rem(b)
```

```
Traceback (most recent call last):
...
NotImplementedError: inversion of the twisting morphism Ring endomorphism of Fraction Field of Univariate Polynomial Ring in t over Rational Field
   Defn: t |--> (t + 1)/(t - 1)
```

```python
>>> from sage.all import *

```
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 Ore 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 – an Ore 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 an Ore polynomial \(g\) with the following property: any Ore 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 Ore polynomials \(u\) and \(v\) such that:

\[ g = a \cdot u + b \cdot v, \]

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 twisting morphism is bijective.

EXAMPLES:

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

```python
>>> from sage.all import *
```

```python
# needs sage.rings.finite_rings
>>> k = GF(Integer(5)^Integer(3), names=('t',)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x',Frob]; (x,) = S._first_ngens(1)
>>> a = (x + t) * (x**Integer(2) + t*x + Integer(1))
```
Noncommutative Polynomials, Release 10.4

(continued from previous page)

```python
>>> b = Integer(2) * (x + t) * (x**Integer(3) + (t+Integer(1))*x**Integer(2)
˓→+ t**Integer(2))
>>> g,u,v = a.left_xgcd(b); g
x + t
>>> a*u + b*v == g
True
```

Specifying monic=False, we can get a nonmonic gcd:

```python
sage: g,u,v = a.left_xgcd(b, monic=False); g
# _needs sage.rings.finite_rings
2*t*x + 4*t + 2
sage: a*u + b*v == g
# _needs sage.rings.finite_rings
True
```

The base ring must be a field:

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

sage: sigma = R.hom([t+1])

sage: S.<x> = R[<x>]

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 twisting morphism must be bijective:

```python
sage: FR = R.fraction_field()

sage: f = FR.hom([FR(t)^2])

sage: S.<x> = FR[<x>]

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
```

(continues on next page)
Traceback (most recent call last):
...
NotImplementedError: inversion of the twisting morphism Ring endomorphism of
→Fraction Field of Univariate Polynomial Ring in t over Rational Field
Defn: t |--> t^2

>>> from sage.all import *
>>> FR = R.fraction_field()
>>> f = FR.hom([FR(t)**Integer(2)])
>>> S = FR['x','f']; (x,) = S._first_nngens(1)
>>> a = (x + t) * (x**Integer(2) + t*x + Integer(1))
>>> b = Integer(2) * (x + t) * (x**Integer(3) + (t+Integer(1))*x**Integer(2)← t**Integer(2))
>>> a.left_xgcd(b)
Traceback (most recent call last):
...
NotImplementedError: inversion of the twisting morphism Ring endomorphism of
→Fraction Field of Univariate Polynomial Ring in t over Rational Field
Defn: t |--> t^2

left_xlcm(other, monic=True)

Return the left lcm $L$ of self and other together with two Ore polynomials $U$ and $V$ such that

$$U \cdot self = V \cdot other = L.$$

EXAMPLES:

sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)

sage: Frob = k.frobenius_endomorphism()

sage: S.<x> = k[x,Frob]

sage: P = (x + t^2) * (x + t)

sage: Q = 2 * (x^2 + t + 1) * (x * t)

sage: L, U, V = P.left_xlcm(Q)

sage: L
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: U * P == L  # needs sage.rings.finite_rings
True

sage: V * Q == L  # needs sage.rings.finite_rings
True

(continues on next page)
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
```

padded_list(n=None)

Return list of coefficients of self up to (but not including) degree n.

Includes 0’s in the positions so that the list always has length exactly n.

INPUT:

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

EXAMPLES:

```
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]
```

(continues on next page)
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

>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> a = Integer(1) + t*x**Integer(3) + t**Integer(2)*x**Integer(5)
>>> a.padded_list()
[1, 0, 0, t, 0, t^2]
>>> a.padded_list(Integer(10))
[1, 0, 0, t, 0, t^2, 0, 0, 0, 0]
>>> len(a.padded_list(Integer(10)))
10
>>> a.padded_list(Integer(3))
[1, 0, 0]
>>> a.padded_list(Integer(0))
[]
>>> a.padded_list(-Integer(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:
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: x.prec()
+Infinity

>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> x.prec()
+Infinity

right_divides(other)
Check if self divides other on the right.

INPUT:
• other – an Ore polynomial in the same ring as self

OUTPUT:
Return True or False.

EXAMPLES:

```sage
# needs sage.rings.finite_rings
k.<t> = GF(5^3)
Frob = k.frobenius_endomorphism()
S.<x> = k['x',Frob]
a = x^2 + t*x + t^2 + 3
b = x^3 + (t + 1)*x^2 + 1
c = a * b
a.right_divides(c)
False
b.right_divides(c)
True
```

Divisibility by 0 does not make sense:

```sage
S(0).right_divides(c)  # this
needs sage.rings.finite_rings
Traceback (most recent call last):
... ZeroDivisionError: division by zero is not valid
```

```sage
from sage.all import *
S(Integer(0)).right_divides(c)   # this
needs sage.rings.finite_rings
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:

```sage
R.<t> = QQ[]
sigma = R.hom([t+1])
S.<x> = R['x',sigma]
a = x^2 + 2*x + t
b = (t+1)*x + t^2
c = a*b
b.right_divides(c)
Traceback (most recent call last):
```

... Not Implemented Error: the leading coefficient of the divisor is not invertible

```python
>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> a = x**Integer(2) + Integer(2)*x + t
>>> b = (t+Integer(1))*x + t**Integer(2)
>>> c = a*b
>>> b.right_divides(c)
Traceback (most recent call last):
... Not Implemented Error: the leading coefficient of the divisor is not invertible
```

right_gcd(other, monic=True)

Return the right gcd of self and other.

INPUT:

• other – an Ore 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 an Ore polynomial \( g \) with the following property: any Ore 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:

```python
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x,Frob]
sage: a = (x**2 + t*x + Integer(1)) * (x + t)
sage: b = Integer(2) * (x**3 + (t+Integer(1))*x**2 + t**Integer(2)) * (x + t)
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=(t,)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x',Frob]; (x,) = S._first_ngens(1)
>>> a = (x**Integer(2) + t*x + Integer(1)) * (x + t)
>>> b = Integer(2) * (x**Integer(3) + (t+Integer(1))*x**Integer(2) +...
>>> a.right_gcd(b)
```

Specifying monic=False, we can get a nonmonic gcd:

```
... Not Implemented Error: the leading coefficient of the divisor is not invertible
```
Noncommutative Polynomials, Release 10.4

sage: a.right_gcd(b,monic=False)  # needs sage.rings.finite_rings
(4*t^2 + 4*t + 1)*x + 4*t^2 + 4*t + 3

>>> from sage.all import *
>>> a.right_gcd(b,monic=False)  # needs sage.rings.finite_rings
(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)
Traceback (most recent call last):
  ... TypeError: the base ring must be a field

>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> a = (x**2 + t*x + 1) * (x + t)
>>> b = 2 * (x**3 + (t+1)*x^2 + t^2) * (x + t)
>>> a.right_gcd(b)
Traceback (most recent call last):
  ... TypeError: the base ring must be a field

right_lcm(other, monic=True)

Return the right lcm of self and other.

INPUT:

- other – an Ore 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 an Ore polynomial l with the following property: any Ore polynomial is a right multiple of g (left divisible by l) iff it is a right multiple of both self and other (left divisible by 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 right lcm do not exist in general): 1) the base ring is a field and 2) the twisting morphism on this field is bijective.

EXAMPLES:

sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)

(continues on next page)
**Noncommutative Polynomials, Release 10.4**

(continued from previous page)

```python
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:

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x',Frob]; (x,) = S._first_ngens(1)
>>> a = (x + t) * (x + t**Integer(2))
>>> b = Integer(2) * (x + t) * (x**Integer(2) + t + Integer(1))
>>> c = a.right_lcm(b); c
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:

```python
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
```

1.2. Univariate Ore polynomials 51
>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> a = (x + t) * (x + t**Integer(2))
>>> b = Integer(2) * (x + t) * (x**Integer(2) + t + Integer(1))
>>> a.right_lcm(b)
Traceback (most recent call last):
... TypeError: the base ring must be a field

And the twisting morphism needs to be bijective:

sage: FR = R.fraction_field()
sage: f = FR.hom([FR(t)**Integer(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 twisting morphism

right_mod(other)
Return the remainder of right division of self by other.

EXAMPLES:
Noncommutative Polynomials, Release 10.4

(continued from previous page)

```python
>>> a = Integer(1) + t*x**Integer(2)
>>> b = x + Integer(1)
>>> a % b
 t + 1
>>> (x**Integer(3) + x - Integer(1)).right_mod(x**Integer(2) - Integer(1))
 2*x - 1
```

**right_monic()**

Return the unique monic Ore polynomial which divides this polynomial on the right and has the same degree.

Given an Ore polynomial $P$ of degree $n$, its right monic is given by $(1/k) \cdot P$, where $k$ is the leading coefficient of $P$.

**EXAMPLES:**

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

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x',Frob]; (x,) = S._first_ngens(1)
>>> a = (Integer(3)*t**Integer(2) + Integer(3)*t + Integer(2))*x^3 +
    (Integer(2)*t**Integer(2) + Integer(3))*x^2 +
    (Integer(4)*t**Integer(2) + t + Integer(4))*x + Integer(2)*t**Integer(2) +
    Integer(2)
>>> 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()                           # needs sage.rings.finite_rings
True
sage: a.is_right_divisible_by(b)                          # needs sage.rings.finite_rings
True
sage: a == a.leading_coefficient() * b                    # needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> b.degree() == a.degree()                           # needs sage.rings.finite_rings
True
>>> a.is_right_divisible_by(b)                          # needs sage.rings.finite_rings
True
>>> a == a.leading_coefficient() * b                    # needs sage.rings.finite_rings
True
```

(continues on next page)
Note that $b$ does not divide $a$ on the right:

```python
sage: a.is_left_divisible_by(b)  # needs sage.rings.finite_rings
False
```

This function does not work if the leading coefficient is not a unit:

```python
sage: R.<t> = QQ[]
sage: der = R.derivation()
sage: S.<x> = R[der]
sage: a = t*x
sage: a.right_monic()
Traceback (most recent call last):
... NotImplementedError: the leading coefficient is not a unit
```

```python
>>> from sage.all import *
>>> R = QQ[t]; (t,) = R._first_ngens(1)
>>> der = R.derivation()
>>> S = R[der]; (x,) = S._first_ngens(1)
>>> a = t*x
>>> a.right_monic()
Traceback (most recent call last):
... NotImplementedError: the leading coefficient is not a unit
```

**right_quo_rem**(*other*)

Return the quotient and remainder of the right Euclidean division of `self` by `other`.

**INPUT:**

- `other` – an Ore polynomial in the same ring as `self`

**OUTPUT:**

- the quotient and the remainder of the right Euclidean division of this Ore polynomial by `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)
sage: b = S.random_element(monic=True)
```
sage: q, r = a.right_quo_rem(b)
sage: a == q*b + r
True

>>> from sage.all import *
>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> a = S.random_element(degree=Integer(4))
>>> b = S.random_element(monic=True)
>>> q, r = a.right_quo_rem(b)
>>> a == q*b + r
True

The leading coefficient of the divisor need to be invertible:

sage: a.right_quo_rem(S(0))
Traceback (most recent call last):
  ... ZeroDivisionError: division by zero is not valid
sage: c = S.random_element()
sage: while not c or c.leading_coefficient().is_unit():
  ... c = S.random_element()
sage: while a.degree() < c.degree():
  ... a = S.random_element(degree=4)
sage: a.right_quo_rem(c)
Traceback (most recent call last):
  ... NotImplementedError: the leading coefficient of the divisor is not invertible

>>> from sage.all import *
>>> a.right_quo_rem(S(Integer(0)))
Traceback (most recent call last):
  ... ZeroDivisionError: division by zero is not valid
>>> c = S.random_element()
>>> while not c or c.leading_coefficient().is_unit():
  ... c = S.random_element()
>>> while a.degree() < c.degree():
  ... a = S.random_element(degree=Integer(4))
>>> a.right_quo_rem(c)
Traceback (most recent call last):
  ... NotImplementedError: the leading coefficient of the divisor is not invertible

right_xgcd\text{\ (\textit{other}, \textit{monic}=\textit{True})}

Return the right gcd of \textit{self} and \textit{other} along with the coefficients for the linear combination.

If \(a\) is \textit{self} and \(b\) is \textit{other}, then there are Ore 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:

- \texttt{other} – an Ore polynomial in the same ring as \texttt{self}
- \texttt{monic} – boolean (default: \texttt{True}); return whether the right gcd should be normalized to be monic

OUTPUT:
• The right gcd of self and other, that is an Ore polynomial \( g \) with the following property: any Ore polynomial is divisible on the right by \( g \) if 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 Ore polynomials \( u \) and \( v \) such that:

\[
g = u \cdot a + v \cdot b
\]

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: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x,Frob]
```

```python
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
```

```python
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
```

```python
sage: u*a + v*b == 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^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

```python
>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> a = (x**Integer(2) + t*x + Integer(1)) * (x + t)
>>> b = Integer(2) * (x**Integer(3) + (t+Integer(1))*x**Integer(2) + ...
                                 + t**Integer(2)) * (x + t)
>>> a.right_xgcd(b)
Traceback (most recent call last):...
TypeError: the base ring must be a field
```

right_lcm(other, monic=True)

Return the right lcm \( L \) of \( self \) and \( other \) together with two Ore polynomials \( U \) and \( V \) such that

\[
self \cdot U = other \cdot V = L.
\]

INPUT:

- **other** — an Ore polynomial in the same ring as \( self \)
- **monic** — a boolean (default: True); whether the right lcm should be normalized to be monic

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)

sage: Frob = k.frobenius_endomorphism()

sage: S.<x> = k['x',Frob]

sage: P = (x + t) * (x + t^2)

sage: Q = Integer(2) * (x + t) * (x^2 + t + Integer(1))

sage: L, U, V = P.right_lcm(Q)

sage: L
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: P * U == L
True

sage: Q * V == L
True
```

```python
>>> from sage.all import *

# needs sage.rings.finite_rings

>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)

>>> Frob = k.frobenius_endomorphism()

>>> S = k['x',Frob]; (x,) = S._first_ngens(1)

>>> P = (x + t) * (x + t**Integer(2))

>>> Q = Integer(2) * (x + t) * (x**Integer(2) + t + Integer(1))

(continues on next page)
```

1.2. Univariate Ore polynomials 57
Noncommutative Polynomials, Release 10.4

(continued from previous page)

```python
>>> L, U, V = P.right_xlcm(Q)
>>> L
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
>>> P * U == L
True
>>> Q * V == L
True
```

**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:**

```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R[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:

```
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
```

```
>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> a = x**Integer(5) + t**Integer(4)*x**Integer(4) +
   +t**Integer(2)*x**Integer(2) + t**Integer(10)
>>> a.shift(Integer(0))
x^5 + t^4*x^4 + t^2*x^2 + t^10
>>> a.shift(-Integer(1))
x^4 + t^4*x^3 + t^2*x
>>> a.shift(-Integer(5))
1
>>> a.shift(Integer(2))
x^7 + t^4*x^6 + t^2*x^4 + t^10*x^2
```

```
>>> from sage.all import *
>>> a >> Integer(2)
x^3 + t^4*x^2 + t^2
>>> a << Integer(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
```

```python
>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x', sigma]; (x,) = S._first_ngens(1)
>>> a = x + t
>>> a.variable_name()
x
```

**variable_name()**

Return the string name of the variable used in `self`.

**EXAMPLES:**

```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'
```

```python
>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x', sigma]; (x,) = S._first_ngens(1)
>>> a = x + t
>>> a.variable_name()
'x'
```
Bases: Morphism

Representation of the canonical homomorphism from a ring \( R \) into an Ore polynomial ring over \( R \).

This class is necessary for automatic coercion from the base ring to the Ore 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
Ore Polynomial base injection morphism:
  From: Univariate Polynomial Ring in t over Rational Field
  To:   Ore Polynomial Ring in x over Univariate Polynomial Ring in t
        over Rational Field twisted by t |---> t + 1
```

```python
>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> S.coerce_map_from(S.base_ring()) #indirect doctest
Ore Polynomial base injection morphism:
  From: Univariate Polynomial Ring in t over Rational Field
  To:   Ore 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: # needs sage.rings.finite_rings
sage: from sage.rings.polynomial.ore_polynomial_element import OrePolynomialBaseringInjection
sage: k.<t> = GF(5^3)
Sage: Frob = k.frobenius_endomorphism()
Sage: S.<x> = k[x,Frob]
Sage: m = OrePolynomialBaseringInjection(k, k[x,Frob])
Sage: m.an_element()
x
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> from sage.rings.polynomial.ore_polynomial_element import OrePolynomialBaseringInjection
>>> k = GF(Integer(5)**Integer(3), names =(t,)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x',Frob]; (x,) = S._first_ngens(1)
>>> m = OrePolynomialBaseringInjection(k, k['x', Frob])
>>> m.an_element()
x
```

section()
Return the canonical homomorphism from the constants of an Ore polynomial ring to the base ring according to self.

```python
class sage.rings.polynomial.ore_polynomial_element.OrePolynomial_generic_dense

Bases: OrePolynomial

Generic implementation of dense Ore polynomial supporting any valid base ring, twisting morphism and twisting derivation.

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:
```

```python
codeblock

degree()

Return the degree of self.

By convention, the zero Ore polynomial has degree −1.

EXAMPLES:
```

```
By convention, the degree of 0 is \(-1\):

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

```
>>> from sage.all import *

>>> S(Integer(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}
```

```
>>> from sage.all import *

>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> a = x**Integer(2012) + t*x**Integer(1006) + t**Integer(3) + Integer(2)*t
>>> a.dict()
{0: t^3 + 2*t, 1006: t, 2012: 1}
```

hilbert_shift(s, var=None)

Return this Ore polynomial with variable shifted by \(s\), i.e. if this Ore polynomial is \(P(x)\), return \(P(x + s)\).

INPUT:

- \(s\) – an element in the base ring
- \(var\) – a string; the variable name

EXAMPLES:

```
sage: R.<t> = GF(7)[]
sage: der = R.derivation()
sage: A.<d> = R[d, der]
sage: L = d^3 + t*d^2
sage: L.hilbert_shift(t)
d^3 + 4*t*d^2 + (5*t^2 + 3)*d + 2*t^3 + 4*t
```

```
>>> from sage.all import *

>>> R = GF(Integer(7))[t]; (t,) = R._first_ngens(1)
>>> der = R.derivation()
>>> A = R['d', der]; (d,) = A._first_ngens(1)

>>> L = d**Integer(3) + t*d**Integer(2)
>>> L.hilbert_shift(t)
d^3 + 4*t*d^2 + (5*t^2 + 3)*d + 2*t^3 + 4*t
```

(continues on next page)
Noncommutative Polynomials, Release 10.4

(d+t)**Integer(3) + t*(d+t)**Integer(2)
d^3 + 4*t*d^2 + (5*t^2 + 3)*d + 2*t^3 + 4*t

One can specify another variable name:

sage: L.hilbert_shift(t, var='x')
x^3 + 4*t*x^2 + (5*t^2 + 3)*x + 2*t^3 + 4*t

When the twisting morphism is not trivial, the output lies in a different Ore polynomial ring:

sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x, Frob]
sage: P = x^2 + a*x + a^2
sage: Q = P.hilbert_shift(a); Q
x^2 + (2*a^2 + a + 4)*x + a^2 + 3*a + 4
sage: Q.parent()
Ore Polynomial Ring in x over Finite Field in a of size 5^3 twisted by a |--> a^5 and a*(|a |--> a^5) - id
sage: Q.parent() is S
False

This behavior ensures that the Hilbert shift by a fixed element defines a homomorphism of rings:

sage: # needs sage.rings.finite_rings
sage: U = S.random_element(degree=5)
sage: V = S.random_element(degree=5)
sage: s = k.random_element()
sage: (U+V).hilbert_shift(s) == U.hilbert_shift(s) + V.hilbert_shift(s)
True
sage: (U*V).hilbert_shift(s) == U.hilbert_shift(s) * V.hilbert_shift(s)
True

1.2. Univariate Ore polynomials
We check that shifting by an element and then by its opposite gives back the initial Ore polynomial:

```python
sage: # needs sage.rings.finite_rings
sage: P = S.random_element(degree=Integer(10))
sage: s = k.random_element()
sage: P.hilbert_shift(s).hilbert_shift(-s) == P
True
```

```
list (copy=True)
Return a list of the coefficients of self.

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: l = a.list(); l
[t^2 + 1, 0, t + 1, 0, 1]
```
```
>>> from sage.all import *
```
```
```python
R = QQ['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> a = Integer(1) + x**Integer(4) + (t+Integer(1))*x**Integer(2) +...
˓→t**Integer(2)
>>> l = a.list(); l
[t^2 + 1, 0, t + 1, 0, 1]
```
```
```
```
Note that \( l \) is a list, it is mutable, and each call to the list method returns a new list:
```
```
```python
sage: type(l)
<... 'list'>
sage: l[0] = 5
sage: a.list()
[t^2 + 1, 0, t + 1, 0, 1]
```
```
>>> from sage.all import *
```
```
```python
>>> type(l)
<... 'list'>
>>> l[Integer(0)] = Integer(5)
```
```
Noncommutative Polynomials, Release 10.4

\begin{verbatim}
>>> a.list()
[t^2 + 1, 0, t + 1, 0, 1]
\end{verbatim}

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

Return the polynomial resulting from discarding all monomials of degree at least \(n\).

**EXAMPLES:**

\begin{verbatim}
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
>>> from sage.all import *
\end{verbatim}

**valuation**

Return the minimal degree of a non-zero monomial of \(self\).

By convention, the zero Ore polynomial has valuation \(+\infty\).

**EXAMPLES:**

\begin{verbatim}
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
>>> from sage.all import *
\end{verbatim}

By convention, the valuation of 0 is \(+\infty\):

\begin{verbatim}
sage: S(0).valuation()
+Infinity
\end{verbatim}
1.3 Univariate skew polynomial rings

This module provides the `SkewPolynomialRing`. In the class hierarchy in Sage, the locution `Skew Polynomial` is used for a Ore polynomial without twisting derivation.

This module also provides:

- the class `SkewPolynomialRing_finite_order`, which is a specialized class for skew polynomial rings over fields equipped with an automorphism of finite order. It inherits from `SkewPolynomialRing` but contains more methods and provides better algorithms.

- the class `SkewPolynomialRing_finite_field`, which is a specialized class for skew polynomial rings over finite fields.

See also:

`OrePolynomialRing`

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
>>> from sage.all import *
>>> S(Integer(0)).valuation()
+Infinity
```

```python
class sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialCenterInjection
Bases: Section

Section of the canonical injection of the center of a skew polynomial ring into this ring.

class sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialCenterInjection(domain, codomain, embed, order)

Bases: RingHomomorphism

Canonical injection of the center of a skew polynomial ring into this ring.

section()

Return a section of this morphism.

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^3)
sage: S.<x> = SkewPolynomialRing(k, k.frobenius_endomorphism())
sage: Z = S.center()
sage: iota = S.convert_map_from(Z)
sage: sigma = iota.section()
```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(5)^3, names=('a',)); (a,) = k._first_ngens(1)
>>> S = SkewPolynomialRing(k, k.frobenius_endomorphism(), names=('x',)); (x,)
>>> Z = S.center()
>>> iota = S.convert_map_from(Z)
>>> sigma = iota.section()
>>> sigma(x**3)

1.3. Univariate skew polynomial rings
**lagrange_polynomial**(points)

Return the minimal-degree polynomial which interpolates the given points.

More precisely, given *n* pairs \((x_1, y_1), \ldots, (x_n, y_n)\) \(\in \mathbb{R}^2\), where \(R\) is \(self.base_ring()\), 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 \(self.twisting_morphism()\).

If the \(x_i\) are linearly independent over the fixed field of \(self.twisting_morphism()\) 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 \(self\); the \(x_i\) should be linearly independent over the fixed field of \(self.twisting_morphism()\)

**OUTPUT:**

The Lagrange polynomial.

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x, Frob]
```

```python
sage: points = [(t, 3*t^2 + 4*t + 4), (t^2, 4*t)]
sage: d = S.lagrange_polynomial(points); d
\(x + t\)
```

```python
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
```

```python
sage: p.multi_point_evaluation([1, t, t^2]) == [t^2 + 3*t + 4, 2*t^2 + 3*t + 1, t^2 + 3*t + 4]
```

```python
True
```

```python
>>> from sage.all import *
```

```python
>>> # needs sage.rings.finite_rings
```

```python
>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
```

```python
>>> Frob = k.frobenius_endomorphism()
```

```python
>>> S = k['x', Frob]; (x,) = S._first_ngens(1)
```

```python
>>> points = [(t, Integer(3)*t**Integer(2) + Integer(4)*t + Integer(4)), (t^2, 4*t)]
```

```python
>>> d = S.lagrange_polynomial(points); d
\(x + t\)
```

```python
>>> R = ZZ['t']; (t,) = R._first_ngens(1)
```

```python
>>> sigma = R.hom([t + Integer(1)])
```

```python
>>> T = R['x', sigma]; (x,) = T._first_ngens(1)
```

```python
>>> points = [(Integer(1), t**Integer(2) + Integer(3)*t + Integer(4)), (t, Integer(3)*t + Integer(4))]
```

```python
>>> p = T.lagrange_polynomial(points); p
```

(continues on next page)
If the $x_i$ are linearly dependent over the fixed field of `self.twisting_morphism()`, then an error is raised:

```python
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 twisting morphism, so a Lagrange polynomial could not be determined (and might not exist)
```

```python
>>> from sage.all import *

```n
```python
sage: T.lagrange_polynomial([(t, Integer(1)), (Integer(2)*t, Integer(3))])
Traceback (most recent call last):
  ...  
ValueError: the given evaluation points are linearly dependent over the fixed field of the twisting morphism, so a Lagrange polynomial could not be determined (and might not exist)
```

### 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.twisting_morphism()`.

- `eval_pts` – list of evaluation points which are linearly independent over the fixed field of the twisting morphism of the associated skew polynomial ring

**OUTPUT:**

The minimal vanishing polynomial.

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
```

```python
sage: Frob = k.frobenius_endomorphism()
```

```python
sage: S.<x> = k[x, Frob]
```

```python
sage: eval_pts = [1, t, t^2]
```

```python
sage: b = S.minimal_vanishing_polynomial(eval_pts); b
x^3 + 4
```

```python
>>> from sage.all import *

```n
```python
>>> from sage.all import *

```n
```python
k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
```

```python
>>> Frob = k.frobenius_endomorphism()
```

```python
>>> S = k['x', Frob]; (x,) = S._first_ngens(1)
```

```python
>>> eval_pts = [Integer(1), t, t^2]
```

```python
>>> b = S.minimal_vanishing_polynomial(eval_pts); b
x^3 + 4
```

The minimal vanishing polynomial evaluates to 0 at each of the evaluation points:
If the evaluation points are linearly dependent over the fixed field of the twisting morphism, then the returned polynomial has lower degree than the number of evaluation points:

```python
>>> S.minimal_vanishing_polynomial([t])
# needs sage.rings.finite_rings
x + 3*t^2 + 3*t
```

```python
>>> S.minimal_vanishing_polynomial([t, 3*t])
# needs sage.rings.finite_rings
x + 3*t^2 + 3*t
```

### Class: `sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_finite_field`

Bases: `SkewPolynomialRing_finite_order`

A specialized class for skew polynomial rings over finite fields.

See also:

- `sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing`
- `sage.rings.polynomial.skew_polynomial_finite_field`

Todo: Add methods related to center of skew polynomial ring, irreducibility, karatsuba multiplication and factorization.
class sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_finite_order (base_ring, morphism, derivation, name, sparse, category=None)

Bases: SkewPolynomialRing

A specialized class for skew polynomial rings whose twising morphism has finite order.

See also:

• sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing
• sage.rings.polynomial.skew_polynomial_finite_order

center (name=None, names=None, default=False)

Return the center of this skew polynomial ring.

Note: If $F$ denotes the subring of $R$ fixed by $\sigma$ and $\sigma$ has order $r$, the center of $K[x, \sigma]$ is $F[x^r]$, that is a univariate polynomial ring over $F$.

INPUT:

• name -- a string or None (default: None); the name for the central variable (namely $x^r$)
• default -- a boolean (default: False); if True, set the default variable name for the center to name

EXAMPLES:

```sage
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]; S
Ore Polynomial Ring in x over Finite Field in t of size 5^3
twisted by t |--> t^5
sage: Z = S.center(); Z
Univariate Polynomial Ring in z over Finite Field of size 5
sage: Z.gen()
z
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x',Frob]; (x,) = S._first_ngens(1); S
Ore Polynomial Ring in x over Finite Field in t of size 5^3
twisted by t |--> t^5
>>> Z = S.center(); Z
Univariate Polynomial Ring in z over Finite Field of size 5
>>> Z.gen()
z
```
We can pass in another variable name:

```
sage: S.center(name='y')  # needs sage.rings.finite_rings
Univariate Polynomial Ring in y over Finite Field of size 5
```

```
>>> from sage.all import *

sage: S.center(name='y')  # needs sage.rings.finite_rings
Univariate Polynomial Ring in y over Finite Field of size 5
```

or use the bracket notation:

```
sage: Zy.<y> = S.center(); Zy  # needs sage.rings.finite_rings
Univariate Polynomial Ring in y over Finite Field of size 5
```

```
sage: y.parent() is Zy  # needs sage.rings.finite_rings
True
```

A coercion map from the center to the skew polynomial ring is set:

```
sage: # needs sage.rings.finite_rings
sage: S.has_coerce_map_from(Zy)  # needs sage.rings.finite_rings
True
```

```
sage: P = y + x; P
x^3 + x
```

```
sage: P.parent() is S  # needs sage.rings.finite_rings
True
```

```
sage: Zy(x^6 + 2*x^3 + 3)  # needs sage.rings.finite_rings
```

```
>>> from sage.all import *

sage: Zy(x^6 + 2*x^3 + 3)  # needs sage.rings.finite_rings
```

```
>>> from sage.all import *

sage: Zy(x^6 + 2*x^3 + 3)  # needs sage.rings.finite_rings
```

(continues on next page)
Two different skew polynomial rings can share the same center:

```python
sage: S1.<x1> = k['x1', Frob]  # needs sage.rings.finite_rings
sage: S2.<x2> = k['x2', Frob]  # needs sage.rings.finite_rings
sage: S1.center() is S2.center()  # needs sage.rings.finite_rings
True
```

```python
>>> from sage.all import *
>>> S1 = k['x1', Frob]; (x1,) = S1._first_ngens(1)  # needs sage.rings.finite_rings
>>> S2 = k['x2', Frob]; (x2,) = S2._first_ngens(1)  # needs sage.rings.finite_rings
>>> S1.center() is S2.center()  # needs sage.rings.finite_rings
True
```

### About the default name of the central variable

A priori, the default is \( z \).

However, a variable name is given the first time this method is called, the given name become the default for the next calls:

```python
sage: # needs sage.rings.finite_rings
sage: K.<t> = GF(11^3)
sage: phi = K.frobenius_endomorphism()
sage: A.<X> = K['X', phi]
sage: C.<u> = A.center()  # first call
sage: C
Univariate Polynomial Ring in u over Finite Field of size 11
sage: A.<u> = A.center()  # second call: the variable name is still u
sage: A
Univariate Polynomial Ring in u over Finite Field of size 11
```

(continues on next page)
We can update the default variable name by passing in the argument default=True:

```
sage: D.<v> = A.center(default=True)
sage: D
Univariate Polynomial Ring in v over Finite Field of size 11
sage: A.center()  # second call: the variable name is still u
Univariate Polynomial Ring in u over Finite Field of size 11
sage: A.center() is D
True
```

### 1.4 Univariate skew polynomials

This module provides the `SkewPolynomial`. In the class hierarchy in Sage, the locution `Skew Polynomial` is used for a Ore polynomial without twisting derivation.

**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.SkewPolynomial_generic_dense

Bases: OrePolynomial_generic_dense

Generic implementation of dense skew polynomial supporting any valid base ring and twisting morphism.

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 twisting morphism to each of its coefficients.

INPUT:

• \(n\) – an integer, the power of conjugation

EXAMPLES:

```sage
R.<t> = QQ[]
K = R.fraction_field()
sigma = K.hom([1 + 1/t])
S.<x> = K[x,sigma]
a = t*x^3 + (t^2 + 1)*x^2 + 2*t
b = a.conjugate(2); b
(\frac{2*t + 1}{t + 1})x^3 + \frac{5*t^2 + 6*t + 2}{t^2 + 2*t + 1}x^2 + \frac{4*t + 2}{t + 1}
```

In principle, negative values for \(n\) are allowed, but Sage needs to be able to invert the twisting morphism:

```sage
b = a.conjugate(2); b
```

```
Traceback (most recent call last):
...  
NotImplementedError: inverse not implemented for morphisms of Fraction Field of Univariate Polynomial Ring in t over Rational Field
```
```python
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: T.<y> = k[y,Frob]
sage: u = T.random_element()
sage: v = u.conjugate(-1)
sage: u*y == y*v
True
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> T = k['y',Frob]; (y,) = T._first_ngens(1)
>>> u = T.random_element()
>>> v = u.conjugate(Integer(-1))
>>> u*y == y*v
True
```

### 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:**

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

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x',Frob]; (x,) = S._first_ngens(1)
>>> a = x + t
>>> modulus = x**Integer(3) + t*x**Integer(2) + (t+Integer(3))*x - Integer(2)
>>> a.left_power_mod(Integer(100),modulus)
(4*t^2 + t + 1)*x^2 + (t^2 + 4*t + 1)*x + 3*t^2 + 3*t
```
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 minimal_vanishing_polynomial().

EXAMPLES:

```sage
# needs sage.rings.finite_rings
k.<t> = GF(5^3)
Frob = k.frobenius_endomorphism()
S.<x> = k['x'],Frob
a = x + t
eval_pts = [1, t, t^2]
c = a.multi_point_evaluation(eval_pts); c
[t + 1, 3*t^2 + 4*t + 4, 4*t]
c == [ a(e) for e in eval_pts ]
True
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x'],Frob; (x,) = S._first_ngens(1)
>>> a = x + t
>>> eval_pts = [Integer(1), t, t**Integer(2)]
>>> c = a.multi_point_evaluation(eval_pts); c
[t + 1, 3*t^2 + 4*t + 4, 4*t]
>>> c == [ a(e) for e in eval_pts ]
True
```

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
# needs sage.rings.finite_rings
k.<t> = GF(5^3)
Frob = k.frobenius_endomorphism()
T.<x> = k['x'],Frob
a = 3*t^2*x^2 + (t + 1)*x + 2
```
Evaluation points outside the base ring is usually not possible due to the twisting morphism:

```
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
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> T = k['x',Frob]; (x,) = T._first_ngens(1)
>>> a = Integer(3)*t**Integer(2)*x**Integer(2) + (t + Integer(1))*x +
   Integer(2)
>>> a(t) #indirect test
2*t^2 + 2*t + 3
>>> a.operator_eval(t)
2*t^2 + 2*t + 3
```

The quotient of the underlying skew polynomial ring by the principal ideal generated by \( \text{modulus} \) is in general not a ring.

**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 self**exp and then reduce it modulo modulus.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x,Frob]
sage: a = x + t
sage: b = a^5  # indirect doctest
sage: b
x^5 + (2*t^2 + 4)*x^4 + (t^2 + 2)*x^3 + 2*x^2 + (4*t^2 + 2)*x + 2*t^2 + 4*t + 4
sage: b == a * a * a * a * a
True
sage: modulus = x^3 + t*x^2 + (t+3)*x - 2
sage: br = a.right_power_mod(5, modulus); br
(t + 1)*x^2 + (2*t^2 + t + 1)*x + 2*t^2 + 4*t + 2
sage: br == b % modulus
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
```

Negative exponents are supported:

```
sage: # needs sage.rings.finite_rings
sage: a^(-5) (x^5 + (2*t^2 + 4)*x^4 + (t^2 + 2)*x^3 + 2*x^2 + (4*t^2 + 2)*x + 2*t^2 + 4*t + 4)^(-1)
sage: b * a^(-5)
1
```

However, they cannot be combined with modulus:

```
sage: a.right_power_mod(-10, modulus)  # needs sage.rings.finite_rings
Traceback (most recent call last):
  ... ValueError: modulus cannot be combined with negative exponent
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x',Frob]; (x,) = S._first_ngens(1)
>>> a = x + t
>>> b = a**Integer(5)  # indirect doctest
>>> b
x^5 + (2*t^2 + 4)*x^4 + (t^2 + 2)*x^3 + 2*x^2 + (4*t^2 + 2)*x + 2*t^2 + 4*t + 4
>>> b == a * a * a * a * a
True
>>> modulus = x**Integer(3) + t*x**Integer(2) + (t+Integer(3))*x - Integer(2)
>>> br = a.right_power_mod(Integer(5), modulus); br
(t + 1)*x^2 + (2*t^2 + t + 1)*x + 2*t^2 + 4*t + 2
>>> br == b % modulus
True
>>> a.right_power_mod(Integer(100), modulus)
(2*t^2 + 3)*x^2 + (t^2 + 4*t + 2)*x + t^2 + 2*t + 1
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x',Frob]; (x,) = S._first_ngens(1)
>>> a = x + t
>>> b = a**Integer(5)  # indirect doctest
>>> b
x^5 + (2*t^2 + 4)*x^4 + (t^2 + 2)*x^3 + 2*x^2 + (4*t^2 + 2)*x + 2*t^2 + 4*t + 4
>>> b == a * a * a * a * a
True
>>> modulus = x**Integer(3) + t*x**Integer(2) + (t+Integer(3))*x - Integer(2)
>>> br = a.right_power_mod(Integer(5), modulus); br
(t + 1)*x^2 + (2*t^2 + t + 1)*x + 2*t^2 + 4*t + 2
>>> br == b % modulus
True
>>> a.right_power_mod(Integer(100), modulus)
(2*t^2 + 3)*x^2 + (t^2 + 4*t + 2)*x + t^2 + 2*t + 1
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x',Frob]; (x,) = S._first_ngens(1)
>>> a = x + t
>>> b = a**Integer(5)  # indirect doctest
>>> b
x^5 + (2*t^2 + 4)*x^4 + (t^2 + 2)*x^3 + 2*x^2 + (4*t^2 + 2)*x + 2*t^2 + 4*t + 4
>>> b == a * a * a * a * a
True
>>> modulus = x**Integer(3) + t*x**Integer(2) + (t+Integer(3))*x - Integer(2)
>>> br = a.right_power_mod(Integer(5), modulus); br
(t + 1)*x^2 + (2*t^2 + t + 1)*x + 2*t^2 + 4*t + 2
>>> br == b % modulus
True
>>> a.right_power_mod(Integer(100), modulus)
(2*t^2 + 3)*x^2 + (t^2 + 4*t + 2)*x + t^2 + 2*t + 1
```
1.5 Univariate dense skew polynomials over a field with a finite order automorphism

AUTHOR:

- Xavier Caruso (2012-06-29): initial version
- Arpit Merchant (2016-08-04): improved docstrings, fixed doctests and refactored classes and methods

```python
# needs sage.rings.finite_rings
Traceback (most recent call last):
... ValueError: modulus cannot be combined with negative exponent
```

```python
class sage.rings.polynomial.skew_polynomial_finite_order.SkewPolynomial_finite_order_dense:
Bases: SkewPolynomial_generic_dense
This method constructs a generic dense skew polynomial over a field equipped with an automorphism of finite order.

INPUT:

- parent – parent of self
- x – list of coefficients from which self can be constructed
- check – flag variable to normalize the polynomial
- construct – boolean (default: False)

bound()

Return a bound of this skew polynomial (i.e. a multiple of this skew polynomial lying in the center).

Note: Since \( b \) is central, it divides a skew polynomial on the left iff it divides it on the right

ALGORITHM:

1. Sage first checks whether \( self \) is itself in the center. It if is, it returns \( self \)
2. If an optimal bound was previously computed and cached, Sage returns it
3. Otherwise, Sage returns the reduced norm of \( self \)

As a consequence, the output of this function may depend on previous computations (an example is given below).

EXAMPLES:

```python
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x', Frob]
sage: Z = S.center(); Z
Univariate Polynomial Ring in z over Finite Field of size 5
```
We observe that the bound is explicitly given as an element of the center (which is a univariate polynomial ring in the variable $z$). We can use conversion to send it in the skew polynomial ring:

```python
sage: S(b)
x^6 + x^3 + 4
```

We check that $b$ is divisible by $a$:

```python
sage: S(b).is_right_divisible_by(a)
True
sage: S(b).is_left_divisible_by(a)
True
```

Actually, $b$ is the reduced norm of $a$:

```python
sage: b == a.reduced_norm()
True
```

Now, we compute the optimal bound of $a$ and see that it affects the behaviour of `bound()`:

```python
sage: a.optimal_bound()
z + 3
sage: a.bound()
z + 3
```
is_central()
Return True if this skew polynomial lies in the center.

EXAMPLES:

```sage
defines k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x', Frob]
sage: x.is_central()
False
sage: (t*x^3).is_central()
False
sage: (x^6 + x^3).is_central()
True
```

optimal_bound()
Return the optimal bound of this skew polynomial (i.e. the monic multiple of this skew polynomial of minimal degree lying in the center).

Note: The result is cached.

EXAMPLES:

```sage
defines k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x', Frob]
sage: Z = S.center(); Z
Univariate Polynomial Ring in z over Finite Field of size 5
sage: a = x^2 + (4*t + 2)*x + 4*t^2 + 3
sage: b = a.optimal_bound(); b
z + 3
```
We observe that the bound is explicitly given as an element of the center (which is a univariate polynomial ring in the variable $z$). We can use conversion to send it in the skew polynomial ring:

```python
sage: S(b)
x^3 + 3
```

We check that $b$ is divisible by $a$:

```python
sage: S(b).is_right_divisible_by(a)
True
sage: S(b).is_left_divisible_by(a)
True
```

`reduced_charpoly` ($\text{var}=\text{None}$)

Return the reduced characteristic polynomial of this skew polynomial.

**INPUT:**

- `\text{var}` - a string, a pair of strings or `None` (default: `None`); the variable names used for the characteristic polynomial and the center

**Note:** The result is cached.

**EXAMPLES:**

```python
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<u> = k[u, Frob]
sage: a = u^3 + (2*t^2 + 3)*u^2 + (4*t^2 + t + 4)*u + 2*t^2 + 2
sage: chi = a.reduced_charpoly()
sage: chi
x^3 + (2*z + 1)*x^2 + (3*z^2 + 4*z)*x + 4*z^3 + z^2 + 1
```
Noncommutative Polynomials, Release 10.4

>>> Frob = k.frobenius_endomorphism()
>>> S = k['u', Frob]; (u,) = S._first_ngens(1)
>>> a = u**Integer(3) + (Integer(2)*t**Integer(2) + Integer(3))*u**Integer(2) →
˓→(Integer(4)*t**Integer(2) + t + Integer(4))*u + Integer(2)*t**Integer(2) →
˓→Integer(2)
>>> chi = a.reduced_charpoly()
>>> chi
x^3 + (2*z + 1)*x^2 + (3*z^2 + 4*z)*x + 4*z^3 + z^2 + 1

The reduced characteristic polynomial has coefficients in the center of $S$, which is itself a univariate polynomial ring in the variable $z = u^3$ over $\mathbb{F}_5$. Hence it appears as a bivariate polynomial:

```
sage: chi.parent()
Univariate Polynomial Ring in x over Univariate Polynomial Ring in z over \Finite Field of size 5
```

```
>>> from sage.all import *
>>> chi.parent()
Univariate Polynomial Ring in x over Univariate Polynomial Ring in z over \Finite Field of size 5
```

The constant coefficient of the reduced characteristic polynomial is the reduced norm, up to a sign:

```
sage: chi[0] == -a.reduced_norm()
True
```

```
>>> from sage.all import *
>>> chi[Integer(0)] == -a.reduced_norm()
True
```

Its coefficient of degree $\deg(a) - 1$ is the opposite of the reduced trace:

```
True
```

```
>>> from sage.all import *
>>> chi[Integer(2)] == -a.reduced_trace()
True
```

By default, the name of the variable of the reduced characteristic polynomial is $x$ and the name of central variable is usually $z$ (see center() for more details about this). The user can specify different names if desired:

```
sage: a.reduced_charpoly(var='T')  # variable name for the caracteristic
→polynomial
T^3 + (2*z + 1)*T^2 + (3*z^2 + 4*z)*T + 4*z^3 + z^2 + 1
```

```
sage: a.reduced_charpoly(var=('T', 'c'))
→polynomial
T^3 + (2*c + 1)*T^2 + (3*c^2 + 4*c)*T + 4*c^3 + c^2 + 1
```

```
>>> from sage.all import *
>>> a.reduced_charpoly(var='T')  # variable name for the caracteristic
→polynomial
T^3 + (2*z + 1)*T^2 + (3*z^2 + 4*z)*T + 4*z^3 + z^2 + 1
```

(continues on previous page)
>>> a.reduced_charpoly(var=('T', 'c'))
T^3 + (2*c + 1)*T^2 + (3*c^2 + 4*c)*T + 4*c^3 + c^2 + 1

See also:
reduced_trace(), reduced_norm()

reduced_norm(var=None)
Return the reduced norm of this skew polynomial.

INPUT:
- var -- a string or False or None (default: None); the variable name; if False, return the list of coefficients

Note: The result is cached.

EXAMPLES:

sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x, Frob]
sage: a = x^3 + (2*t^2 + 3)*x^2 + (4*t^2 + t + 4)*x + 2*t^2 + 2
sage: N = a.reduced_norm(); N
z^3 + 4*z^2 + 4

The reduced norm lies in the center of \( S \), which is a univariate polynomial ring in the variable \( z = x^3 \) over \( F_5 \):

sage: N.parent()
Univariate Polynomial Ring in z over Finite Field of size 5
sage: N.parent() is S.center()
True

We can use explicit conversion to view \( N \) as a skew polynomial:

sage: S(N)
x^9 + 4*x^6 + 4
By default, the name of the central variable is usually \( z \) (see \texttt{center()} for more details about this). However, the user can specify a different variable name if desired:

\[
\texttt{sage: a.reduced_norm(var='u')}
\]

\[
u^3 + 4*u^2 + 4
\]

When passing in \texttt{var=False}, a tuple of coefficients (instead of an actual polynomial) is returned:

\[
\texttt{sage: a.reduced_norm(var=False)}
\]

\[
(4, 0, 4, 1)
\]

\[
\texttt{>>> from sage.all import *}
\texttt{>>> a.reduced_norm(var=False)}
\]

\[
(4, 0, 4, 1)
\]

\[
\texttt{ALGORITHM:}
\]

If \( r (= \) the order of the twist map) is small compared to \( d (= \) the degree of this skew polynomial), the reduced norm is computed as the determinant of the multiplication by \( P (= \) this skew polynomial) acting on \( K[X, \sigma] (= \) the underlying skew ring) viewed as a free module of rank \( r \) over \( K[X^r]\).

Otherwise, the reduced norm is computed as the characteristic polynomial of the left multiplication by \( X \) on the quotient \( K[X, \sigma]/K[X, \sigma]P \) (which is a \( K \)-vector space of dimension \( d \)).

See also:

\texttt{reduced_trace()}, \texttt{reduced_charpoly()}

\texttt{reduced_trace}(\texttt{var=None})

Return the reduced trace of this skew polynomial.

INPUT:

\[
\bullet \text{ var – a string or False or None (default: None); the variable name; if False, return the list of coefficients}
\]

EXAMPLES:

\[
\texttt{sage: k.<t> = GF(5^3)}
\]

\[
\texttt{sage: Frob = k.frobenius_endomorphism()}
\]

\[
\texttt{sage: S.<x> = k['x', Frob]}
\]

\[
\texttt{sage: a = x^3 + (2*t^2 + 3)*x^2 + (4*t^2 + t + 4)*x + 2*t^2 + 2}
\]

\[
\texttt{sage: tr = a.reduced_trace(); tr}
\]

\[
3*z + 4
\]

\[
\texttt{>>> from sage.all import *}
\]

\[
\texttt{>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)}
\]

\[
\texttt{>>> Frob = k.frobenius_endomorphism()}
\]

\[
\texttt{>>> S = k['x', Frob]; (x,) = S._first_ngens(1)}
\]

\[
\texttt{>>> a = x**Integer(3) + (Integer(2)*t**Integer(2) + Integer(3))*x + 2*t^2 + 2}
\]

(continues on next page)
The reduced trace lies in the center of $S$, which is a univariate polynomial ring in the variable $z = x^3$ over $F_5$:

```python
sage: tr.parent()
Univariate Polynomial Ring in z over Finite Field of size 5
sage: tr.parent() is S.center()
True
```

We can use explicit conversion to view $tr$ as a skew polynomial:

```python
sage: S(tr)
3*x^3 + 4
```

By default, the name of the central variable is usually $z$ (see `center()` for more details about this). However, the user can specify a different variable name if desired:

```python
sage: a.reduced_trace(var='u')
3*u + 4
```

When passing in `var=False`, a tuple of coefficients (instead of an actual polynomial) is returned:

```python
sage: a.reduced_trace(var=False)
(4, 3)
```

See also:

`reduced_norm()`, `reduced_charpoly()`

1.5. Univariate dense skew polynomials over a field with a finite order automorphism 87
1.6 Univariate dense skew polynomials over finite fields

This module provides the class: `sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field_dense`, which constructs a single univariate skew polynomial over a finite field equipped with the Frobenius endomorphism. Among other things, it implements the fast factorization algorithm designed in [CL2017].

AUTHOR:
- Xavier Caruso (2012-06-29): initial version
- Arpit Merchant (2016-08-04): improved docstrings, fixed doctests and refactored classes and methods

```python
class sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field_dense
    Bases: SkewPolynomial_finite_order_dense

    count_factorizations()
    Return the number of factorizations (as a product of a unit and a product of irreducible monic factors) of this skew polynomial.

    EXAMPLES:
    sage: k.<t> = GF(5^3)
    sage: Frob = k.frobenius_endomorphism()
    sage: S.<x> = k[x,Frob]
    sage: a = x^4 + (4*t + 3)*x^3 + t^2*x^2 + (4*t^2 + 3*t)*x + 3*t
    sage: a.count_factorizations()
    216
    >>> from sage.all import *
    >>> Z = S.center(names=(x3,)); (x3,) = Z._first_ngens(1)
    >>> N = x3**5 + Integer(4)*x3**4 + Integer(4)*x3**2 + Integer(4)*x3 + Integer(3)
    >>> N.is_irreducible()
    True
    >>> S(N).count_factorizations()
    30537115626
```

We illustrate that an irreducible polynomial in the center have in general a lot of distinct factorizations in the skew polynomial ring:

```python
sage: Z.<x3> = S.center()
```

```python
sage: N = x3^5 + 4*x3^4 + 4*x3^2 + 4*x3 + 3
sage: N.is_irreducible()
True
sage: S(N).count_factorizations()
30537115626
```
**count_irreducible_divisors()**

Return the number of irreducible monic divisors of this skew polynomial.

**Note:** One can prove that there are always as many left irreducible monic divisors as right irreducible monic divisors.

**EXAMPLES:**

```sage
k.<t> = GF(5^3)
k.<x> = k['x',Frob]

>>> from sage.all import *
>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x',Frob]; (x,) = S._first_ngens(1)

We illustrate that a skew polynomial may have a number of irreducible divisors greater than its degree:

```sage
a = x^4 + (4*t + 3)*x^3 + t^2*x^2 + (4*t^2 + 3*t)*x + 3*t
a.count_irreducible_divisors()
```

```sage
12
```

We illustrate that an irreducible polynomial in the center have in general a lot of irreducible divisors in the skew polynomial ring:

```sage
Z.<x3> = S.center()
N = x3^5 + 4*x3^4 + 4*x3^2 + 4*x3 + 3; N
N.is_irreducible()
S(N).count_irreducible_divisors()
```

```sage
9768751
```

**factor (uniform=False)**

Return a factorization of this skew polynomial.

**INPUT:**
• **uniform** – a boolean (default: False); whether the output irreducible divisor should be uniformly distributed among all possibilities

**EXAMPLES:**

```sage
ek.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = x^3 + (t^2 + 4*t + 2)*x^2 + (3*t + 3)*x + t^2 + 1
sage: F = a.factor(); F  # random
(x + t^2 + 4) * (x + t + 3) * (x + t)
sage: F.value() == a
True

>>> from sage.all import *
>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
>>> S = k['x',Frob]; (x,) = S._first_ngens(1)
>>> a = x**Integer(3) + (t**Integer(2) + Integer(4)*t + Integer(2))*x**Integer(2) + (Integer(3)*t + Integer(3))*x + t**Integer(2) + Integer(1)
>>> F = a.factor(); F  # random
(x + t^2 + 4) * (x + t + 3) * (x + t)
>>> F.value() == a
True
```

The result of the factorization is cached. Hence, if we try again to factor *a*, we will get the same answer:

```sage
sage: a.factor()  # random
(x + t^2 + 4) * (x + t + 3) * (x + t)
```

However, the algorithm is probabilistic. Hence if we first reinitialize *a*, we may get a different answer:

```sage
sage: a = x^3 + (t^2 + 4*t + 2)*x^2 + (3*t + 3)*x + t^2 + 1
sage: F = a.factor(); F  # random
(x + t^2 + t + 2) * (x + 2*t^2 + t + 4) * (x + t)
sage: F.value() == a
True

>>> from sage.all import *
>>> a = x**Integer(3) + (t**Integer(2) + Integer(4)*t + Integer(2))*x**Integer(2) + (Integer(3)*t + Integer(3))*x + t**Integer(2) + Integer(1)
>>> F = a.factor(); F  # random
(x + t^2 + t + 2) * (x + 2*t^2 + t + 4) * (x + t)
>>> F.value() == a
True
```

There is a priori no guarantee on the distribution of the factorizations we get. Passing in the keyword `uniform=True` ensures the output is uniformly distributed among all factorizations:

```sage
sage: a.factor(uniform=True)  # random
(x + t^2 + 4) * (x + t) * (x + t + 3)
```

(continues on next page)
By convention, the zero skew polynomial has no factorization:

\[ \text{sage: } S(0).\text{factor()} \]
Traceback (most recent call last):
...
ValueError: factorization of 0 not defined

factorizations()

Return an iterator over all factorizations (as a product of a unit and a product of irreducible monic factors) of this skew polynomial.

EXAMPLES:

\[ \text{sage: } k.<t> = GF(5^3) \]
\[ \text{sage: } \text{Frob} = k\text{.frobenius\_endomorphism()} \]
\[ \text{sage: } S.<x> = k[x,\text{Frob}] \]
\[ \text{sage: } a = x^3 + (t^2 + 1)*x^2 + (2*t + 3)*x + t^2 + t + 2 \]
\[ \text{sage: } \text{iter} = a\text{.factorizations()}; \text{iter} \]
<...generator object at 0x...>
\[ \text{sage: } \text{next(iter)} \] # random
\[ (x + 3*t^2 + 4*t) * (x + 2*t^2) * (x + 4*t^2 + 4*t + 2) \]
\[ \text{sage: } \text{next(iter)} \] # random
\[ (x + 3*t^2 + 4*t) * (x + 3*t^2 + 2*t + 2) * (x + 4*t^2 + t + 2) \]
We can use this function to build the list of factorizations of $a$:

```python
sage: factorizations = [ F for F in a.factorizations() ]
```

We do some checks:

```python
sage: len(factorizations) == a.count_factorizations()
True
sage: len(factorizations) == Set(factorizations).cardinality()  # check no__
    --> duplicates
True
sage: for F in factorizations:
    ....:    assert F.value() == a, "factorization has a different value"
    ....:    for d, _ in F:
    ....:        assert d.is_irreducible(), "a factor is not irreducible"
```

Note that the algorithm used in this method is probabilistic. As a consequence, if we call it two times with
the same input, we can get different orderings:

```python
>>> from sage.all import *
>>> factorizations2 = [ F for F in a.factorizations() ]
>>> factorizations == factorizations2  # random
False
>>> sorted(factorizations) == sorted(factorizations2)
True
```

```
is_irreducible()
```

Return True if this skew polynomial is irreducible.

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
sage: a.is_irreducible()
(continues on next page)```
False
\begin{verbatim}
sage: a.factor()
\((x + 4*t^2 + 4*t + 1) \times (x + 3*t + 2)\)
\end{verbatim}
\begin{verbatim}
sage: a = x^2 + t*x + t + 1
sage: a.is_irreducible()
True
sage: a.factor()
x^2 + t*x + t + 1
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)

>>> Frob = k.frobenius_endomorphism()

>>> S = k[x,Frob]; (x,) = S._first_ngens(1)

>>> a = x**Integer(2) + t*x + Integer(1)

>>> a.is_irreducible()
False

>>> a.factor()
\((x + 4*t^2 + 4*t + 1) \times (x + 3*t + 2)\)

>>> a = x**Integer(2) + t*x + t + Integer(1)

>>> a.is_irreducible()
True

>>> a.factor()
x^2 + t*x + t + 1
\end{verbatim}

Skew polynomials of degree 1 are of course irreducible:
\begin{verbatim}
sage: a = x + t
sage: a.is_irreducible()
True
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> a = x + t

>>> a.is_irreducible()
True
\end{verbatim}

A random irreducible skew polynomial is irreducible:
\begin{verbatim}
sage: a = S.random_irreducible(degree=4,monic=True); a  # random
\(x^4 + (t + 1)*x^3 + (3*t^2 + 2*t + 3)*x^2 + 3*t*x + 3*t\)
sage: a.is_irreducible()
True
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> a = S.random_irreducible(degree=Integer(4),monic=True); a  # random
\(x^4 + (t + 1)*x^3 + (3*t^2 + 2*t + 3)*x^2 + 3*t*x + 3*t\)

>>> a.is_irreducible()
True
\end{verbatim}

By convention, constant skew polynomials are not irreducible:
\begin{verbatim}
sage: S(1).is_irreducible()
False
\end{verbatim}
left_irreducible_divisor\(\) (\texttt{uniform=False})

Return a left irreducible divisor of this skew polynomial.

**INPUT:**

- \texttt{uniform} – a boolean (default: \texttt{False}); whether the output irreducible divisor should be uniformly distributed among all possibilities

**EXAMPLES:**

```python
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x,Frob]
sage: a = x^6 + 3*t*x^5 + (3*t + 1)*x^3 + (4*t^2 + 3*t + 4)*x^2 + (t^2 + 2)*x + 4*t^2 + 3*t + 3
sage: dl = a.left_irreducible_divisor(); dl
# random
x^3 + (t^2 + t + 2)*x^2 + (t + 2)*x + 3*t^2 + t + 4
sage: a.is_left_divisible_by(dl)
True
```

The algorithm is probabilistic. Hence, if we ask again for a left irreducible divisor of \(a\), we may get a different answer:

```python
sage: a.left_irreducible_divisor()  # random
x^3 + (4*t + 3)*x^2 + (2*t^2 + 3*t + 4)*x + 4*t^2 + 2*t
```

We can also generate uniformly distributed irreducible monic divisors as follows:

```python
sage: a.left_irreducible_divisor(\texttt{uniform=True})  # random
x^3 + (4*t^2 + 3*t + 4)*x^2 + (t^2 + t + 3)*x + 2*t^2 + 3
```
sage: a.left_irreducible_divisor(uniform=True)  # random
x^3 + (2*t^2 + t + 4)*x^2 + (2*t^2 + 4*t + 4)*x + 2*t + 3
sage: a.left_irreducible_divisor(uniform=True)  # random
x^3 + (t^2 + t + 2)*x^2 + (3*t^2 + t)*x + 2*t + 1

By convention, the zero skew polynomial has no irreducible divisor:

sage: S(0).left_irreducible_divisor()
Traceback (most recent call last):
  ... ValueError: 0 has no irreducible divisor

left_irreducible_divisors()
Return an iterator over all irreducible monic left divisors of this skew polynomial.

EXAMPLES:

sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x,Frob]
sage: a = x^4 + 2*t*x^3 + 3*t^2*x^2 + (t^2 + t + 1)*x + 4*t + 3
sage: iter = a.left_irreducible_divisors(); iter
<...generator object at 0x...>
sage: next(iter)  # random
x + 3*t + 3
sage: next(iter)  # random
x + 4*t + 2

1.6. Univariate dense skew polynomials over finite fields 95
We can use this function to build the list of all monic irreducible divisors of $a$:

```
sage: leftdiv = [ d for d in a.left_irreducible_divisors() ]
```

Note that the algorithm is probabilistic. As a consequence, if we build again the list of left monic irreducible divisors of $a$, we may get a different ordering:

```
sage: leftdiv2 = [ d for d in a.left_irreducible_divisors() ]
```

```
sage: Set(leftdiv) == Set(leftdiv2)
True
```

### right_irreducible_divisor ($\text{uniform}=\text{False}$)

Return a right irreducible divisor of this skew polynomial.

**INPUT:**

- `uniform` – a boolean (default: False); whether the output irreducible divisor should be uniformly distributed among all possibilities

**EXAMPLES:**

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x,Frob]
sage: a = x^6 + 3*t*x^5 + (3*t + 1)*x^3 + (4*t^2 + 3*t + 4)*x^2 + (t^2 + 2)*x + 4*t^2 + 3*t + 3
```

```
sage: dr = a.right_irreducible_divisor(); dr  # random
x^3 + (2*t^2 + t + 4)*x^2 + (4*t + 1)*x + 4*t^2 + t + 1
```

```
sage: a.is_right_divisible_by(dr)
True
```

Right divisors are cached. Hence, if we ask again for a right divisor, we will get the same answer:

```
sage: a.right_irreducible_divisor()  # random
x^3 + (2*t^2 + t + 4)*x^2 + (4*t + 1)*x + 4*t^2 + t + 1
```
However the algorithm is probabilistic. Hence, if we first reinitialize \( a \), we may get a different answer:

```python
sage: a = x^6 + 3*t*x^5 + (3*t + 1)*x^3 + (4*t^2 + 3*t + 4)*x^2 + (t^2 + 2)*x + 4*t^2 + 3*t + 3
sage: a.right_irreducible_divisor()  # random
x^3 + (t^2 + 3*t + 4)*x^2 + (t + 2)*x + 4*t^2 + t + 1
```

We can also generate uniformly distributed irreducible monic divisors as follows:

```python
sage: a.right_irreducible_divisor(uniform=True)  # random
x^3 + (4*t + 2)*x^2 + (2*t^2 + 2*t + 2)*x + 2*t^2 + 2
sage: a.right_irreducible_divisor(uniform=True)  # random
x^3 + (t^2 + 2)*x^2 + (3*t^2 + 1)*x + 4*t^2 + 2*t
sage: a.right_irreducible_divisor(uniform=True)  # random
x^3 + x^2 + (4*t^2 + 2*t + 4)*x + t^2 + 3
```

By convention, the zero skew polynomial has no irreducible divisor:

```python
sage: S(0).right_irreducible_divisor()
Traceback (most recent call last):
  ...
ValueError: 0 has no irreducible divisor
```

```python
sage: S(Integer(0)).right_irreducible_divisor()
Traceback (most recent call last):
  ...
ValueError: 0 has no irreducible divisor
```

**right_irreducible_divisors()**

Return an iterator over all irreducible monic right divisors of this skew polynomial.

**EXAMPLES:**

```python
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
```
We can use this function to build the list of all monic irreducible divisors of $a$:

```python
sage: rightdiv = [ d for d in a.right_irreducible_divisors() ]
```

Note that the algorithm is probabilistic. As a consequence, if we build again the list of right monic irreducible divisors of $a$, we may get a different ordering:

```python
sage: rightdiv2 = [ d for d in a.right_irreducible_divisors() ]
sage: rightdiv == rightdiv2
False
sage: Set(rightdiv) == Set(rightdiv2)
True
```

---

**type** ($N$)

Return the $N$-type of this skew polynomial (see definition below).

**INPUT:**

- $N$ – an irreducible polynomial in the center of the underlying skew polynomial ring

**Note:** The result is cached.
DEFINITION:

The $N$-type of a skew polynomial $a$ is the Partition $(t_0, t_1, t_2, \ldots)$ defined by

$$t_0 + \cdots + t_i = \frac{\deg \gcd(a, N^i)}{\deg N},$$

where $\deg N$ is the degree of $N$ considered as an element in the center.

This notion has an important mathematical interest because it corresponds to the Jordan type of the $N$-typical part of the associated Galois representation.

EXAMPLES:

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x,Frob]
sage: Z = S.center(); x3 = Z.gen()

sage: a = x^4 + x^3 + (4*t^2 + 4)*x^2 + (t^2 + 2)*x + 4*t^2
sage: N = x3^2 + x3 + 1
sage: a.type(N)
[1]
sage: N = x3 + 1
sage: a.type(N)
[2]

sage: a = x^3 + (3*t^2 + 1)*x^2 + (3*t^2 + t + 1)*x + t + 1
sage: N = x3 + 1
sage: a.type(N)
[2, 1]
```

If $N$ does not divide the reduced map of $a$, the type is empty:

```
sage: N = x3 + 2
sage: a.type(N)
[]
```
If $a = N$, the type is just $[r]$ where $r$ is the order of the twisting morphism $\text{Pro}B$:

```
sage: N = x^3^2 + x^3 + 1
sage: S(N).type(N)
[3]
```

$N$ must be irreducible:

```
sage: N = (x^3 + 1) * (x^3 + 2)
sage: a.type(N)
Traceback (most recent call last):
  ...  
ValueError: N is not irreducible
```

1.7 Fraction fields of Ore polynomial rings

Sage provides support for building the fraction field of any Ore polynomial ring and performing basic operations in it. The fraction field is constructed by the method `sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing.fraction_field()` as demonstrated below:

```
sage: R.<t> = QQ[]
sage: der = R.derivation()
sage: A.<d> = R['d', der]
sage: K = A.fraction_field()
sage: K
Ore Function Field in d over Fraction Field of Univariate Polynomial Ring in t over Rational Field twisted by d/dt
```

```
>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> der = R.derivation()
>>> A = R['d', der]; (d,) = A._first_ngens(1)
>>> K = A.fraction_field()
>>> K
Ore Function Field in d over Fraction Field of Univariate Polynomial Ring in t over Rational Field twisted by d/dt
```
The simplest way to build elements in $K$ is to use the division operator over Ore polynomial rings:

```python
sage: f = 1/d
sage: f
```

```plaintext
d^(-1)
```

```python
sage: f.parent() is K
True
```

```python
>>> from sage.all import *

>>> f = Integer(1)/d

>>> f
```

```plaintext
(d - 1/t)^(-1) * t
```

```python
>>> from sage.all import *

>>> g = t / d

>>> g
```

```plaintext
(d - 1/t)^(-1) * t
```

Representations of Elements:

Elements in $K$ are internally represented by fractions of the form $s^{-1}t$ with the denominator on the left. Notice that, because of noncommutativity, this is not the same that fractions with denominator on the right. For example, a fraction created by the division operator is usually displayed with a different numerator and/or a different denominator:

```python
sage: g = t / d
sage: g
```

```plaintext
(t - 1/d)^(-1) * d
```

The left numerator and right denominator are accessible as follows:

```python
sage: g.left_numerator()
```

```python
t
```

```python
sage: g.right_denominator()
```

```python
d
```

Similarly, the methods `OrePolynomial.left_denominator()` and `OrePolynomial.right_numerator()` give access to the Ore polynomials $s$ and $t$ in the representation $s^{-1}t$:

```python
sage: g.left_denominator()
```

```python
d - 1/t
```

```python
sage: g.right_numerator()
```

```python
t
```
We favored the writing $s^{-1}t$ because it always exists. On the contrary, the writing $st^{-1}$ is only guaranteed when the twisting morphism defining the Ore polynomial ring is bijective. As a consequence, when the latter assumption is not fulfilled (or actually if Sage cannot invert the twisting morphism), computing the left numerator and the right denominator fails:

```
sage: # needs sage.rings.function_field
sage: sigma = R.hom([t^2])
sage: S.<x> = R['x', sigma]
sage: f = F.random_element()
sage: while not f:
    ....: f = F.random_element()
sage: f.left_numerator()
Traceback (most recent call last):
... 
NotImplementedError: inversion of the twisting morphism
```

On a related note, fractions are systematically simplified when the twisting morphism is bijective but they are not otherwise. As an example, compare the two following computations:

```
>>> from sage.all import *
>>> # needs sage.rings.function_field
>>> sigma = R.hom([t^Integer(2)])
>>> S = R['x', sigma]; (x,) = S._first_ngens(1)
>>> F = S.fraction_field()
>>> f = F.random_element()
>>> while not f:
    ...  f = F.random_element()
>>> f.left_numerator()
Traceback (most recent call last):
... 
NotImplementedError: inversion of the twisting morphism
```

```
sage: # needs sage.rings.function_field
sage: P = d^2 + t*d + 1
sage: Q = d + t^2
sage: D = d^3 + t^2 + 1
sage: f = P^(-1) * Q
sage: (d^2 + t*d + 1)^(-1) * (d + t^2)
sage: g = (D*P)^(-1) * (D*Q)
sage: (d^2 + t*d + 1)^(-1) * (d + t^2)
```

(continues on next page)
sage: g = (D*P)^(-1) * (D*Q)
sage: g
(x^5 + t^8*x^4 + x^3 + (t^2 + 1)*x^2 + (t^3 + t)*x + t^2 + 1)^(-1)
* (x^4 + t^16*x^3 + (t^2 + 1)*x + t^4 + t^2)
sage: f == g
True

>>> from sage.all import *
>>> # needs sage.rings.function_field
>>> P = d**Integer(2) + t*d + Integer(1)
>>> Q = d + t**Integer(2)
>>> D = d**Integer(3) + t**Integer(2) + Integer(1)
>>> f = P**(-Integer(1)) * Q
>>> f
(d^2 + t*d + 1)^(-1) * (d + t^2)
>>> g = (D*P)**(-Integer(1)) * (D*Q)
>>> g
(d^2 + t*d + 1)^(-1) * (d + t^2)

>>> # needs sage.rings.function_field
>>> P = x**Integer(2) + t*x + Integer(1)
>>> Q = x + t**Integer(2)
>>> D = x**Integer(3) + t**Integer(2) + Integer(1)
>>> f = P**(-Integer(1)) * Q
>>> f
(x^2 + t*x + 1)^(-1) * (x + t^2)
>>> g = (D*P)**(-Integer(1)) * (D*Q)
>>> g
(x^5 + t^8*x^4 + x^3 + (t^2 + 1)*x^2 + (t^3 + t)*x + t^2 + 1)^(-1)
* (x^4 + t^16*x^3 + (t^2 + 1)*x + t^4 + t^2)

OPERATIONS:

Basic arithmetical operations are available:

sage: # needs sage.rings.function_field
sage: f = 1 / d
sage: g = 1 / (d + t)
sage: u = f + g; u
(d^2 + ((t^2 - 1)/t)*d)^(-1) * (2*d + (t^2 - 2)/t)
sage: v = f - g; v
(d^2 + ((t^2 - 1)/t)*d)^(-1) * t
sage: u + v
d^(-1) * 2
sage: f * g
(d^2 + t*d)^(-1)
sage: f / g
d^(-1) * (d + t)

>>> from sage.all import *
>>> # needs sage.rings.function_field
>>> f = Integer(1) / d
>>> g = Integer(1) / (d + t)

1.7. Fraction fields of Ore polynomial rings 103
>>> u = f + g; u
(d^2 + ((t^2 - 1)/t)*d)^(-1) * (2*d + (t^2 - 2)/t)
>>> v = f - g; v
(d^2 + ((t^2 - 1)/t)*d)^(-1) * t
>>> u + v
d^(-1) * 2
>>> f * g
(d^2 + t*d)^(-1)
>>> f / g
d^(-1) * (d + t)

Of course, multiplication remains noncommutative:

```python
sage: # needs sage.rings.function_field
sage: g * f
(d^2 + t*d + 1)^(-1)
sage: g^(-1) * f
(d - 1/t)^(-1) * (d + (t^2 - 1)/t)
```

AUTHOR:

- Xavier Caruso (2020-05)

class sage.rings.polynomial.ore_function_field.OreFunctionCenterInjection

    Canonical injection of the center of a Ore function field into this field.

    section()

    Return a section of this morphism.

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^3)
sage: S.<x> = SkewPolynomialRing(k, k.frobenius_endomorphism())
sage: K = S.fraction_field()
sage: Z = K.center()
sage: iota = K.coerce_map_from(Z)
sage: sigma = iota.section()
sage: sigma(x^3 / (x^6 + 1))
z/(z^2 + 1)
```
```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('a',)); (a,) = k._first_ngens(1)
>>> S = SkewPolynomialRing(k, k.frobenius_endomorphism(), names=('x',)); (x,)
˓→ S._first_ngens(1)
>>> K = S.fraction_field()
>>> Z = K.center()
>>> iota = K.coerce_map_from(Z)
>>> sigma = iota.section()
>>> sigma(x**Integer(3) / (x**Integer(6) + Integer(1)))
z/(z^2 + 1)
```

```python
class sage.rings.polynomial.ore_function_field.OreFunctionField
    ring, category=None)

Bases: Parent, UniqueRepresentation

A class for fraction fields of Ore polynomial rings.

Element = None

change_var(var)

Return the Ore function field in variable var with the same base ring, twisting morphism and twisting derivation as self.

INPUT:

• var – a string representing the name of the new variable.

EXAMPLES:
```
characteristic()

Return the characteristic of this Ore function field.

EXAMPLES:

```sage
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S = R['x',sigma]
sage: S.fraction_field().characteristic()  # needs sage.rings.function_field
0
sage: # needs sage.rings.finite_rings
sage: k.<u> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S = k['y',Frob]
sage: S.fraction_field().characteristic()  # needs sage.rings.function_field
5
```

fraction_field()

Return the fraction field of this Ore function field, i.e. this Ore function field itself.

EXAMPLES:

```sage
sage: R.<t> = QQ[]
sage: der = R.derivation()
sage: A.<d> = R[d, der]
sage: K = A.fraction_field(); K
Ore Function Field in d
over Fraction Field of Univariate Polynomial Ring in t over Rational Field
twisted by d/dt
sage: K.fraction_field()  # needs sage.rings.function_field
is K
True
```

(continues on next page)
der = R.derivation()
>>> A = R['d', der]; (d,) = A._first_ngens(1)
>>> K = A.fraction_field(); K
Ore Function Field in d
over Fraction Field of Univariate Polynomial Ring in t over Rational Field
twisted by d/dt
>>> K.fraction_field()
Ore Function Field in d
over Fraction Field of Univariate Polynomial Ring in t over Rational Field
twisted by d/dt
>>> K.fraction_field() is K
True

gen (n=0)
Return the indeterminate generator of this Ore function field.

INPUT:

- n – index of generator to return (default: 0). Exists for compatibility with other polynomial rings.

EXAMPLES:

```sage
# needs sage.rings.finite_rings
k.<a> = GF(5^4)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x, Frob]
sage: K = S.fraction_field()
sage: K.gen()
x
```

```sage
# needs sage.rings.finite_rings
from sage.all import *

k = GF(Integer(5)**Integer(4), names=('a',)); (a,) = k._first_ngens(1)
Frob = k.frobenius_endomorphism()
S = k['x', Frob]; (x,) = S._first_ngens(1)
K = S.fraction_field()
K.gen()
x
```

gens ()
Return the tuple of generators of self.

EXAMPLES:

```sage
# needs sage.rings.finite_rings
k.<a> = GF(5^4)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x', Frob]
sage: K = S.fraction_field()
sage: K.gens()
(x,
```

```sage
# needs sage.rings.finite_rings
from sage.all import *

k = GF(Integer(5)**Integer(4), names=('a',)); (a,) = k._first_ngens(1)
Frob = k.frobenius_endomorphism()
S = k['x', Frob]; (x,) = S._first_ngens(1)
```
Noncommutative Polynomials, Release 10.4

>>> K = S.fraction_field()
>>> K.gens()
(x,)

gens_dict()

Return a {name: variable} dictionary of the generators of this Ore function field.

EXAMPLES:

sage: # needs sage.rings.finite_rings
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = OrePolynomialRing(R, sigma)
sage: K = S.fraction_field()
sage: K.gens_dict()
{'x': x}

is_commutative()

Return True if this Ore function field is commutative, i.e. if the twisting morphism is the identity and the twisting derivation vanishes.

EXAMPLES:

sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x, Frob]
sage: K = S.fraction_field()
sage: K.is_commutative()
False
sage: T.<y> = k[y, Frob**Integer(3)]
sage: L = T.fraction_field()
sage: L.is_commutative()
True

>>> from sage.all import *

# needs sage.rings.finite_rings

k = GF(Integer(5)**Integer(3), names=('a',)); (a,) = k._first_ngens(1)
Frob = k.frobenius_endomorphism()
S = k['x', Frob]; (x,) = S._first_ngens(1)
K = S.fraction_field()
K.is_commutative()
False
T = k['y', Frob**Integer(3)]; (y,) = T._first_ngens(1)
L = T.fraction_field()
L.is_commutative()
True
is_exact()  
Return True if elements of this Ore function field are exact. This happens if and only if elements of the base ring are exact.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x'], Frob
sage: K = S.fraction_field()
sage: K.is_exact()
   True
sage: # needs sage.rings.padics
sage: k.<u> = Qq(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x'], Frob
sage: K = S.fraction_field()
sage: K.is_exact()
   False
```

is_field(proof=False)
Return always True since Ore function field are (skew) fields.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x'], Frob
sage: K = S.fraction_field()
sage: S.is_field()
   False
sage: K.is_field()
   True
```

```
is_finite()

Return False since Ore function field are not finite.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: k.is_finite()
True
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x, Frob]
sage: K = S.fraction_field()
sage: K.is_finite()
False
```

is_sparse()

Return True if the elements of this Ore function field are sparsely represented.

**Warning:** Since sparse Ore polynomials are not yet implemented, this function always returns False.

EXAMPLES:

```
sage: # needs sage.rings.function_field sage.rings.real_mpfr
sage: R.<t> = RR[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R[x, sigma]
sage: K = S.fraction_field()
sage: K.is_sparse()
False
```

(continues on next page)
ngens()

Return the number of generators of this Ore function field, which is 1.

EXAMPLES:

```python
sage: K = S.fraction_field()
sage: K.is_sparse()
False
```

parameter(n=0)

Return the indeterminate generator of this Ore function field.

INPUT:

- n – index of generator to return (default: 0). Exists for compatibility with other polynomial rings.

EXAMPLES:

```python
sage: from sage.all import *
sage: # needs sage.rings.function_field sage.rings.real_mpfr
sage: R = RR['t']; (t,) = R._first_ngens(1)
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: K = S.fraction_field()
sage: K.ngens()
1
```

random_element(degree=2, monic=False, *args, **kwds)

Return a random Ore function in this field.

INPUT:

- degree – (default: 2) an integer or a list of two integers; the degrees of the denominator and numerator

```python
>>> from sage.all import *
>>> # needs sage.rings.function_field sage.rings.real_mpfr
>>> R = RR['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> K = S.fraction_field()
>>> K.ngens()
1
```

random_element(degree=2, monic=False, *args, **kwds)

Return a random Ore function in this field.

INPUT:

- degree – (default: 2) an integer or a list of two integers; the degrees of the denominator and numerator

```python
>>> from sage.all import *
>>> # needs sage.rings.function_field sage.rings.real_mpfr
>>> R = RR['t']; (t,) = R._first_ngens(1)
>>> sigma = R.hom([t+Integer(1)])
>>> S = R['x',sigma]; (x,) = S._first_ngens(1)
>>> K = S.fraction_field()
>>> K.ngens()
1
```
• **monic** – (default: False) if True, return a monic Ore function with monic numerator and denominator

• **args**, **kwds** – passed in to the `random_element()` method for the base ring

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x, Frob]
sage: K = S.fraction_field()
sage: K.random_element()  # random
(x^2 + (2*t^2 + t + 1)*x + 2*t^2 + 2*t + 3)^(-1)
* ((2*t^2 + 3)*x^2 + (4*t^2 + t + 4)*x + 2*t^2 + 2)
sage: K.random_element(monic=True)  # random
(x^2 + (4*t^2 + 3*t + 4)*x + 4*t^2 + t)^(-1)
* ((t + 4)*x^3 + (4*t^2 + 2*t + 2)*x^2 + (2*t^2 + 3*t + 3)*x + 3*t^2 + 3*t + 1)
sage: K.random_element(degree=2)  # random
(x^2 + (4*t^2 + 2*t + 2)*x + 4*t^2 + t + 2)^(-1)
* (3*t^2 + t + 1)*x^5 + (2*t^2 + 2*t)*x^4 + (t^2 + 2*t + 4)*x^3
+ (3*t^2 + 2*t)*x^2 + (t^2 + t + 4)*x
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x', Frob]; (x,) = S._first_ngens(1)
>>> K = S.fraction_field()
>>> K.random_element()  # random
(x^2 + (2*t^2 + t + 1)*x + 2*t^2 + 2*t + 3)^(-1)
* ((2*t^2 + 3)*x^2 + (4*t^2 + t + 4)*x + 2*t^2 + 2)
>>> K.random_element(monic=True)  # random
(x^2 + (4*t^2 + 3*t + 4)*x + 4*t^2 + t)^(-1)
* ((t + 4)*x^3 + (4*t^2 + 2*t + 2)*x^2 + (2*t^2 + 3*t + 3)*x + 3*t^2 + 3*t + 1)
>>> K.random_element(degree=2)  # random
(x^2 + (4*t^2 + 2*t + 2)*x + 4*t^2 + t + 2)^(-1)
* (3*t^2 + t + 1)*x^5 + (2*t^2 + 2*t)*x^4 + (t^2 + 2*t + 4)*x^3
+ (3*t^2 + 2*t)*x^2 + (t^2 + t + 4)*x
```

```
twisting_derivation()

Return the twisting derivation defining this Ore function field or None if this Ore function field is not twisted by a derivation.

EXAMPLES:

```python
sage: R.<t> = QQ[]
sage: der = R.derivation(); der
d/dt
sage: A.<d> = R['d', der]
```

(continues on next page)
sage: F = A.fraction_field()
sage: F.twisting_derivation()
d/dt

sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x, Frob]
sage: K = S.fraction_field()
sage: K.twisting_derivation()

See also:

sage.rings.polynomial.ore_polynomial_element.OrePolynomial.
twisting_derivation(), twisting_morphism()

twisting_morphism(n=1)

Return the twisting endomorphism defining this Ore function field iterated \(n\) times or None if this Ore function field is not twisted by an endomorphism.

INPUT:

* \(n\) – an integer (default: 1)

EXAMPLES:

\begin{verbatim}
>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> der = R.derivation(); der
d/dt
>>> A = R['d', der]; (d,) = A._first_ngens(1)
>>> F = A.fraction_field()
>>> F.twisting_derivation()
d/dt

>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('a',)); (a,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x', Frob]; (x,) = S._first_ngens(1)
>>> K = S.fraction_field()
>>> K.twisting_derivation()
\end{verbatim}
When the Ore polynomial ring is only twisted by a derivation, this method returns nothing:

```python
sage: der = R.derivation()
sage: A.<d> = R['x', der]
sage: F = A.fraction_field()  # needs sage.rings.function_field
```

See also:

sage.rings.polynomial.ore_polynomial_element.OrePolynomial.twisting_morphism(), twisting_derivation()

class sage.rings.polynomial.ore_function_field.OreFunctionField_with_large_center

Bases: OreFunctionField

A specialized class for Ore polynomial fields whose center has finite index.

**center** *(name=None, names=None, default=False)*

Return the center of this Ore function field.

**Note:** One can prove that the center is a field of rational functions over a subfield of the base ring of this Ore function field.

**INPUT:**

- name – a string or None (default: None); the name for the central variable
- default – a boolean (default: False); if True, set the default variable name for the center to name

**EXAMPLES:**

```python
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
```
We can pass in another variable name:

```
sage: K.center(name=y)  # needs sage.rings.finite_rings
Fraction Field of Univariate Polynomial Ring in y over Finite Field of size 5
```

or use the bracket notation:

```
sage: Z.<y> = K.center(); Zy
# needs sage.rings.finite_rings
Fraction Field of Univariate Polynomial Ring in y over Finite Field of size 5
```

A coercion map from the center to the Ore function field is set:

```
sage: K.has_coerce_map_from(Zy)  # needs sage.rings.finite_rings
True
```

and pushout works:

```
sage: x.parent()  # needs sage.rings.finite_rings
Ore Polynomial Ring in x over Finite Field in t of size 5^3 twisted by t |--> t^5
sage: y.parent()  # needs sage.rings.finite_rings
Fraction Field of Univariate Polynomial Ring in y over Finite Field of size 5
sage: P = x + y; P
```

(continues on next page)
x^3 + x
sage: P.parent()
Ore Function Field in x over Finite Field in \( t \) of size 5^3 twisted by \( t \mid \rightarrow \) \( t^5 \)

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> x.parent()
Ore Polynomial Ring in x over Finite Field in \( t \) of size 5^3 twisted by \( t \mid \rightarrow \) \( t^5 \)
>>> y.parent()
Fraction Field of Univariate Polynomial Ring in y over Finite Field of size 5
>>> P = x + y; P
x^3 + x
>>> P.parent()
Ore Function Field in x over Finite Field in \( t \) of size 5^3 twisted by \( t \mid \rightarrow \) \( t^5 \)
```

A conversion map in the reverse direction is also set:

```
sage: # needs sage.rings.finite_rings
te supported
sage: Zy(x^(-6) + 2)
(2*y^2 + 1)/y^2
sage: Zy(1/x^2)
Traceback (most recent call last):
... ValueError: x^(-2) is not in the center
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> Zy(x**(-Integer(6)) + Integer(2))
(2*y^2 + 1)/y^2
>>> Zy(Integer(1)/x**Integer(2))
Traceback (most recent call last):
... ValueError: x^(-2) is not in the center
```

**ABOUT THE DEFAULT NAME OF THE CENTRAL VARIABLE:**

A priori, the default is \( z \).

However, a variable name is given the first time this method is called, the given name become the default for the next calls:

```
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(11^3)
sage: phi = k.frobenius_endomorphism()
sage: S.<X> = k[X, phi]
sage: K = S.fraction_field()
sage: C.<u> = K.center()  # first call
sage: C
Fraction Field of Univariate Polynomial Ring in u over Finite Field of size 11
sage: K.center()  # second call: the variable name is still u
Fraction Field of Univariate Polynomial Ring in u over Finite Field of size 11
```

116 Chapter 1. Univariate Ore Polynomials
We can update the default variable name by passing in the argument `default=True`:

```
sage: D.<v> = K.center(default=True)
sage: D
Fraction Field of Univariate Polynomial Ring in v over Finite Field of size 11
sage: K.center()
Fraction Field of Univariate Polynomial Ring in v over Finite Field of size 11
```

1.8 Fraction field elements of Ore polynomial rings

**AUTHOR:**

- Xavier Caruso (2020-05)

**EXAMPLES:**

```
sage: k.<a> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x'], Frob
```
sage: K = S.fraction_field()
sage: iota = K.coerce_map_from(k)
sage: sigma = iota.section(); sigma
Generic map:
    From: Ore Function Field in x over Finite Field in a of size 5^3
twisted by a |--> a^5
    To: Finite Field in a of size 5^3

>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> from sage.rings.polynomial.ore_polynomial_element import ConstantOrePolynomialSection

k = GF(Integer(5)**Integer(3), names=('a',)); (a,) = k._first_ngens(1)
Frob = k.frobenius_endomorphism()
S = k['x', Frob]; (x,) = S._first_ngens(1)
K = S.fraction_field()
iota = K.coerce_map_from(k)
sigma = iota.section(); sigma

Generic map:
    From: Ore Function Field in x over Finite Field in a of size 5^3
twisted by a |--> a^5
    To: Finite Field in a of size 5^3

class sage.rings.polynomial.ore_function_element.OreFunction

    (parent, numerator, denominator=None, simplify=True)

Bases: AlgebraElement

An element in a Ore function field.

hilbert_shift (s, var=None)

    Return this Ore function with variable shifted by s, i.e. if this Ore function is \( f(x) \), return \( f(x+s) \).

INPUT:

    * s – an element in the base ring
    * var – a string; the variable name

EXAMPLES:

sage: R.<t> = GF(7)[]
sage: der = R.derivation()
sage: A.<d> = R['d', der]
sage: K = A.fraction_field()

sage: f = 1 / (d-t)
sage: f.hilbert_shift(t)
d^(-1)

>>> from sage.all import *
>>> R = GF(Integer(7))['t']; (t,) = R._first_ngens(1)
>>> der = R.derivation()
>>> A = R['d', der]; (d,) = A._first_ngens(1)
>>> K = A.fraction_field()

>>> f = Integer(1) / (d-t)
One can specify another variable name:

```python
sage: f.hilbert_shift(t, var='x')
x^(-1)
```

When the twisting morphism is not trivial, the output lies in a different Ore polynomial ring:

```python
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x', Frob]
sage: K = S.fraction_field()
sage: f = (x-a)^(-2)
sage: g = f.hilbert_shift(a); g
x^(-2)
sage: g.parent()
Ore Function Field in x over Finite Field in a of size 5^3 twisted by a |--> a^5 and a*(a |--> a^5) - id
sage: g.parent() is S
False
```

This behavior ensures that the Hilbert shift by a fixed element defines a homomorphism of fields:

```python
sage: # needs sage.rings.finite_rings
sage: U = K.random_element(degree=5)
sage: V = K.random_element(degree=5)
sage: s = k.random_element()
sage: (U+V).hilbert_shift(s) == U.hilbert_shift(s) + V.hilbert_shift(s)
True
sage: (U*V).hilbert_shift(s) == U.hilbert_shift(s) * V.hilbert_shift(s)
True
```
is_zero()

Return **True** if this element is equal to zero.

**EXAMPLES:**

```python
sage: R.<t> = GF(3)[]
sage: der = R.derivation()
sage: A.<d> = R[x, der]
sage: f = t/d
sage: f.is_zero()
False
sage: (f-f).is_zero()
True
```

left_denominator()

Return $s$ if this element reads $s^{-1}t$.

**WARNING:**

When the twisting morphism is bijective, there is a unique irreducible fraction of the form $s^{-1}t$ representing this element. Here irreducible means that $s$ and $t$ have no nontrivial common left divisor. Under this additional assumption, this method always returns this distinguished denominator $s$.

On the contrary, when the twisting morphism is not bijective, this method returns the denominator of some fraction representing the input element. However, the software guarantees that the method `right_numerator()` outputs the numerator of the *same* fraction.

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x', Frob]
sage: s = x + a
sage: t = x^2 + a*x + a^2
sage: f = s^(-1) * t
sage: f.left_denominator()
x + a
```
Noncommutative Polynomials, Release 10.4

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('a',)); (a,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x', Frob]; (x,) = S._first_ngens(1)
>>> s = x + a
>>> t = x**Integer(2) + a*x + a**Integer(2)
>>> f = s**(-Integer(1)) * t
>>> f.left_denominator()
x + a
```

In the example below, a simplification occurs:

```python
sage: # needs sage.rings.finite_rings
sage: u = S.random_element(degree=2)
sage: g = (u*s)^(-1) * (u*t)
sage: g.left_denominator()
x + a
```

When the twisting morphism is not invertible, simplifications do not occur in general:

```python
sage: R.<z> = GF(11)[]
sage: sigma = R.hom([z^2])
sage: S.<x> = R[x, sigma]
sage: s = (x + z)**Integer(2)
sage: t = (x + z) * (x**Integer(2) + z**Integer(2))
sage: f = s**(-Integer(1)) * t
>>> f.left_denominator()
x^2 + (z^2 + z)*x + z^2
```

However, the following always holds true:

```python
sage: f == f.left_denominator()^(-1) * f.right_numerator()  # needs sage.rings.function_field
True
```
See also:

`right_numerator()`, `left_numerator()`, `right_denominator()`

`left_numerator()`

Return $t$ if this element reads $ts^{-1}$.

**WARNING:**

When the twisting morphism is bijective, there is a unique irreducible fraction of the form $ts^{-1}$ representing this element. Here irreducible means that $s$ and $t$ have no nontrivial common right divisor. Under this additional assumption, this method always returns this distinguished numerator $t$.

On the contrary, when the twisting morphism is not bijective, the existence of the writing $ts^{-1}$ is not guaranteed in general. In this case, this method raises an error.

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x, Frob]
sage: s = x + a
sage: t = x^2 + a*x + a^2
sage: f = t/s
```

```python
>>> from sage.all import *
>>> # needs sage.rings.function_field
>>> f == f.left_denominator()**(Integer(-1)) * f.right_numerator()  # needs sage.rings.function_field
True
```

In the example below, a simplification occurs:

```python
sage: # needs sage.rings.finite_rings
sage: u = S.random_element(degree=2)
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('a',)); (a,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x', Frob]; (x,) = S._first_ngens(1)
>>> s = x + a
>>> t = x^2 + a*x + a^2
```

```python
>>> f = t/s
```

```python
sage: # needs sage.rings.finite_rings
sage: u = S.random_element(degree=Integer(2))
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> g = (t*u) / (s*u)
```

```python
sage: # needs sage.rings.function_field
sage: g.left_numerator()  # needs sage.rings.function_field
x^2 + a*x + a^2
```

```python
sage: g.left_numerator()  # needs sage.rings.function_field
x^2 + a*x + a^2
```

In the example below, a simplification occurs:

```python
sage: # needs sage.rings.finite_rings
sage: u = S.random_element(degree=2)
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> u = S.random_element(degree=Integer(2))
>>> g = (t*u) / (s*u)
```

```python
sage: g.left_numerator()  # needs sage.rings.function_field
x^2 + a*x + a^2
```
**right_denominator()**

Return \( s \) if this element reads \( ts^{-1} \).

**WARNING:**

When the twisting morphism is bijective, there is a unique irreducible fraction of the form \( ts^{-1} \) representing this element. Here irreducible means that \( s \) and \( t \) have no nontrivial common right divisor. Under this additional assumption, this method always returns this distinguished denominator \( s \).

On the contrary, when the twisting morphism is not bijective, the existence of the writing \( ts^{-1} \) is not guaranteed in general. In this case, this method raises an error.

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^3)
Sage: Frob = k.frobenius_endomorphism()
Sage: S.<x> = k['x', Frob]
Sage: s = x + a
Sage: t = x^2 + a*x + a^2
Sage: f = t/s
Sage: f.right_denominator()
x + a
```

In the example below, a simplification occurs:

```python
sage: # needs sage.rings.finite_rings
sage: u = S.random_element(degree=2)
Sage: g = (t*u) / (s*u)
Sage: g.right_denominator()
x + a
```

See also:

* left_numerator(), left_denominator(), right_numerator()
When the twisting morphism is bijective, there is a unique irreducible fraction of the form \( s^{-1}t \) representing this element. Here irreducible means that \( s \) and \( t \) have no nontrivial common left divisor. Under this additional assumption, this method always returns this distinguished numerator \( t \).

On the contrary, when the twisting morphism is not bijective, this method returns the numerator of some fraction representing the input element. However, the software guarantees that the method \( \text{left_denominator}() \) outputs the numerator of the same fraction.

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^3)

sage: Frob = k.frobenius_endomorphism()

sage: S.<x> = k[x, Frob]

sage: s = x + a

sage: t = x^2 + a*x + a^2

sage: f = s^(-1) * t

sage: f.right_numerator()
```

```
x^2 + a*x + a^2
```

```python
>>> from sage.all import *

>>> # needs sage.rings.finite_rings

>>> k = GF(Integer(5)**Integer(3), names=('a',)); (a,) = k._first_ngens(1)

>>> Frob = k.frobenius_endomorphism()

>>> S = k['x', Frob]; (x,) = S._first_ngens(1)

>>> s = x + a

>>> t = x**Integer(2) + a*x + a**Integer(2)

>>> f = s**(-Integer(1)) * t

>>> f.right_numerator()
```

```
x^2 + a*x + a^2
```

In the example below, a simplification occurs:

```python
sage: # needs sage.rings.finite_rings
sage: u = S.random_element(degree=2)

sage: g = (u*s)**(-1) * (u*t)

sage: g.right_numerator()
```

```
x^2 + a*x + a^2
```

```python
>>> from sage.all import *

>>> # needs sage.rings.finite_rings

>>> u = S.random_element(degree=Integer(2))

>>> g = (u*s)**(-Integer(1)) * (u*t)

>>> g.right_numerator()
```

```
x^2 + a*x + a^2
```

See also:

\( \text{left_denominator}(), \text{left_numerator}(), \text{right_denominator}() \)

```python
class sage.rings.polynomial.ore_function_element.OreFunctionBaseringInjection
```

Bases: Morphism

Representation of the canonical homomorphism from a field \( k \) into a Ore function field over \( k \).

This class is needed by the coercion system.
an_element()

Return an element of the codomain of the ring homomorphism.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x, Frob]
sage: K = S.fraction_field()
sage: m = K.coerce_map_from(k)
sage: m.an_element()
x
```

section()

Return the canonical homomorphism from the constants of a Ore function field to its base field.

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(3), names=('t',)); (t,) = k._first_ngens(1)
>>> Frob = k.frobenius_endomorphism()
>>> S = k['x',Frob]; (x,) = S._first_ngens(1)
>>> K = S.fraction_field()
>>> m = K.coerce_map_from(k)
>>> m.an_element()
x
```
The reduced norm lies in the center of $S$, which is the fraction field of a univariate polynomial ring in the variable $z = x^3$ over $GF(5)$.

We can use explicit conversion to view $N$ as a skew polynomial:

By default, the name of the central variable is usually $z$ (see sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_finite_order.center() for more details about this). However, the user can specify a different variable name if desired:

```
reduced_trace (var=None)
```
Return the reduced trace of this element.

**INPUT:**

- var – a string or None (default: None); the name of the central variable

**EXAMPLES:**

```
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k[x, Frob]
sage: K = S.fraction_field()
sage: a = 1 / (x^2 + t)
sage: tr = a.reduced_trace(); tr
3/(z^2 + 2)
```

The reduced trace lies in the center of $S$, which is the fraction field of a univariate polynomial ring in the variable $z = x^3$ over $GF(5)$:

```
sage: tr.parent()  
Fraction Field of Univariate Polynomial Ring in z over Finite Field of size 5
sage: tr.parent() is K.center()  
True
```

We can use explicit conversion to view $tr$ as a Ore function:

```
sage: K(tr)  
(x^6 + 2)^(-1) * 3
```

By default, the name of the central variable is usually $z$ (see `sage.rings.polynomial.skew_polynomial_ring.OreFunctionField_with_large_center.center()` for more details about this). However, the user can specify a different variable name if desired:
```
sage: a.reduced_trace(var='u')
→ needs sage.rings.finite_rings
3/(u^2 + 2)
```

```
>>> from sage.all import *
>>> a.reduced_trace(var='u')
→ needs sage.rings.finite_rings
3/(u^2 + 2)
```
2.1 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
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)
Noncommutative Polynomials, Release 10.4

(continued from previous page)

```python
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, nc-relations: {y*x: -x*y}

sage: y*x + 7
-x*y + 7
```

```python
>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(3), names =(x, y, z,)); (x, y, z,)
... = A._first_ngens(3)
>>> P = A.g_algebra(relations={y*x:-x*y}, order = 'lex', names=(x, y, z,)); (x, y, z,)
... = P._first_ngens(3)

>>> P
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-
relations: {y*x: -x*y}

>>> y*x + Integer(1)/Integer(2)
-x*y + 1/2

>>> A = FreeAlgebra(GF(Integer(17)), Integer(3), names=('x', 'y', 'z',)); (x, y, z,)
... = A._first_ngens(3)
>>> P = A.g_algebra(relations={y*x:-x*y}, order = 'lex', names=('x', 'y', 'z',)); (x, y, z,)
... = P._first_ngens(3)

>>> P
Noncommutative Multivariate Polynomial Ring in x, y, z over Finite Field of size 17, nc-relations: {y*x: -x*y}

>>> y*x + Integer(7)
-x*y + 7
```

Raw use of this class; this is not the intended use!

```python
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),
```

(continues on next page)
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, non-operating relations: {y*x: -2*x*y + 17}

sage: R
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, non-operating 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:

sage: from sage.rings.polynomial.plural import ExteriorAlgebra
sage: E = ExteriorAlgebra(QQ, ['x', 'y', 'z']); E
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, z^2, x^2, y^2, z^2, x*y, y*x, z*y, x*z, y*z, z*x)

\texttt{sage}: \texttt{sorted(E.cover().domain().relations().items(), key=str)}
\texttt{[(y*x, -x*y), (z*x, -x*z), (z*y, -y*z)]}
\texttt{sage}: \texttt{sorted(E.cover().kernel().gens(), key=str)}
\texttt{[x^2, y^2, z^2]}
\texttt{sage}: E.inject_variables()
\texttt{Defining xbar, ybar, zbar}
\texttt{sage}: x, y, z = (xbar, ybar, zbar)
\texttt{sage}: y*x
\texttt{-x*y}
\texttt{sage}: \texttt{all(v^2==0 for v in E.gens())}
\text{True}
\texttt{sage}: E.one()
\texttt{1}

```python
>>> from sage.all import *
>>> from sage.rings.polynomial.plural import ExteriorAlgebra
>>> E = ExteriorAlgebra(QQ, ['x', 'y', 'z']) ; E
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)

>>> sorted(E.cover().domain().relations().items(), key=str)
\texttt{[(y*x, -x*y), (z*x, -x*z), (z*y, -y*z)]}

>>> sorted(E.cover().kernel().gens(), key=str)
\texttt{[x^2, y^2, z^2]}

>>> E.inject_variables()
\texttt{Defining xbar, ybar, zbar}

>>> x, y, z = (xbar, ybar, zbar)

>>> y*x
\texttt{-x*y}

>>> \texttt{all(v**Integer(2)==Integer(0) for v in E.gens())}
\text{True}

>>> E.one()
\texttt{1}
```

class \texttt{sage.rings.polynomial.plural.ExteriorAlgebraplural}

\texttt{Bases: NCPolynomialRingPlural}

class \texttt{sage.rings.polynomial.plural.G_AlgFactory}

\texttt{Bases: UniqueFactory}

A factory for the creation of g-algebras as unique parents.

\texttt{create_key_and_extra_args}(base\_ring, c, d, names=\texttt{None}, order=\texttt{None}, category=\texttt{None}, check=\texttt{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’.

```python
class sage.rings.polynomial.plural.NCPolynomialRing_plural
Bases: 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
case: H.category()
Category of algebras over Rational Field
case: TestSuite(H).run()

>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer( 3), names=('x', 'y', 'z',)); (x, y, z,) = A._
˓→first_ngens(3)
>>> H = A.g_algebra((y*x+x*y-z, z*x:x*z+Integer(2)*x, z*y:y*z-Integer(2)*y))
>>> H._is_category_initialized()
True
>>> H.category()
Category of algebras over Rational Field
>>> 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

>>> from sage.all import *
>>> H = A.g_algebra((z*x:x*z+Integer(2)*x, z*y:y*z-Integer(2)*y), names=('x', 'y',
˓→ 'z',)); (x, y, z,) = H._first_ngens(3)
>>> x*y == y*x
True

algebra_generators()
Return the algebra generators of self.

EXAMPLES:

sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order='lex')
sage: P.algebra_generators()
Finite family {x: x, 'y': y, 'z': z}
```
```python
>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z')); (x, y, z) = A._first_ngens(3)
>>> P = A.g_algebra(relations=(y*x:-x*y), order='lex')
>>> P.algebra_generators()
Finite family {'x': x, 'y': y, 'z': z}
```

**free_algebra()**

Return 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
```

```python
>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z')); (x, y, z) = A._first_ngens(3)
>>> P = A.g_algebra(relations=(y*x:-x*y), order = 'lex')
>>> B = P.free_algebra()
>>> A == B
True
```

**gen(n=0)**

Return the \(n\)-th generator of this noncommutative polynomial ring.

**INPUT:**

- \(n\) – an integer \(\geq 0\)

**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)
```

```
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(Integer(1))
(x, y)
```

Note that the generators are not cached:

```
sage: P.gen(1) is P.gen(1)
False
```

```
sage: from sage.all import *
>>> P.gen(Integer(1)) is P.gen(Integer(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: 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) 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 - x + 2*y^2 + 2*z^2) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {y*x: -x*y}
```

is_commutative()
Return False.

Todo: Provide a mathematically correct answer.

EXAMPLES:

```
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
```
>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = A._first_ngens(3)
>>> P = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
>>> P.is_commutative()
False

is_field(*args, **kwargs)
Return False.
EXAMPLES:

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

>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = A._first_ngens(3)
>>> P = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
>>> P.is_field()
False

monomial_all_divisors(t)
Return a list of all monomials that divide t.
Coefficients are ignored.

INPUT:
- t – a monomial

OUTPUT:
a list of monomials

EXAMPLES:

sage: 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]

>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = A._first_ngens(3)
>>> P = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
>>> P.inject_variables()
Defining x, y, z
>>> P.monomial_all_divisors(x**Integer(2)*z**Integer(3))
[x, x^2, z, x*z, x^2*z, z^2, x*z^2, x^2*z^2, z^3, x*z^3, x^2*z^3]

ALGORITHM: addwithcarry idea by Toon Segers
monomial_divides \((a, b)\)

Return \(False\) if \(a\) does not divide \(b\) and \(True\) otherwise. Coefficients are ignored.

**INPUT:**

- \(a\) – monomial
- \(b\) – monomial

**EXAMPLES:**

```sage
A.<x,y,z> = FreeAlgebra(QQ, 3)
P = A.g_algebra(relations={y*x:-x*y}, order='lex')
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
```

```python
>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z',)); (x, y, z) = A._first_ngens(3)
>>> P = A.g_algebra(relations={y*x:-x*y}, order='lex')
>>> P.inject_variables()
Defining x, y, z
>>> P.monomial_divides(x*y*z, x**Integer(3)*y**Integer(2)*z**Integer(4))
True
>>> P.monomial_divides(x**Integer(3)*y**Integer(2)*z**Integer(4), x*y*z)
False
```

monomial_lcm \((f, g)\)

LCM for monomials. Coefficients are ignored.

**INPUT:**

- \(f\) – monomial
- \(g\) – monomial

**EXAMPLES:**

```sage
A.<x,y,z> = FreeAlgebra(QQ, 3)
P = A.g_algebra(relations={y*x:-x*y}, order='lex')
P.inject_variables()
Defining x, y, z
sage: P.monomial_lcm(3/2*x*y,x)
x*y
```

```python
>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z',)); (x, y, z) = A._first_ngens(3)
>>> P = A.g_algebra(relations={y*x:-x*y}, order='lex')
>>> P.inject_variables()
Defining x, y, z
```

(continues on next page)
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)
True
sage: P.monomial_pairwise_prime(1/2*x^3*y^2, 3/4*y^3)
False
```

monomial_quotient \( (f, g, \text{coeff=False}) \)

Return \( f/g \), where both \( f \) and \( g \) are treated as monomials.

Coefficients are ignored by default.

**INPUT:**

- \( f \) – monomial
- \( g \) – monomial
- \( \text{coeff} \) – divide coefficients as well (default: False)

**EXAMPLES:**

```python
>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z'),); (x, y, z,)=A._
˓→first_ngens(3)
>>> P = A.g_algebra(relations={y*x:-x*y}, order='lex')
>>> P.inject_variables()
Defining x, y, z
>>> P.monomial_quotient(x*Integer(2)*z*Integer(3), y*Integer(4))
True
>>> P.monomial_quotient(Integer(1)/Integer(2)*x*Integer(3)*y*Integer(2), Integer(3)/Integer(4)*y*Integer(3))
False
```
```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_quotient(3/2*x*y,x,coeff=True)
3/2*y
```

```python
>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z')); (x, y, z,) = A._first_ngens(3)
>>> P = A.g_algebra(relations={y*x:-x*y}, order='lex')
>>> P.inject_variables()
Defining x, y, z
>>> P.monomial_quotient(Integer(3)/Integer(2)*x*y,x,coeff=True)
3/2*y
```

Note that \( Z \) behaves differently if `coeff=True`:

```python
sage: P.monomial_quotient(2*x,3*x)
1
sage: P.monomial_quotient(2*x,3*x,coeff=True)
2/3
```

```python
>>> from sage.all import *
>>> P.monomial_quotient(Integer(2)*x,Integer(3)*x)
1
>>> P.monomial_quotient(Integer(2)*x,Integer(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 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 \((\text{flt}, g)\) is returned, \((0, 0)\) otherwise, where \( \text{flt} = f/g.lm() \).

It is assumed that \( G \) is iterable and contains *only* elements in this polynomial ring.

Coefficients are ignored.

**INPUT:**

- \( f \) – monomial
- \( G \) – list/set of mpolynomials

**EXAMPLES:**

```python
sage: 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: f = x*y^2
```

(continues on next page)
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)

>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z')); (x, y, z,) = A._
˓→first_ngens(3)
>>> P = A.g_algebra(relations={y*x:-x*y}, order='lex')
>>> P.inject_variables()
Defining x, y, z

>>> f = x*y**Integer(2)
>>> G = [ Integer(3)/Integer(2)*x**Integer(3) + y**Integer(2) + Integer(1)/
˓→Integer(2), Integer(1)/Integer(4)*x*y + Integer(2)/Integer(7), Integer(1)/
˓→Integer(2) ]
>>> P.monomial_reduce(f,G)
(y, 1/4*x*y + 2/7)

ngens()

Return 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

>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z')); (x, y, z,) = A._
˓→first_ngens(3)
>>> P = A.g_algebra(relations={y*x:-x*y}, order = 'lex', names=('x', 'y', 'z',
˓→)); (x, y, z,) = P._first_ngens(3)
>>> 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()
{z*x: x*z + 2*z, 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}
```python
>>> from sage.all import *

>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = A._first_ngens(3)
>>> H = A.g_algebra({z*x:x*z+Integer(2)*x, z*y:y*z-Integer(2)*y}, names=('x', 'y', 'z',)); (x, y, z,) = H._first_ngens(3)
>>> x*y == y*x
True
>>> H.relations()
{z*x: x*z + 2*x, z*y: y*z - 2*y}
>>> 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: 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:**

```
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
```
```
>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'z', 'y')); (x, z, y,) = A._
˓→first_ngens(3)
>>> R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
>>> R.inject_variables()
Defining x, z, y
>>> f=x*y+y=Integer(5)
>>> f.coefficient({x:Integer(0),y:Integer(1)})
1
>>> f.coefficient({x:Integer(0)})
y + 5
>>> f=(Integer(1)+y+y**Integer(2))*(Integer(1)+x+x**Integer(2))
>>> f.coefficient({x:Integer(0)})
z + y^2 + y + 1
>>> f.coefficient(x)
y^2 - y + 1
>>> f.coefficient([Integer(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.
```
```
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')
```
```
(continues on next page)
AUTHOR:

- Joel B. Mohler (2007-10-31)

constant_coefficient()

Return the constant coefficient of this multivariate polynomial.

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: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.constant_coefficient()  # outputs the full polynomial
5
```

```python
>>> from sage.all import *
>>> A = FreeAlgebra(GF(Integer(389)), Integer( 3), names =(x, z, y,)); (x, z, y) = A._first_ngens(3)
>>> P = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
>>> P.inject_variables()
Defining x, z, y
>>> f = Integer(3)*x**Integer(2) - Integer(2)*y + Integer(7)*x**Integer(2)*y**Integer(2) + Integer(5)
>>> f.constant_coefficient()  # outputs the full polynomial
5
```

```python
>>> from sage.all import *
>>> A = FreeAlgebra(GF(Integer(389)), Integer(3), names=('x', 'z', 'y')); (x, z, y) = A._first_ngens(3)
>>> P = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
>>> P.inject_variables()
Defining x, z, y
>>> f = x*y+Integer(5)
```
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.

INPUT:

- x – multivariate polynomial (a generator of the parent of self) If x is not specified (or is None), return the total degree, which is the maximum degree of any monomial.

OUTPUT:

integer

EXAMPLES:

```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 = 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
```
sage: A.<y0,y1,y2> = FreeAlgebra(QQ, 3)
sage: R = A.g_algebra(relations={y1*y0:-y0*y1 + y2}, order='lex')
sage: R.inject_variables()
Defining y0, y1, y2
sage: q = 3*y0*y1*y1*y2; q
3*y0*y1^2*y2
sage: q.degrees()
(1, 2, 1)
sage: (q + y0^5).degrees()
(5, 2, 1)

>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(3), names=('y0', 'y1', 'y2',)); (y0, y1, y2,)
˓→ A._first_ngens(3)
>>> R = A.g_algebra(relations={y1*y0:-y0*y1 + y2}, order='lex')
>>> R.inject_variables()
Defining y0, y1, y2
>>> q = Integer(3)*y0*y1*y1*y2; q
3*y0*y1^2*y2
>>> q.degrees()
(1, 2, 1)
>>> (q + y0**Integer(5)).degrees()
(5, 2, 1)

\textbf{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)
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 + (7)*x^2 + (3))
sage: f.dict()
{(0, 0, 0): 3, (1, 2, 3): 2, (2, 0, 0): 7}

>>> from sage.all import *
>>> A = FreeAlgebra(GF(Integer(389)), Integer(3), names=('x', 'y', 'z',)); (x, y, z,)
˓→ A._first_ngens(3)
>>> R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
>>> R.inject_variables()
Defining x, z, y
>>> f = (Integer(2)*x*y**3*z**2 + (Integer(7))*x**2 + (Integer(3)))
>>> f.dict()
{(0, 0, 0): 3, (1, 2, 3): 2, (2, 0, 0): 7}

\textbf{exponents (as\_ETuples=True)}
Return the exponents of the monomials appearing in this polynomial.

INPUT:

\begin{itemize}
\item \texttt{as\_ETuples} = (default: True) if True returns the result as an list of ETuples otherwise returns a list of tuples
\end{itemize}
EXAMPLES:

```python
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^3 + y + 2*z^2
sage: f.exponents()
[(3, 0, 0), (0, 2, 0), (0, 0, 1)]
sage: f.exponents(as_ETuples=False)
[(3, 0, 0), (0, 2, 0), (0, 0, 1)]
```

```python
>>> from sage.all import *

>>> A = FreeAlgebra(GF(Integer(389)), Integer(3), names=('x', 'z', 'y')); (x, z, y) = A._first_ngens(3)
>>> R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
>>> R.inject_variables()
Defining x, z, y
>>> f = x**Integer(3) + y + Integer(2)*z**Integer(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)]
```

```python
is_constant()
Return True if this polynomial is constant.

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_constant()
False
sage: P(Integer(1)).is_constant()
True
```

```python
>>> from sage.all import *

>>> A = FreeAlgebra(GF(Integer(389)), Integer(3), names=('x', 'z', 'y')); (x, z, y) = A._first_ngens(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(Integer(1)).is_constant()
True
```

```python
is_homogeneous()
Return True if this polynomial is homogeneous.

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()
```

(continues on next page)
Defining x, z, y
sage: (x+y+z).is_homogeneous()
True
sage: (x.parent()(0)).is_homogeneous()
True
sage: (x+y^2+z^3).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

>> from sage.all import *
>>> A = FreeAlgebra(GF(Integer(389)), Integer(3), names=('x', 'z', 'y'))
>>> A._first_ngens(3)
(x, z, y)
>> P = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
>>> P.inject_variables()
Defining x, z, y
>>> (x+y).is_homogeneous()
True
>>> (x.parent()(Integer(0))).is_homogeneous()
True
>>> (x+y**Integer(2)+z**Integer(3)).is_homogeneous()
False
>>> (x**Integer(2) + y**Integer(2)).is_homogeneous()
True
>>> (x**Integer(2) + y**Integer(2)*x).is_homogeneous()
False
>>> (x**Integer(2)*y + y**Integer(2)*x).is_homogeneous()
True

is_monomial()
Return True if this polynomial is a monomial.
A monomial is defined to be a product of generators with coefficient 1.

EXAMPLES:
sage: 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

>> from sage.all import *
>>> A = FreeAlgebra(GF(Integer(389)), Integer(3), names=('x', 'z', 'y'))
>>> A._first_ngens(3)
(continues on next page)
>>> P = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
>>> P.inject_variables()
Defining x, z, y
>>> x.is_monomial()
True
>>> (Integer(2)*x).is_monomial()
False
>>> (x*y).is_monomial()
True
>>> (x*y + x).is_monomial()
False

is_zero()
Return True if this polynomial is zero.

EXAMPLES:

>>> from sage.all import *

lc()
Leading coefficient of this polynomial with respect to the term order of self.parent().

EXAMPLES:
```python
>>> from sage.all import *

>>> A = FreeAlgebra(GF(Integer(7)), Integer(3), names=('x', 'y', 'z',)); (x, y, z) = A._first_ngens(3)
>>> R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
>>> R.inject_variables()
Defining x, y, z

>>> f = Integer(3)*x**Integer(1)*y**Integer(2) + Integer(2)*y**Integer(3)*z**Integer(4)
>>> f.lc()
3

>>> f = Integer(5)*x**Integer(3)*y**Integer(2)*z**Integer(4) + Integer(4)*x**Integer(3)*y**Integer(2)*z**Integer(1)
>>> f.lc()
5
```

`lm()`

Return 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: 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()
Defining x, y, z
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: A.<x,y,z> = FreeAlgebra(GF(127), 3)
sage: R = A.g_algebra(relations={y*x:-x*y + z}, order='degrevlex')
sage: R.inject_variables()
Defining x, y, z
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
```
Return the 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
```
monomial_coefficient (mon)

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

This function contrasts with the function coefficient() which returns the coefficient of a monomial viewing this polynomial in a polynomial ring over a base ring having fewer variables.

INPUT:

• mon – a monomial

OUTPUT:

coefficient in base ring

See also:

For coefficients in a base ring of fewer variables, look at coefficient()

EXAMPLES:

```python
sage: 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
```

(continues on next page)
sage: f.monomial_coefficient(x^9)
388
sage: f.monomial_coefficient(x^10)
0

```python
>>> from sage.all import *
>>> A = FreeAlgebra(GF(Integer(389)), Integer(3), names=('x', 'z', 'y',)); (x, z, y,)
>>> P = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
>>> P.inject_variables()
Defining x, z, y
The parent of the return is a member of the base ring.
>>> f = Integer(2) * x * y
>>> c = f.monomial_coefficient(x*y); c
2
>>> c.parent()
Finite Field of size 389
>>> f = y**Integer(2) + y**Integer(2)*x - x**Integer(9) - Integer(7)*x +
   Integer(5)*x*y
>>> f.monomial_coefficient(y**Integer(2))
1
>>> f.monomial_coefficient(x*y)
5
>>> f.monomial_coefficient(x**Integer(9))
388
>>> f.monomial_coefficient(x**Integer(10))
0
```

`monomial_coefficients(copy=True)`

Return a dictionary representation of `self` with the keys the exponent vectors and the values the corresponding coefficients.

**INPUT:**

- “copy” – ignored

**EXAMPLES:**

```python
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 = (2*x*y^3*z^2 + (7)*x^2 + (3))
sage: d = f.monomial_coefficients(False); d
{(0, 0, 0): 3, (1, 2, 3): 2, (2, 0, 0): 7}
sage: d.clear()
sage: f.monomial_coefficients()
{(0, 0, 0): 3, (1, 2, 3): 2, (2, 0, 0): 7}
```
Noncommutative Polynomials, Release 10.4

>>> f = (Integer(2)*x*y**Integer(3)*z**Integer(2) +
    -Integer(7))*x**Integer(2) + (Integer(3))
>>> d = f.monomial_coefficients(False); d
{(0, 0, 0): 3, (1, 2, 3): 2, (2, 0, 0): 7}
>>> d.clear()
>>> f.monomial_coefficients()
{(0, 0, 0): 3, (1, 2, 3): 2, (2, 0, 0): 7}

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(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]

The result of reduction is not the normal form, if one reduces by a list of polynomials:
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: 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
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
```

(continues on next page)
>>> from sage.all import *

>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'z', 'y'),); (x, z, y,) = A._first_ngens(3)

>>> R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')

>>> R.inject_variables()

Defining x, z, y

>>> f=Integer(2)*x*y**Integer(3)*z**Integer(2)

>>> f.total_degree()

6

>>> f=Integer(4)*x**Integer(2)*y**Integer(2)*z**Integer(3)

>>> f.total_degree()

7

>>> f=Integer(99)*x**Integer(6)*y**Integer(3)*z**Integer(9)

>>> f.total_degree()

18

>>> f=x*y**Integer(3)*z**Integer(6)+Integer(3)*x**Integer(2)

>>> f.total_degree()

10

>>> f=z**Integer(9)+Integer(8)*x**Integer(4)*y**Integer(5)*z

>>> f.total_degree()

10

>>> f=z**Integer(9)+Integer(10)*x**Integer(4)*y**Integer(8)*x**Integer(2)

>>> f.total_degree()

10

sage.rings.polynomial.plural.\texttt{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:

\begin{verbatim}
>>> from sage.rings.polynomial.plural import SCA

>>> E = SCA(QQ, ['x', 'y', 'z'], [0, 1], order = 'degrevlex')

>>> E

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)

>>> E.inject_variables()

Defining xbar, ybar, zbar

>>> x,y,z = (xbar,ybar,zbar)

>>> x*y

-x*y

(continues on next page)
sage: z*x  
x*z  
sage: x^2  
0  
sage: y^2  
0  
sage: z^2  
z^2  
sage: E.one()  
1

```python
>>> from sage.all import *  
>>> from sage.rings.polynomial.plural import SCA  
>>> E = SCA(QQ, ['x', 'y', 'z'], [Integer(0), Integer(1)], order = 'degrevlex')  
>>> E  
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)  
>>> E.inject_variables()  
Defining xbar, ybar, zbar  
>>> x,y,z = (xbar,ybar,zbar)  
>>> y*x  
-x*y  
>>> z*x  
x*z  
>>> x**Integer(2)  
0  
>>> y**Integer(2)  
0  
>>> z**Integer(2)  
z^2  
>>> 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:**

```python
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")  
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
```
```python
>>> from sage.all import *
>>> H = PolynomialRing(QQ, Integer(3), names=('x', 'y', 'z')); (x, y, z, ) = H._first_ngens(3)
>>> from sage.libs.singular.function import singular_function
>>> ringlist = singular_function('ringlist')
>>> ring = singular_function("ring")

>>> L = ringlist(H, ring=H); L
[0, ['x', 'y', 'z'], [[dp', (1, 1, 1)], ['C', (0, )]], [0]]

>>> len(L)
4

>>> W = ring(L, ring=H); W
<RingWrap>

```
Construct \texttt{NCPolynomialRing\_plural} from \texttt{ringWrap}, assuming the ground field to be \texttt{base\_ring}

\textbf{EXAMPLES:}

\begin{verbatim}
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]
[0 0 0]
]
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}
\end{verbatim}

\texttt{>>> from sage.all import *}
\texttt{>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z')); (x, y, z) = A._
\rightarrow first\_ngens(3)}
\texttt{>>> H = A.g_algebra({y*x:x*y-Integer(1)})}
\texttt{>>> H.inject_variables()}
Defining x, y, z
\texttt{>>> z*x}
x*z
\texttt{>>> z*y}
y*z
\texttt{>>> y*x}
x*y - 1
\end{verbatim}
Noncommutative Polynomials, Release 10.4

(continued from previous page)

```python
>>> I = H.ideal([y**Integer(2), x**Integer(2), z**Integer(2)-Integer(1)])
>>> I._groebner_basis_libsingular()
[1]

>>> from sage.libs.singular.function import singular_function
>>> ringlist = singular_function('ringlist')
>>> ring = singular_function("ring")

>>> 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],
<BLANKLINE>
[ 0 -1 0]
[ 0 0 0]
[ 0 0 0]
]
>>> len(L)
6

>>> W = ring(L, ring=H); W

>>> from sage.rings.polynomial.plural import new_NRing
>>> R = new_NRing(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.new_Ring(rw, 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: 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]
[ 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}

>>> from sage.all import *

>>> A = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = A._first_ngens(3)

>>> H = A.g_algebra({y*x:x*y-Integer(1)})

>>> H.inject_variables()
Defining x, y, z

>>> z*x
x*z

>>> z*y
y*z

>>> y*x
x*y - 1

>>> I = H.ideal([y**Integer(2), x **Integer(2), z **Integer(2)-Integer(1)]

>>> I._groebner_basis_libsingular()
[1]

>>> from sage.libs.singular.function import singular_function

>>> ringlist = singular_function('ringlist')

>>> ring = singular_function("ring")

>>> 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]
[ 0 0 0]
]
sage: len(L)
6

>>> W = ring(L, ring=H); W
sage.rings.polynomial.plural.unpickle_NCPolynomial_plural(R, d)

Auxiliary function to unpickle a non-commutative polynomial.
CHAPTER
THREE

INDICES AND TABLES

• Index
• Module Index
• Search Page
sage.rings.polynomial.ore_function_element, 117
sage.rings.polynomial.ore_function_field, 100
sage.rings.polynomial.ore_polynomial_element, 19
sage.rings.polynomial.ore_polynomial_ring, 1
sage.rings.polynomial.plural, 129
sage.rings.polynomial.skew_polynomial_element, 74
sage.rings.polynomial.skew_polynomial_finite_field, 88
sage.rings.polynomial.skew_polynomial_finite_order, 80
sage.rings.polynomial.skew_polynomial_ring, 66
INDEX

A

algebra_generators() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 133
an_element() (sage.rings.polynomial.ore_function_element.OreFunctionBaseringInjection method), 124
an_element() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialBaseringInjection method), 60

B

base_ring() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 24
bound() (sage.rings.polynomial.skew_polynomial_finite_order.SkewPolynomial_finite_order_dense method), 80

C

center() (sage.rings.polynomial.ore_function_field.OreFunctionField_with_large_center method), 114
center() (sage.rings.polynomial.skew_polynomial_finite_order.SkewPolynomial_finite_order_dense method), 71
change_var() (sage.rings.polynomial.ore_function_field.OreFunctionField method), 105
change_var() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialRing method), 6
change_variable_name() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 24
characteristic() (sage.rings.polynomial.ore_function_field.OreFunctionField method), 105
characteristic() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 7
coefficient() (sage.rings.polynomial.plural.NCPolynomial_plural method), 141
coefficients() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 25
coefficients() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial_generic_dense method), 61
conjugate() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial_generic_dense method), 75
constant_coefficient() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 26
constant_coefficient() (sage.rings.polynomial.plural.NCPolynomial_plural method), 143
ConstantOreFunctionSection (class in sage.rings.polynomial.ore_function_element), 117
ConstantOrePolynomialSection (class in sage.rings.polynomial.ore_polynomial_element), 19
count_factorizations() (sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field_dense method), 88
count_irreducible_divisors() (sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field_dense method), 88
create_key_and_extra_args() (sage.rings.polynomial.plural.G_AlgFactory method), 132
create_object() (sage.rings.polynomial.plural.G_AlgFactory method), 133

D
degree() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 26
degree() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial_generic_dense method), 61
degree() (sage.rings.polynomial.plural.NCPolynomial_plural method), 144
degrees() (sage.rings.polynomial.plural.NCPolynomial_plural method), 144
dict() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial_generic_dense method), 62
dict() (sage.rings.polynomial.plural.NCPolynomial_plural method), 145
Element (sage.rings.polynomial.ore_function_field.OreFunctionField attribute), 105
Element (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing attribute), 6
exponents() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 27
exponents() (sage.rings.polynomial.plural.NCPolynomial_plural method), 145
ExteriorAlgebra() (in module sage.rings.polynomial.plural), 131
ExteriorAlgebra_plural (class in sage.rings.polynomial.plural), 132
factor() (sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field_dense method), 89
factorizations() (sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field_dense method), 91
fraction_field() (sage.rings.polynomial.ore_function_field.OreFunctionField method), 106
fraction_field() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing method), 8
free_algebra() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 134
G
G_AlgFactory (class in sage.rings.polynomial.plural), 132
gen() (sage.rings.polynomial.ore_function_field.OreFunctionField method), 107
gen() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing method), 9
gen() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 134
gens() (sage.rings.polynomial.ore_function_field.OreFunctionField method), 107
gens() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing method), 9
gens_dict() (sage.rings.polynomial.ore_function_field.OreFunctionField method), 108
gens_dict() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing method), 10
H
hamming_weight() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 27
hilbert_shift() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 118
hilbert_shift() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial_generic_dense method), 62
ideal() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 134
is_central() (sage.rings.polynomial.skew_polynomial_ring.OrePolynomialFiniteOrder method), 82
is_commutative() (sage.rings.polynomial.ore_function_field.OreFunctionField method), 108
is_commutative() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing method), 10
is_commutative() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 135
is_constant() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 28
is_constant() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 146
is_exact() (sage.rings.polynomial.ore_function_field.OreFunctionField method), 108
is_exact() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing method), 11
is_field() (sage.rings.polynomial.ore_function_field.OreFunctionField method), 109
is_field() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing method), 12
is_field() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 136
is_finite() (sage.rings.polynomial.ore_function_field.OreFunctionField method), 110
is_finite() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing method), 12
is_homogeneous() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 146
is_irreducible() (sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field_dense method), 92
is_left_divisible_by() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 28
is_monic() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 29
is_monomial() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 30
is_monomial() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 147
is_nilpotent() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 31
is_one() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 31
is_right_divisible_by() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 31
is_sparse() (sage.rings.polynomial.ore_function_field.OreFunctionField method), 110
is_sparse() (sage.rings.polynomial.ore_polyno-
mial_ring.OrePolynomialRing method), 12
is_term() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 33
is_unit() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 34
is_zero() (sage.rings.polynomial.ore_function_element.OreFunction method), 120
is_zero() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 34
is_zero() (sage.rings.polynomial.plural.NCPolynomial_plural method), 148
lagrange_polynomial() (sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing method), 67
lc() (sage.rings.polynomial.plural.NCPolynomial_plural method), 148
leading_coefficient() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 34
left_denominator() (sage.rings.polynomial.ore_function_element.OreFunction method), 120
left_divides() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 35
left_gcd() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 36
left_irreducible_divisor() (sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field_dense method), 94
left_irreducible_divisors() (sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field_dense method), 95
left_lcm() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 38
left_mod() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 39
left_monic() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 40
left_numerator() (sage.rings.polynomial.ore_function_element.OreFunction method), 122
left_power_mod() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial_generic_dense method), 76
left_quo_rem() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 41
left_xgcd() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 43
left_xlcm() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 45
list() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial_generic_dense method), 64
lm() (sage.rings.polynomial.plural.NCPolynomial_plural method), 149
lt() (sage.rings.polynomial.plural.NCPolynomial_plural method), 150

M
minimal_vanishing_polynomial() (sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing method), 69
module
sage.rings.polynomial.ore_function_element, 117
sage.rings.polynomial.ore_function_field, 100
sage.rings.polynomial.ore_polynomial_element, 19
sage.rings.polynomial.ore_polynomial_ring, 1
sage.rings.polynomial.plural, 129
sage.rings.polynomial.skew_polynomial_element, 74
sage.rings.polynomial.skew_polynomial_finite_field, 88
sage.rings.polynomial.skew_polynomial_finite_field_dense, 80
sage.rings.polynomial.skew_polynomial_ring, 66
monomial_all_divisors() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 136
monomial_coefficient() (sage.rings.polynomial.plural.NCPolynomial_plural method), 151
monomial_coefficients() (sage.rings.polynomial.plural.NCPolynomial_plural method), 152
monomial_divides() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 136
monomial_lcm() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 137
monomial_pairwise_prime() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 138
monomial_quotient() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 138
monomial_reduce() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 138
multi_point_evaluation() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial_generic_dense method), 76

Index 169
### N
- NCPolynomial_plural (class in sage.rings.polynomial.plural), 141
- NCPolynomialRing_plural (class in sage.rings.polynomial.plural), 133
- new_CRing() (in module sage.rings.polynomial.plural), 156
- new_NRing() (in module sage.rings.polynomial.plural), 157
- new_Ring() (in module sage.rings.polynomial.plural), 159
- ngens() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 46
- number_of_terms() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 46

### O
- operator_eval() (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial_generic_dense method), 77
- optimal_bound() (sage.rings.polynomial.skew_polynomial_finite_order.SkewPolynomial_finite_order_dense method), 82
- OreFunction (class in sage.rings.polynomial.ore_function_element), 118
- OreFunction_with_large_center (class in sage.rings.polynomial.ore_function_element), 125
- OreFunctionBaseringInjection (class in sage.rings.polynomial.ore_function_element), 124
- OreFunctionCenterInjection (class in sage.rings.polynomial.ore_function_field), 104
- OreFunctionField (class in sage.rings.polynomial.ore_function_field), 105
- OreFunctionField_with_large_center (class in sage.rings.polynomial.ore_function_field), 114
- OrePolynomial (class in sage.rings.polynomial.ore_polynomial_element), 19
- OrePolynomial_generic_dense (class in sage.rings.polynomial.ore_polynomial_element), 61
- OrePolynomialBaseringInjection (class in sage.rings.polynomial.ore_polynomial_element), 59
- OrePolynomialRing (class in sage.rings.polynomial.ore_polynomial_ring), 1

### P
- padded_list() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 46
- parameter() (sage.rings.polynomial.ore_function_field.OreFunctionField method), 111
- parameter() (sage.rings.polynomial.ore_polynomial.ore_polynomial_element.OrePolynomial method), 13
- prec() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 47
- random_element() (sage.rings.polynomial.ore_function_field.OreFunctionField method), 111
- random_element() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing method), 14
- random_irreducible() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing method), 15
- reduce() (sage.rings.polynomial.plural.NCPolynomial_plural method), 153
- reduced_charpoly() (sage.rings.polynomial.skew_polynomial_finite_order.SkewPolynomial_finite_order_dense method), 83
- reduced_norm() (sage.rings.polynomial.ore_function_element.OreFunction_with_large_center method), 125
- reduced_norm() (sage.rings.polynomial.skew_polynomial_finite_order.SkewPolynomial_finite_order_dense method), 85
- reduced_trace() (sage.rings.polynomial.ore_function_element.OreFunction_with_large_center method), 126
- reduced_trace() (sage.rings.polynomial.skew_polynomial_finite_order.SkewPolynomial_finite_order_dense method), 86
- relations() (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 140
- right_denominator() (sage.rings.polynomial.ore_function_element.OreFunction method), 122
- right_divides() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 47
- right_gcd() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 49
- right_irreducible_divisor() (sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field_dense method), 96
- right_irreducible_divisors() (sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field_dense method), 97
- right_lcm() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 50
Noncommutative Polynomials, Release 10.4

right_mod()  (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 52
right_monic()  (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 53
right_numerator()  (sage.rings.polynomial.ore_function_element.OreFunction method), 123
right_power_mod()  (sage.rings.polynomial.skew_polynomial_element.SkewPolynomial method), 78
right_quo_rem()  (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 54
right_xgcd()  (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 55
right_xlcm()  (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 57

S
sage.rings.polynomial.ore_function_element
module, 117
sage.rings.polynomial.ore_function_field
module, 100
sage.rings.polynomial.ore_polynomial_element
module, 19
sage.rings.polynomial.ore_polynomial_ring
module, 1
sage.rings.polynomial.plural
module, 129
sage.rings.polynomial.skew_polynomial_element
module, 74
sage.rings.polynomial.skew_polynomial_finite_field
module, 88
sage.rings.polynomial.skew_polynomial_finite_order
module, 80
sage.rings.polynomial.skew_polynomial_ring
module, 66
SCA()  (in module sage.rings.polynomial.plural), 155
section()  (sage.rings.polynomial.ore_function_element.OrePolynomial method), 125
section()  (sage.rings.polynomial.ore_function_field.OrePolynomial method), 104
section()  (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 60

section()  (sage.rings.polynomial.skew_polynomial_element.OrePolynomial method), 52
SectionOreFunctionCenterInjection  (class in sage.rings.polynomial.ore_function_field), 117
SectionSkewPolynomialCenterInjection  (class in sage.rings.polynomial.skew_polynomial_element), 66
shift()  (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 58
SkewPolynomial_finite_field_dense  (class in sage.rings.polynomial.skew_polynomial_finite_field), 88
SkewPolynomial_finite_order_dense  (class in sage.rings.polynomial.skew_polynomial_finite_order), 80
SkewPolynomial_generic_dense  (class in sage.rings.polynomial.skew_polynomial_element), 74
SkewPolynomialCenterInjection  (class in sage.rings.polynomial.skew_polynomial_element), 66
SkewPolynomialRing  (class in sage.rings.polynomial.skew_polynomial_element), 67
SkewPolynomialRing_finite_field  (class in sage.rings.polynomial.skew_polynomial_element), 70
SkewPolynomialRing_finite_order  (class in sage.rings.polynomial.skew_polynomial_element), 70
square()  (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 58

T
term_order()  (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 141
total_degree()  (sage.rings.polynomial.plural.NCPolynomialRing_plural method), 154
truncate()  (sage.rings.polynomial.ore_polynomial_element.OrePolynomial_generic_dense method), 65
twisting_derivation()  (sage.rings.polynomial.ore_function_field.OreFunctionField method), 112
twisting_derivation()  (sage.rings.polynomial.ore_polynomial_ring.OrePolynomial method), 16
twisting_morphism()  (sage.rings.polynomial.ore_function_field.OreFunctionField method), 113
twisting_morphism()  (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 16

Index 171
type()  (sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field_dense method), 98

U
unpickle_NCPolynomial_plural() (in module sage.rings.polynomial.plural), 161

V
valuation()  (sage.rings.polynomial.ore_polynomial_element.OrePolynomial_generic_dense method), 65
variable_name()  (sage.rings.polynomial.ore_polynomial_element.OrePolynomial method), 59