
Noncommutative Polynomials

Release 10.2

The Sage Development Team

Dec 06, 2023

CONTENTS

1	Univariate Ore Polynomials	1
2	Noncommutative Multivariate Polynomials	79
3	Indices and Tables	99
	Python Module Index	101
	Index	103

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, morphism,  
                                                                    derivation, name, sparse,  
                                                                    category=None)
```

Bases: *UniqueRepresentation, Algebra*

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

Given a ring R and a ring automorphism σ of R and a σ -derivation ∂ , 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 $\mathbb{F}_{5^3}[x, \text{Frob}]$ where `Frob` is the Frobenius endomorphism:

```
sage: k.<a> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S = OrePolynomialRing(k, Frob, 'x')
sage: 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: T.<x> = k['x', Frob]
sage: 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]
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
True
```

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

```
sage: x*a
(2*a^2 + 4*a + 4)*x
sage: Frob(a)*x
(2*a^2 + 4*a + 4)*x
```

The case of a twisting derivation

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

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

Again, the brackets notation is available:

```
sage: B.<d> = R['d', der]
sage: A is B
True
```

and computations can be carried out:

```
sage: d*t
t*d + 1
```

The combined case

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

```
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 |--> u^3 on the residue field
sage: der = F.derivation(3, twist=sigma); der
(3 + 0(3^21))*([Frob] - id)

sage: M.<X> = F['X', der]
sage: 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 + 0(3^21))*([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]
Traceback (most recent call last):
...
ValueError: variable name 'Frobenius endomorphism ...' is not alphanumeric
```

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: 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

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

Rings with different variables names are different:

```
sage: S is k['y', Frob]
False
```

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

```
sage: S is k['x', Frob^2]
False
sage: S is k['x', Frob^3]
False
sage: S is k['x', Frob^4]
True
```

Todo:

- Sparse Ore Polynomial Ring
 - Multivariate Ore Polynomial Ring
-

Element = None

`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:

```
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:

```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: R['x',sigma].characteristic()
0

sage: k.<u> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: k['y',Frob].characteristic()
5
```

fraction_field()

Return the fraction field of this skew ring.

EXAMPLES:

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

sage: f = 1/(x + a); f
(x + a)^(-1)
sage: f.parent() is K
True
```

Below is another example with differential operators:

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

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

See also:

[*sage.rings.polynomial.ore_function_field*](#)

gen(*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:

```

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: y is x
True
sage: S.gen(0)
x

```

This is also known as the parameter:

```

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

```

gens_dict()

Return a {name: variable} dictionary of the generators of 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}

```

is_commutative()

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

EXAMPLES:

```

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

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

```

(continues on next page)

(continued from previous page)

```
sage: 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: 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
```

is_field(*proof=False*)

Return always False since Ore polynomial rings are never fields.

EXAMPLES:

```
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:

```
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:

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

ngens()

Return the number of generators of this Ore polynomial ring, which is 1.

EXAMPLES:

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

```
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: y is x
True
sage: S.gen(0)
x
```

This is also known as the parameter:

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

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

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`, `**kwargs` – 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 `**kwargs`.

EXAMPLES:

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x', Frob]
sage: S.random_element() # random
(2*t^2 + 3)*x^2 + (4*t^2 + t + 4)*x + 2*t^2 + 2
sage: S.random_element(monic=True) # random
x^2 + (2*t^2 + t + 1)*x + 3*t^2 + 3*t + 2
```

Use `degree` to obtain polynomials of higher degree:

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

If a tuple of two integers is given for the `degree` argument, a random integer will be chosen between the first and second element of the tuple as the degree, both inclusive:

```
sage: S.random_element(degree=(2,7)) # random
(3*t^2 + 1)*x^4 + (4*t + 2)*x^3 + (4*t + 1)*x^2
+ (t^2 + 3*t + 3)*x + 3*t^2 + 2*t + 2
```

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

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`, `**kwargs` - passed in to the `random_element` method for the base ring

EXAMPLES:

```
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)
```

(continues on next page)

(continued from previous page)

```
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:

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

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=1)

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:

```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x', sigma]
sage: S.twisting_morphism()
Ring endomorphism of Univariate Polynomial Ring in t over Rational Field
Defn: t |--> t + 1
sage: S.twisting_morphism() == sigma
True
sage: S.twisting_morphism(10)
Ring endomorphism of Univariate Polynomial Ring in t over Rational Field
Defn: t |--> t + 10
```

If *n* is negative, Sage tries to compute the inverse of the twisting morphism:

```
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:

```

sage: K = R.fraction_field()
sage: phi = K.hom([(t+1)/(t-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

```

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: 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 `OrePolynomialBasingInjection` handle conversion from an Ore polynomial ring to its base ring and vice versa.

AUTHORS:

- Xavier Caruso (2020-05)

class `sage.rings.polynomial.ore_polynomial_element.ConstantOrePolynomialSection`

Bases: `Map`

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:

```

sage: from sage.rings.polynomial.ore_polynomial_element import ConstantOrePolynomialSection
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

```

```
class sage.rings.polynomial.ore_polynomial_element.OrePolynomial
```

Bases: `AlgebraElement`

Abstract base class for Ore polynomials.

This class must be inherited from and have key methods overridden.

Definition

Let R be a commutative ring equipped with an automorphism σ and a σ -derivation ∂ .

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):


```

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

```

and some elements in it:

```

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

```

Ring operations are supported:

```

sage: a + b
x^2 + (t + 2)*x + t^2 + t + 1
sage: a - b
-x^2 - t*x - t^2 + t + 1

sage: a * b
x^3 + (2*t + 3)*x^2 + (2*t^2 + 4*t + 2)*x + t^3 + t^2
sage: b * a
x^3 + (2*t + 4)*x^2 + (2*t^2 + 3*t + 2)*x + t^3 + t^2
sage: a * b == b * a
False

sage: b^2
x^4 + (2*t + 4)*x^3 + (3*t^2 + 7*t + 6)*x^2
+ (2*t^3 + 4*t^2 + 3*t + 1)*x + t^4
sage: b^2 == b*b
True

```

Sage also implements arithmetic over Ore polynomial rings. You will find below a short panorama:

```

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

```

The operators `//` and `%` give respectively the quotient and the remainder of the *right* Euclidean division:

```

sage: q == c // b
True
sage: r == c % b
True

```

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

```

sage: a = x^2
sage: a(t)
doctest:...: FutureWarning: This class/method/function is marked as experimental.

```

(continues on next page)

(continued from previous page)

It, its functionality or its interface might change without a formal deprecation.
See <https://github.com/sagemath/sage/issues/13215> for details.

$t + 2$

Here is another example over a finite field:

```
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^4 + (4*t + 1)*x^3 + (t^2 + 3*t + 3)*x^2 + (3*t^2 + 2*t + 2)*x + (3*t^2 + 1)
sage: b = (2*t^2 + 3)*x^2 + (3*t^2 + 1)*x + 4*t + 2
sage: q, r = a.left_quo_rem(b)
sage: q
(4*t^2 + t + 1)*x^2 + (2*t^2 + 2*t + 2)*x + 2*t^2 + 4*t + 3
sage: r
(t + 2)*x + 3*t^2 + 2*t + 4
sage: a == b*q + r
True
```

Once we have Euclidean divisions, we have for free gcd and lcm (at least if the base ring is a field):

```
sage: # needs sage.rings.finite_rings
sage: a = (x + t) * (x + t^2)^2
sage: b = (x + t) * (t*x + t + 1) * (x + t^2)
sage: a.right_gcd(b)
x + t^2
sage: 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: 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
sage: c.is_right_divisible_by(a)
True
sage: c.is_right_divisible_by(b)
True
```

The right lcm is defined similarly as the least degree polynomial $c = au = bv$ for some u, v :

```
sage: d = a.right_lcm(b); d
x^5 + (t^2 + 1)*x^4 + (3*t^2 + 3*t + 3)*x^3 + (3*t^2 + t + 2)*x^2 + (4*t^2 + 3*t)*x + 4*t + 4
sage: d.is_left_divisible_by(a)
True
```

(continues on next page)

(continued from previous page)

```
True
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:

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

`change_variable_name(var)`

Change the name of the variable of `self`.

This will create the 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]
sage: a.coefficients(sparse=False)
[t^2 + 1, 0, t + 1, 0, 1]
```

constant_coefficient()

Return the constant coefficient (i.e., the coefficient of term of degree 0) of *self*.

EXAMPLES:

```
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:

```
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
```

exponents()

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

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = 1 + x^4 + (t+1)*x^2 + t^2
sage: a.exponents()
[0, 2, 4]
```

hamming_weight()

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

This is also known as the weight, hamming weight or sparsity.

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = 1 + x^4 + (t+1)*x^2 + t^2
sage: a.number_of_terms()
3
```

This is also an alias for `hamming_weight`:

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

is_constant()

Return whether `self` is a constant polynomial.

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: R(2).is_constant()
True
sage: (x + 1).is_constant()
False
```

is_left_divisible_by(*other*)

Check if `self` is divisible by `other` on the left.

INPUT:

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

OUTPUT:

Return True or False.

EXAMPLES:

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

Divisibility by 0 does not make sense:

```

sage: c.is_left_divisible_by(S(0))
Traceback (most recent call last):
...
ZeroDivisionError: division by zero is not valid

```

is_monic()

Return True if this Ore polynomial is monic.

The zero polynomial is by definition not monic.

EXAMPLES:

```

sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x + t
sage: a.is_monic()
True
sage: a = 0*x
sage: a.is_monic()
False
sage: a = t*x^3 + x^4 + (t+1)*x^2
sage: a.is_monic()
True
sage: a = (t^2 + 2*t)*x^2 + x^3 + t^10*x^5
sage: a.is_monic()
False

```

is_monomial()

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

EXAMPLES:

```

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

```

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:

```
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
```

is_one()

Test whether this polynomial is 1.

EXAMPLES:

```
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
```

is_right_divisible_by(*other*)

Check if `self` is divisible by `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
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
```

(continues on next page)

(continued from previous page)

```
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:

```
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:

```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x^2 + 2*x + t
sage: b = (t+1)*x + t^2
sage: c = a*b
sage: c.is_right_divisible_by(b)
Traceback (most recent call last):
...
NotImplementedError: the leading coefficient of the divisor is not invertible
```

is_term()

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

EXAMPLES:

```
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: x.is_term()
True
sage: R(1).is_term()
True
sage: (3*x^5).is_term()
True
sage: (1+3*x^5).is_term()
False
```

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

```
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:

```
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:

```
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x + 1
sage: a.is_zero()
False
sage: b = S.zero()
sage: b.is_zero()
True
```

leading_coefficient()

Return the coefficient of the highest-degree monomial of `self`.

EXAMPLES:

```
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:

```
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:

```
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
```

Divisibility by 0 does not make sense:

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

left_gcd(other, monic=True)

Return the left gcd of **self** and **other**.

INPUT:

- **other** – 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:

```
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:

```
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:

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

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

(continues on next page)

(continued from previous page)

```

x^5 + (2*t^2 + t + 4)*x^4 + (3*t^2 + 4)*x^3 + (3*t^2 + 3*t + 2)*x^2 + (t^2 + t + 2)*x
sage: c.is_right_divisible_by(a)
True
sage: c.is_right_divisible_by(b)
True
sage: a.degree() + b.degree() == c.degree() + a.right_gcd(b).degree()
True

```

Specifying `monic=False`, we *can* get a nonmonic lcm:

```

sage: a.left_lcm(b, monic=False) #
↳needs sage.rings.finite_rings
(t^2 + t)*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

```

`left_mod(other)`

Return the remainder of left division of `self` by `other`.

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 = 1 + t*x^2
sage: b = x + 1
sage: a.left_mod(b)
2*t^2 + 4*t

```

`left_mononic()`

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 σ is the twisting morphism.

EXAMPLES:

```

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

```

(continues on next page)

(continued from previous page)

```

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: # needs sage.rings.finite_rings
sage: b.degree() == a.degree()
True
sage: a.is_left_divisible_by(b)
True
sage: twist = S.twisting_morphism(-a.degree())
sage: a == b * twist(a.leading_coefficient())
True

```

Note that b does not divide a on the right:

```

sage: 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:

```

sage: R.<t> = QQ[]
sage: der = R.derivation()
sage: S.<x> = R['x', der]
sage: a = t*x
sage: a.left_monic()
Traceback (most recent call last):
...
NotImplementedError: 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 `other` is not a unit or if Sage can't invert the twisting morphism.

EXAMPLES:

```

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

```

(continues on next page)

(continued from previous page)

```

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 = (3*t^2 + 4*t + 2)*x^2 + (2*t^2 + 4*t + 3)*x + 2*t^2 + t + 1
sage: q,r = a.left_quo_rem(b)
sage: a == b*q + r
True

```

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

```

sage: R.<t> = QQ[]
sage: K = R.fraction_field()
sage: sigma = K.hom([(t+1)/(t-1)])
sage: S.<x> = K['x',sigma]
sage: a = (-2*t^2 - t + 1)*x^3 + (-t^2 + t)*x^2 + (-12*t - 2)*x - t^2 - 95*t + 1
sage: b = x^2 + (5*t - 6)*x - 4*t^2 + 4*t - 1
sage: a.left_quo_rem(b)
Traceback (most recent call last):
...
NotImplementedError: inversion of the twisting morphism Ring endomorphism of
Fraction Field of Univariate Polynomial Ring in t over Rational Field
Defn: t |--> (t + 1)/(t - 1)

```

`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:

```

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

```

(continues on next page)

(continued from previous page)

```

sage: S.<x> = k['x',Frob]
sage: a = (x + t) * (x^2 + t*x + 1)
sage: b = 2 * (x + t) * (x^3 + (t+1)*x^2 + t^2)
sage: g,u,v = a.left_xgcd(b); g
x + t
sage: a*u + b*v == g
True

```

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

```

sage: g,u,v = a.left_xgcd(b, monic=False); g #_
↳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:

```

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

```

And the twisting morphism must be bijective:

```

sage: FR = R.fraction_field()
sage: f = FR.hom([FR(t)^2])
sage: S.<x> = FR['x',f]
sage: a = (x + t) * (x^2 + t*x + 1)
sage: b = 2 * (x + t) * (x^3 + (t+1)*x^2 + t^2)
sage: a.left_xgcd(b)
Traceback (most recent call last):
...
NotImplementedError: inversion of the 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

$$U \cdot \text{self} = V \cdot \text{other} = L.$$

EXAMPLES:

```

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

```

(continues on next page)

(continued from previous page)

```

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

```

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 list on the right 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]
sage: a.padded_list(10)
[1, 0, 0, t, 0, t^2, 0, 0, 0, 0]
sage: len(a.padded_list(10))
10

```

(continues on next page)

(continued from previous page)

```

sage: a.padded_list(3)
[1, 0, 0]
sage: a.padded_list(0)
[]
sage: a.padded_list(-1)
Traceback (most recent call last):
...
ValueError: n must be at least 0

```

prec()

Return the precision of self.

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

EXAMPLES:

```

sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: x.prec()
+Infinity

```

right_divides(other)

Check if self divides other on the right.

INPUT:

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

OUTPUT:

Return True or False.

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^2 + t*x + t^2 + 3
sage: b = x^3 + (t + 1)*x^2 + 1
sage: c = a * b
sage: a.right_divides(c)
False
sage: b.right_divides(c)
True

```

Divisibility by 0 does not make sense:

```

sage: S(0).right_divides(c)
↳ 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[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x^2 + 2*x + t
sage: b = (t+1)*x + t^2
sage: c = a*b
sage: b.right_divides(c)
Traceback (most recent call last):
...
NotImplementedError: the leading coefficient of the divisor is not invertible
```

right_gcd(other, monic=True)

Return the right gcd of self and other.

INPUT:

- other – 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:

```
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 + 1) * (x + t)
sage: b = 2 * (x^3 + (t+1)*x^2 + t^2) * (x + t)
sage: a.right_gcd(b)
x + t
```

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

```
sage: a.right_gcd(b,monic=False) #
↳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):
```

(continues on next page)

(continued from previous page)

```
...
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)
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 + 1
sage: c.is_left_divisible_by(a)
True
sage: c.is_left_divisible_by(b)
True
sage: a.degree() + b.degree() == c.degree() + a.left_gcd(b).degree()
True
```

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

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

The base ring needs to be a field:

```
sage: 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)
```

(continues on next page)

(continued from previous page)

```
sage: 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)^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 Ring endomorphism of
Fraction Field of Univariate Polynomial Ring in t over Rational Field
Defn: t |--> t^2
```

right_mod(*other*)

Return the remainder of right division of *self* by *other*.

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = 1 + t*x^2
sage: b = x + 1
sage: a % b
t + 1
sage: (x^3 + x - 1).right_mod(x^2 - 1)
2*x - 1
```

right_monic()

Return the unique monic 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:

```
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
```

Check list:

```

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

```

Note that b does not divide a on the right:

```

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:

```

sage: R.<t> = QQ[]
sage: der = R.derivation()
sage: S.<x> = R['x', der]
sage: a = t*x
sage: 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:

```

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

```

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

```

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

Return the right gcd of *self* and *other* along with the coefficients for the linear combination.

If *a* is *self* and *b* is *other*, then there are 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:

- *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.)
- 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:

```

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 + 1) * (x + t)
sage: b = 2 * (x^3 + (t+1)*x^2 + t^2) * (x + t)
sage: g,u,v = a.right_xgcd(b); g
x + t
sage: u*a + v*b == g
True

```

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

```

sage: g,u,v = a.right_xgcd(b, monic=False); g
↪needs sage.rings.finite_rings
(4*t^2 + 4*t + 1)*x + 4*t^2 + 4*t + 3
sage: u*a + v*b == g
↪needs sage.rings.finite_rings
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

```

right_xlcm(other, monic=True)

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

$$\text{self} \cdot U = \text{other} \cdot V = L.$$

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:

```

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 = 2 * (x + t) * (x^2 + t + 1)
sage: L, U, V = P.right_xlcm(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
↪4
sage: P * U == L
True
sage: 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['x',sigma]
sage: a = x^5 + t^4*x^4 + t^2*x^2 + t^10
sage: a.shift(0)
x^5 + t^4*x^4 + t^2*x^2 + t^10
sage: a.shift(-1)
x^4 + t^4*x^3 + t^2*x
sage: a.shift(-5)
1
sage: a.shift(2)
x^7 + t^4*x^6 + t^2*x^4 + t^10*x^2

```

One can also use the infix shift operator:

```

sage: a >> 2
x^3 + t^4*x^2 + t^2
sage: a << 2
x^7 + t^4*x^6 + t^2*x^4 + t^10*x^2

```

square()

Return the square of self.

EXAMPLES:

```

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

sage: der = R.derivation()
sage: A.<d> = R['d', der]
sage: (d + t).square()
d^2 + 2*t*d + t^2 + 1

```

variable_name()

Return the string name of the variable used in self.

EXAMPLES:

```

sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x + t
sage: a.variable_name()
'x'

```

class sage.rings.polynomial.ore_polynomial_element.OrePolynomialBasingInjection

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:

`PolynomialBasingInjection`

EXAMPLES:

```
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
```

`an_element()`

Return an element of the codomain of the ring homomorphism.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: from sage.rings.polynomial.ore_polynomial_element import _
      _OrePolynomialBasingInjection
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: m = OrePolynomialBasingInjection(k, k['x', Frob])
sage: m.an_element()
x
```

`section()`

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

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:

```
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)

(continued from previous page)

```
sage: a.coefficients(sparse=False)
[t^2 + 1, 0, t + 1, 0, 1]
```

degree()

Return the degree of `self`.

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

EXAMPLES:

```
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x^2 + t*x^3 + t^2*x + 1
sage: a.degree()
3
```

By convention, the degree of 0 is -1 :

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

dict()

Return a dictionary representation of `self`.

EXAMPLES:

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

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
sage: (d+t)^3 + t*(d+t)^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 an 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
```

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

```
sage: # needs sage.rings.finite_rings
sage: P = S.random_element(degree=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:

```
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]
```

Note that *l* is a list, it is mutable, and each call to the list method returns a new list:

```
sage: type(l)
<... 'list'>
sage: l[0] = 5
```

(continues on next page)

(continued from previous page)

```
sage: a.list()
[t^2 + 1, 0, t + 1, 0, 1]
```

truncate(*n*)

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

EXAMPLES:

```
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = t*x^3 + x^4 + (t+1)*x^2
sage: a.truncate(4)
t*x^3 + (t + 1)*x^2
sage: a.truncate(3)
(t + 1)*x^2
```

valuation()

Return the minimal degree of a non-zero monomial of `self`.

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

EXAMPLES:

```
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t+1])
sage: S.<x> = R['x',sigma]
sage: a = x^2 + t*x^3 + t^2*x
sage: a.valuation()
1
```

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

```
sage: S(0).valuation()
+Infinity
```

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

class sage.rings.polynomial.skew_polynomial_ring.**SectionSkewPolynomialCenterInjection**

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:

```
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()
sage: sigma(x^3)
z
```

class sage.rings.polynomial.skew_polynomial_ring.**SkewPolynomialRing**(*base_ring*, *morphism*,
derivation, *name*, *sparse*,
category=None)

Bases: [OrePolynomialRing](#)

Initialize self.

INPUT:

- *base_ring* – a commutative ring
- *twisting_morphism* – an automorphism of the base ring
- *name* – string or list of strings representing the name of the variables of ring
- *sparse* – boolean (default: False)
- *category* – a category

EXAMPLES:

```
sage: R.<t> = ZZ[]
sage: sigma = R.hom([t + 1])
sage: S.<x> = SkewPolynomialRing(R, sigma)
sage: S.category()
Category of algebras over Univariate Polynomial Ring in t over Integer Ring
sage: S([1]) + S([-1])
```

(continues on next page)

(continued from previous page)

0

```
sage: TestSuite(S).run()
```

`lagrange_polynomial(points)`

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

More precisely, given n pairs $(x_1, y_1), \dots, (x_n, y_n) \in R^2$, 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 $(x_1, y_1), \dots, (x_n, y_n)$ 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:

```
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x', Frob]
sage: points = [(t, 3*t^2 + 4*t + 4), (t^2, 4*t)]
sage: d = S.lagrange_polynomial(points); d
x + t

sage: R.<t> = ZZ[]
sage: sigma = R.hom([t + 1])
sage: T.<x> = R['x', sigma]
sage: points = [(1, t^2 + 3*t + 4), (t, 2*t^2 + 3*t + 1), (t^2, t^2 + 3*t + 4)]
sage: p = T.lagrange_polynomial(points); p
((-t^4 - 2*t - 3)/-2)*x^2 + (-t^4 - t^3 - t^2 - 3*t - 2)*x
+ (-t^4 - 2*t^3 - 4*t^2 - 10*t - 9)/-2
sage: p.multi_point_evaluation([1, t, t^2]) == [t^2 + 3*t + 4, 2*t^2 + 3*t + 1,
↪ t^2 + 3*t + 4]
True
```

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

```
sage: T.lagrange_polynomial([(t, 1), (2*t, 3)])
Traceback (most recent call last):
...
ValueError: the given evaluation points are linearly dependent over the fixed_
↪ field of the 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:

```
sage: # needs sage.rings.finite_rings
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x', Frob]
sage: eval_pts = [1, t, t^2]
sage: b = S.minimal_vanishing_polynomial(eval_pts); b
x^3 + 4
```

The minimal vanishing polynomial evaluates to 0 at each of the evaluation points:

```
sage: eval = b.multi_point_evaluation(eval_pts); eval
[0, 0, 0]
```

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:

```
sage: S.minimal_vanishing_polynomial([t])
x + 3*t^2 + 3*t
sage: S.minimal_vanishing_polynomial([t, 3*t])
x + 3*t^2 + 3*t
```

```
class sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing_finite_field(base_ring,
                                                                                     morphism,
                                                                                     derivation,
                                                                                     names,
                                                                                     sparse,
                                                                                     cate-
                                                                                     gory=None)
```

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,
                                                                                   cate-
                                                                                   gory=None)
```

Bases: *SkewPolynomialRing*

A specialized class for skew polynomial rings whose twisting 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 σ and σ 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: # 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
```

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
```

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)
True
sage: P = y + x; P
x^3 + x
sage: P.parent()
Ore Polynomial Ring in x over Finite Field in t of size 5^3
twisted by t |--> t^5
sage: P.parent() is S
True
```

together with a conversion map in the reverse direction:

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

sage: Zy(x^2) #_
↪needs sage.rings.finite_rings
Traceback (most recent call last):
...
ValueError: x^2 is not in the center
```

Two different skew polynomial rings can share the same center:

```
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
```

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

We can update the default variable name by passing in the argument `default=True`:

```
sage: # needs sage.rings.finite_rings
sage: D.<v> = A.center(default=True)
sage: D
Univariate Polynomial Ring in v over Finite Field of size 11
sage: A.center()
Univariate Polynomial Ring in v 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[]
sage: K = R.fraction_field()
sage: sigma = K.hom([1 + 1/t])
sage: S.<x> = K['x',sigma]
sage: a = t*x^3 + (t^2 + 1)*x^2 + 2*t
sage: b = a.conjugate(2); b
((2*t + 1)/(t + 1))*x^3 + ((5*t^2 + 6*t + 2)/(t^2 + 2*t + 1))*x^2 + (4*t + 2)/
↪(t + 1)
sage: x^2*a == b*x^2
True
```

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

```
sage: b = a.conjugate(-1)
Traceback (most recent call last):
...
NotImplementedError: inverse not implemented for morphisms of Fraction Field of
↳ Univariate Polynomial Ring in t over Rational Field
```

Here is a working example:

```
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
```

`left_power_mod(exp, modulus)`

Return the remainder of `self**exp` in the left euclidean division by `modulus`.

INPUT:

- `exp` – an Integer
- `modulus` – a skew polynomial in the same ring as `self`

OUTPUT:

Remainder of `self**exp` in the left euclidean division by `modulus`.

REMARK:

The quotient of the underlying skew polynomial ring by the principal ideal generated by `modulus` is in general *not* a ring.

As a consequence, Sage first computes exactly `self**exp` and then reduce it modulo `modulus`.

EXAMPLES:

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = x + t
sage: modulus = x^3 + t*x^2 + (t+3)*x - 2
sage: a.left_power_mod(100,modulus)
(4*t^2 + t + 1)*x^2 + (t^2 + 4*t + 1)*x + 3*t^2 + 3*t
```

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

operator_eval(*eval_pt*)

Evaluate self at *eval_pt* by the operator evaluation method.

INPUT:

- *eval_pt* – element of the base ring of self

OUTPUT:

The value of the polynomial at the point specified by the argument.

EXAMPLES:

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: T.<x> = k['x',Frob]
sage: a = 3*t^2*x^2 + (t + 1)*x + 2
sage: a(t) #indirect test
2*t^2 + 2*t + 3
sage: a.operator_eval(t)
2*t^2 + 2*t + 3
```

Evaluation points outside the base ring is usually not possible due to the 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
```

right_power_mod(*exp*, *modulus*)

Return the remainder of `self**exp` in the right 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 right euclidean division by `modulus`.

REMARK:

The quotient of the underlying skew polynomial ring by the principal ideal generated by `modulus` is in general *not* a ring.

As a consequence, Sage first computes exactly `self**exp` and then reduce it modulo `modulus`.

EXAMPLES:

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = x + t
sage: 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: 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)
Traceback (most recent call last):
...
ValueError: modulus cannot be combined with negative exponent
```

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

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. If it 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:

```
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

sage: a = x^2 + (4*t + 2)*x + 4*t^2 + 3
sage: b = a.bound(); b
z^2 + z + 4
```

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:

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

We check that b is divisible by a :

```
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 :

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

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

```
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: 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: 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:

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

We check that b is divisible by a :

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

reduced_charpoly(*var=None*)

Return the reduced characteristic polynomial of this skew polynomial.

INPUT:

- *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:

```
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
```

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 \mathbf{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
```

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

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

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

```
sage: chi[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 characteristic
↪ 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'))
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 \mathbf{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 [`center\(\)`](#) for more details about this). However, the user can specify a different variable name if desired:

```
sage: a.reduced_norm(var='u')
u^3 + 4*u^2 + 4
```

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

```
sage: a.reduced_norm(var=False)
(4, 0, 4, 1)
```

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:

[`reduced_trace\(\)`](#), [`reduced_charpoly\(\)`](#)

reduced_trace(var=None)

Return the reduced trace of this skew polynomial.

INPUT:

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

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: tr = a.reduced_trace(); tr
3*z + 4
```

The reduced trace lies in the center of S , which is a univariate polynomial ring in the variable $z = x^3$ over \mathbf{F}_5 :

```
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:

```
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:

```
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:

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

See also:

[`reduced_norm\(\)`](#), [`reduced_charpoly\(\)`](#)

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

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
```

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

```
sage: Z.<x3> = S.center()
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)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
```

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
sage: a.count_irreducible_divisors()
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()
sage: N = x3^5 + 4*x3^4 + 4*x3^2 + 4*x3 + 3; N
x3^5 + 4*x3^4 + 4*x3^2 + 4*x3 + 3
sage: N.is_irreducible()
True
sage: S(N).count_irreducible_divisors()
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: k.<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
```

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

```
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: 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
```

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

(continues on next page)

(continued from previous page)

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

By convention, the zero skew polynomial has no factorization:

```
sage: S(0).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:

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x',Frob]
sage: a = x^3 + (t^2 + 1)*x^2 + (2*t + 3)*x + t^2 + t + 2
sage: iter = a.factorizations(); iter
<...generator object at 0x...>
sage: next(iter) # random
(x + 3*t^2 + 4*t) * (x + 2*t^2) * (x + 4*t^2 + 4*t + 2)
sage: 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 :

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

We do some checks:

```
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:

```
sage: factorizations2 = [ F for F in a.factorizations() ]
sage: factorizations == factorizations2 # random
False
sage: sorted(factorizations) == sorted(factorizations2)
True
```

is_irreducible()

Return True if this skew polynomial is irreducible.

EXAMPLES:

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

sage: a = x^2 + t*x + 1
sage: a.is_irreducible()
False
sage: a.factor()
(x + 4*t^2 + 4*t + 1) * (x + 3*t + 2)

sage: a = x^2 + t*x + t + 1
sage: a.is_irreducible()
True
sage: a.factor()
x^2 + t*x + t + 1
```

Skew polynomials of degree 1 are of course irreducible:

```
sage: a = x + t
sage: a.is_irreducible()
True
```

A random irreducible skew polynomial is irreducible:

```
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
```

By convention, constant skew polynomials are not irreducible:

```
sage: S(1).is_irreducible()
False
sage: S(0).is_irreducible()
False
```

left_irreducible_divisor(*uniform=False*)

Return a left 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: dl = a.left_irreducible_divisor(); dl # random
x^3 + (t^2 + t + 2)*x^2 + (t + 2)*x + 3*t^2 + t + 4
```

(continues on next page)

(continued from previous page)

```
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:

```
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:

```
sage: a.left_irreducible_divisor(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
```

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(uniform=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:

```
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:

```
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:

```
sage: S(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:

```
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.right_irreducible_divisors(); iter
```

(continues on next page)

(continued from previous page)

```
<...generator object at 0x...>
sage: next(iter) # random
x + 2*t^2 + 4*t + 4
sage: next(iter) # random
x + 3*t^2 + 4*t + 1
```

We can use this function to build the list of all monic irreducible divisors of a :

```
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:

```
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, \dots) defined by

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

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]
```

(continues on next page)

(continued from previous page)

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

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

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

If $a = N$, the type is just $[r]$ where r is the order of the twisting morphism Frob :

```
sage: N = x^3 + x + 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
```

The simplest way to build elements in K is to use the division operator over Ore polynomial rings:

```
sage: f = 1/d
sage: f
d^(-1)
sage: f.parent() is K
True
```

REPRESENTATION 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:

```
sage: g = t / d
sage: g
(d - 1/t)^(-1) * t
```

The left numerator and right denominator are accessible as follows:

```
sage: g.left_numerator()
t
sage: g.right_denominator()
d
```

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

```
sage: g.left_denominator()
d - 1/t
sage: g.right_numerator()
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: sigma = R.hom([t^2])
sage: S.<x> = R['x', sigma]
sage: F = S.fraction_field()
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 Ring endomorphism of Fraction_
↪Field of Univariate Polynomial Ring in t over Rational Field
Defn: t |--> t^2
```

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:

```
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: f
(d^2 + t*d + 1)^(-1) * (d + t^2)
sage: g = (D*P)^(-1) * (D*Q)
sage: g
(d^2 + t*d + 1)^(-1) * (d + t^2)

sage: P = x^2 + t*x + 1
sage: Q = x + t^2
```

(continues on next page)

(continued from previous page)

```

sage: D = x^3 + t^2 + 1
sage: f = P^(-1) * Q
sage: f
(x^2 + t*x + 1)^(-1) * (x + t^2)
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

```

OPERATIONS:

Basic arithmetical operations are available:

```

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)

```

Of course, multiplication remains noncommutative:

```

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(domain, codomain,
ringembed)

```

Bases: [RingHomomorphism](#)

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

section()

Return a section of this morphism.

EXAMPLES:

```

sage: k.<a> = GF(5^3)
sage: S.<x> = SkewPolynomialRing(k, k.frobenius_endomorphism())
sage: K = S.fraction_field()
sage: Z = K.center()

```

(continues on next page)

(continued from previous page)

```

sage: iota = K.coerce_map_from(Z)
sage: sigma = iota.section()
sage: sigma(x^3 / (x^6 + 1))
z/(z^2 + 1)

```

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

Bases: Algebra, 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:

```

sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: R.<x> = OrePolynomialRing(k,Frob)
sage: K = R.fraction_field()
sage: K
Ore Function Field in x over Finite Field in t of size 5^3 twisted by t |--> t^5

sage: Ky = K.change_var('y'); Ky
Ore Function Field in y over Finite Field in t of size 5^3 twisted by t |--> t^5
sage: Ky is K.change_var('y')
True

```

characteristic()

Return the characteristic of this Ore function field.

EXAMPLES:

```

sage: R.<t> = QQ[]
sage: sigma = R.hom([t+1])
sage: S = R['x',sigma]
sage: S.fraction_field().characteristic()
0

sage: k.<u> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S = k['y',Frob]
sage: S.fraction_field().characteristic()
5

```

fraction_field()

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

EXAMPLES:

```

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
sage: K.fraction_field()
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() 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: 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

```

gens_dict()

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

EXAMPLES:

```

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: 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

```

(continues on next page)

(continued from previous page)

```
sage: T.<y> = k['y', Frob^3]
sage: L = T.fraction_field()
sage: 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: 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: 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: 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: 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: 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
```

ngens()

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

EXAMPLES:

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

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:

```
sage: 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
```

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

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
- *monic* – (default: False) if True, return a monic Ore function with monic numerator and denominator
- **args, **kws* – passed in to the `random_element` method for the base ring

EXAMPLES:


```

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) * (x^2 + (2*t^2 + t + 3)*x + 3*t^2
↳ + t + 2)
sage: K.random_element(degree=3) # random
(x^3 + (2*t^2 + 3)*x^2 + (2*t^2 + 4)*x + t + 3)^(-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=[2,5]) # 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:

```

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

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:

```

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

```

(continues on next page)

(continued from previous page)

```

sage: K = S.fraction_field()
sage: K.twisting_morphism()
Ring endomorphism of Fraction Field of Univariate Polynomial Ring in t over
↳ Rational Field
Defn: t |--> t + 1

```

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: F = A.fraction_field()
sage: F.twisting_morphism()

```

See also:

```

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

```

```

class sage.rings.polynomial.ore_function_field.OreFunctionField_with_large_center(ring,
                                                                                      cate-
                                                                                      gory=None)

```

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:

```

sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: S.<x> = k['x', Frob]
sage: K = S.fraction_field()

sage: Z = K.center(); Z
Fraction Field of Univariate Polynomial Ring in z over Finite Field of size 5

```

We can pass in another variable name:

```

sage: K.center(name='y')
Fraction Field of Univariate Polynomial Ring in y over Finite Field of size 5

```

or use the bracket notation:

```
sage: Zy.<y> = K.center(); Zy
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)
True
```

and pushout works:

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

A conversion map in the reverse direction is also set:

```
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
```

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: 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
```

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
```

```
class sage.rings.polynomial.ore_function_field.SectionOreFunctionCenterInjection(embed)
```

Bases: [Section](#)

Section of the canonical injection of the center of a Ore function field into this field

1.8 Fraction field elements of Ore polynomial rings

AUTHOR:

- Xavier Caruso (2020-05)

```
class sage.rings.polynomial.ore_function_element.ConstantOreFunctionSection
```

Bases: [Map](#)

Representation of the canonical homomorphism from the constants of a Ore function field to the base field.

This class is needed by the coercion system.

EXAMPLES:

```
sage: from sage.rings.polynomial.ore_polynomial_element import _
      ↪ ConstantOrePolynomialSection
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()
sage: 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()
```

(continues on next page)

(continued from previous page)

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

One can specify another variable name:

```
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:

```
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 an homomorphism of fields:

```
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:

```
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:

```
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
```

In the example below, a simplification occurs:

```
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:

```
sage: R.<z> = GF(11)[z]
sage: sigma = R.hom([z^2])
sage: S.<x> = R['x', sigma]
sage: s = (x + z)^2
sage: t = (x + z) * (x^2 + z^2)
sage: f = s^(-1) * t
sage: f.left_denominator()
x^2 + (z^2 + z)*x + z^2
```

However, the following always holds true:

```
sage: f == f.left_denominator()^(-1) * f.right_numerator()
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:

```
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.left_numerator()
x^2 + a*x + a^2
```

In the example below, a simplification occurs:

```
sage: u = S.random_element(degree=2)
sage: g = (t*u) / (s*u)
sage: g.left_numerator()
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:

```
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:

```
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\(\)`](#)

right_numerator()

Return t 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 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 `left_denominator()` outputs the numerator of the *same* fraction.

EXAMPLES:

```
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
```

In the example below, a simplification occurs:

```
sage: u = S.random_element(degree=2)
sage: g = (u*s)^(-1) * (u*t)
sage: g.right_numerator()
x^2 + a*x + a^2
```

See also:

`left_denominator()`, `left_numerator()`, `right_denominator()`

```
class sage.rings.polynomial.ore_function_element.OreFunctionBasingInjection(domain,
                                                                              codomain)
```

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: 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.


```
class sage.rings.polynomial.ore_function_element.OreFunction_with_large_center(parent,
                                                                              numerator,
                                                                              denomina-
                                                                              tor=None,
                                                                              sim-
                                                                              plify=True)
```

Bases: *OreFunction*

A special class for elements of Ore function fields whose center has finite index.

reduced_norm(*var=None*)

Return the reduced norm of this Ore function.

INPUT:

- *var* – a string or None (default: None); the name of the central variable

EXAMPLES:

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

sage: a = (x + t) / (x^2 + t^2)
sage: N = a.reduced_norm(); N
(z + 2)/(z^2 + 4)
```

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 $GF(5)$.

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

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

```
sage: K(N)
(x^6 + 4)^(-1) * (x^3 + 2)
```

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:

```
sage: a.reduced_norm(var='u')
(u + 2)/(u^2 + 4)
```

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: 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')
3/(u^2 + 2)
```

NONCOMMUTATIVE MULTIVARIATE POLYNOMIALS

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 \mathbf{Q} , and
- finite fields \mathbf{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 $\mathbf{Q}(a)$ and \mathbf{Z} support
- Simon King (2009-04): improved coercion
- Martin Albrecht (2009-05): added $\mathbf{Z}/n\mathbf{Z}$ support, refactoring
- Martin Albrecht (2009-06): refactored the code to allow better re-use

Todo: extend functionality towards those of libSINGULARs commutative part

EXAMPLES:

We show how to construct various noncommutative polynomial rings:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P.<x,y,z> = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
```

(continues on next page)

(continued from previous page)

```

sage: P
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-
↪relations: {y*x: -x*y}

sage: y*x + 1/2
-x*y + 1/2

sage: A.<x,y,z> = FreeAlgebra(GF(17), 3)
sage: P.<x,y,z> = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
sage: P
Noncommutative Multivariate Polynomial Ring in x, y, z over Finite Field of size 17, nc-
↪relations: {y*x: -x*y}

sage: y*x + 7
-x*y + 7

```

Raw use of this class; *this is not the intended use!*

```

sage: from sage.matrix.constructor import Matrix
sage: c = Matrix(3)
sage: c[0,1] = -2
sage: c[0,2] = 1
sage: c[1,2] = 1

sage: d = Matrix(3)
sage: d[0, 1] = 17
sage: P = QQ['x','y','z']
sage: c = c.change_ring(P)
sage: d = d.change_ring(P)

sage: from sage.rings.polynomial.plural import NCPolynomialRing_plural
sage: R.<x,y,z> = NCPolynomialRing_plural(QQ, c = c, d = d, order=TermOrder('lex',3),
↪category=Algebras(QQ))
sage: R
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-
↪relations: {y*x: -2*x*y + 17}

sage: R.term_order()
Lexicographic term order

sage: a,b,c = R.gens()
sage: f = 57 * a^2*b + 43 * c + 1; f
57*x^2*y + 43*z + 1

```

`sage.rings.polynomial.plural.ExteriorAlgebra(base_ring, names, order='degrevlex')`

Return the exterior algebra on some generators

This is also known as a Grassmann algebra. This is a finite dimensional algebra, where all generators anti-commute.

See [Wikipedia article Exterior algebra](#)

INPUT:

- `base_ring` – the ground ring

- names – a list of variable names

EXAMPLES:

```
sage: from sage.rings.polynomial.plural import ExteriorAlgebra
sage: E = ExteriorAlgebra(QQ, ['x', 'y', 'z']) ; E #random
Quotient of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational_
↪Field, nc-relations: {z*x: -x*z, z*y: -y*z, y*x: -x*y} by the ideal (z^2, y^2, x^
↪2)
sage: sorted(E.cover().domain().relations().items(), key=str)
[(y*x, -x*y), (z*x, -x*z), (z*y, -y*z)]
sage: sorted(E.cover().kernel().gens(), key=str)
[x^2, y^2, z^2]
sage: E.inject_variables()
Defining xbar, ybar, zbar
sage: x,y,z = (xbar,ybar,zbar)
sage: y*x
-x*y
sage: all(v^2==0 for v in E.gens())
True
sage: E.one()
1
```

```
class sage.rings.polynomial.plural.ExteriorAlgebra_plural
```

Bases: [NCPolynomialRing_plural](#)

```
class sage.rings.polynomial.plural.G_AlgFactory
```

Bases: [UniqueFactory](#)

A factory for the creation of g-algebras as unique parents.

```
create_key_and_extra_args(base_ring, c, d, names=None, order=None, category=None, check=None)
```

Create a unique key for g-algebras.

INPUT:

- base_ring - a ring
- c,d - two matrices
- names - a tuple or list of names
- order - (optional) term order
- category - (optional) category
- check - optional bool

```
create_object(version, key, **extra_args)
```

Create a g-algebra to a given unique key.

INPUT:

- key - a 6-tuple, formed by a base ring, a tuple of names, two matrices over a polynomial ring over the base ring with the given variable names, a term order, and a category
- extra_args - a dictionary, whose only relevant key is 'check'.

```
class sage.rings.polynomial.plural.NCPolynomialRing_plural
```

Bases: [Ring](#)

A non-commutative polynomial ring.

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: H._is_category_initialized()
True
sage: H.category()
Category of algebras over Rational Field
sage: TestSuite(H).run()
```

Note that two variables commute if they are not part of the given relations:

```
sage: H.<x,y,z> = A.g_algebra({z*x:x*z+2*x, z*y:y*z-2*y})
sage: x*y == y*x
True
```

free_algebra()

The free algebra of which this is the quotient.

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
sage: B = P.free_algebra()
sage: A == B
True
```

gen(*n=0*)

Returns the *n*-th generator of this noncommutative polynomial ring.

INPUT:

- *n* – an integer ≥ 0

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
sage: P.gen(), P.gen(1)
(x, y)
```

Note that the generators are not cached:

```
sage: P.gen(1) is P.gen(1)
False
```

ideal(**gens*, ***kws*)

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 - 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}
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
```

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
```

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
```

(continues on next page)

(continued from previous page)

```
sage: P.monomial_all_divisors(x^2*z^3)
[x, x^2, z, x*z, x^2*z, z^2, x*z^2, x^2*z^2, z^3, x*z^3, x^2*z^3]
```

ALGORITHM: addwithcarry idea by Toon Segers

monomial_divides(a, b)

Return False if a does not divide b and True otherwise.

Coefficients are ignored.

INPUT:

- a – monomial
- b – monomial

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order='lex')
sage: P.inject_variables()
Defining x, y, z

sage: P.monomial_divides(x*y*z, x^3*y^2*z^4)
True
sage: P.monomial_divides(x^3*y^2*z^4, x*y*z)
False
```

monomial_lcm(f, g)

LCM for monomials. Coefficients are ignored.

INPUT:

- f - monomial
- g - monomial

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order='lex')
sage: P.inject_variables()
Defining x, y, z

sage: P.monomial_lcm(3/2*x*y, x)
x*y
```

monomial_pairwise_prime(g, h)

Return True if h and g are pairwise prime.

Both h and g are treated as monomials.

Coefficients are ignored.

INPUT:

- h - monomial
- g - monomial

EXAMPLES:

```
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*, *coeff*=False)

Return f/g , where both f and g are treated as monomials.

Coefficients are ignored by default.

INPUT:

- f - monomial
- g - monomial
- *coeff* - divide coefficients as well (default: False)

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order='lex')
sage: P.inject_variables()
Defining x, y, z

sage: P.monomial_quotient(3/2*x*y, x, coeff=True)
3/2*y
```

Note that \mathbb{Z} behaves differently if *coeff*=True:

```
sage: P.monomial_quotient(2*x, 3*x)
1
sage: P.monomial_quotient(2*x, 3*x, coeff=True)
2/3
```

Warning: Assumes that the head term of f is a multiple of the head term of g and return the multiplicant m . If this rule is violated, funny things may happen.

monomial_reduce(*f*, *G*)

Try to find a g in G where $g.lm()$ divides f . If found (flt, g) is returned, $(0, 0)$ otherwise, where flt is $f/g.lm()$.

It is assumed that G is iterable and contains *only* elements in this polynomial ring.

Coefficients are ignored.

INPUT:

- f - monomial

- G - list/set of mpolynomials

EXAMPLES:

```
sage: 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
sage: G = [ 3/2*x^3 + y^2 + 1/2, 1/4*x*y + 2/7, 1/2 ]
sage: P.monomial_reduce(f,G)
(y, 1/4*x*y + 2/7)
```

ngens()

Returns the number of variables in this noncommutative polynomial ring.

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P.<x,y,z> = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
sage: P.ngens()
3
```

relations(add_commutative=False)

Return the relations of this g-algebra.

INPUT:

add_commutative (optional bool, default False)

OUTPUT:

The defining relations. There are some implicit relations: Two generators commute if they are not part of any given relation. The implicit relations are not provided, unless add_commutative==True.

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({z*x:x*z+2*x, z*y:y*z-2*y})
sage: x*y == y*x
True
sage: H.relations()
{z*x: x*z + 2*x, z*y: y*z - 2*y}
sage: H.relations(add_commutative=True)
{y*x: x*y, z*x: x*z + 2*x, z*y: y*z - 2*y}
```

term_order()

Return the term ordering of the noncommutative ring.

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
sage: P.term_order()
Lexicographic term order
```

(continues on next page)

(continued from previous page)

```
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
```

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')
sage: R.inject_variables()
Defining x, z, y
sage: f=x*y+5
sage: c=f.coefficient({x:0,y:0}); c
5
sage: parent(c)
Noncommutative Multivariate Polynomial Ring in x, z, y over Finite Field of
size 389, nc-relations: {y*x: -x*y + z}

```

AUTHOR:

- Joel B. Mohler (2007-10-31)

constant_coefficient()

Return the constant coefficient of this multivariate polynomial.

EXAMPLES:

```

sage: A.<x,z,y> = FreeAlgebra(GF(389), 3)
sage: P = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: P.inject_variables()
Defining x, z, y
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.constant_coefficient()
5
sage: f = 3*x^2
sage: f.constant_coefficient()
0

```

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:

```

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

```

(continues on next page)

(continued from previous page)

```

sage: f.degree(y)
2
sage: (y^10*x - 7*x^2*y^5 + 5*x^3).degree(x)
3
sage: (y^10*x - 7*x^2*y^5 + 5*x^3).degree(y)
10

```

degrees()

Returns a tuple with the maximal degree of each variable in this polynomial. The list of degrees is ordered by the order of the generators.

EXAMPLES:

```

sage: 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)

```

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}

```

exponents(as_ETuples=True)

Return the exponents of the monomials appearing in this polynomial.

INPUT:

- `as_ETuples` - (default: `True`) if `True` returns the result as an list of ETuples otherwise returns a list of tuples

EXAMPLES:

```

sage: A.<x,z,y> = FreeAlgebra(GF(389), 3)
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

```

(continues on next page)

(continued from previous page)

```
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)]
```

is_constant()

Return True if this polynomial is constant.

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_constant()
False
sage: P(1).is_constant()
True
```

is_homogeneous()

Return True if this polynomial is homogeneous.

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+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
```

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
```

(continues on next page)

(continued from previous page)

```

sage: (2*x).is_monomial()
False
sage: (x*y).is_monomial()
True
sage: (x*y + x).is_monomial()
False

```

is_zero()

Return True if this polynomial is zero.

EXAMPLES:

```

sage: A.<x,z,y> = FreeAlgebra(QQ, 3)
sage: R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: R.inject_variables()
Defining x, z, y

sage: x.is_zero()
False
sage: (x-x).is_zero()
True

```

lc()

Leading coefficient of this polynomial with respect to the term order of `self.parent()`.

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.lc()
3

sage: f = 5*x^3*y^2*z^4 + 4*x^3*y^2*z^1
sage: f.lc()
5

```

lm()

Returns the lead monomial of `self` with respect to the term order of `self.parent()`.

In Sage a monomial is a product of variables in some power without a coefficient.

EXAMPLES:

```

sage: A.<x,y,z> = FreeAlgebra(GF(7), 3)
sage: R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: R.inject_variables()
Defining x, y, z
sage: f = x^1*y^2 + y^3*z^4
sage: f.lm()
x*y^2

```

(continues on next page)

(continued from previous page)

```

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

```

lt()

Leading term of this polynomial.

In Sage a term is a product of variables in some power and a coefficient.

EXAMPLES:

```

sage: A.<x,y,z> = FreeAlgebra(GF(7), 3)
sage: R = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: R.inject_variables()
Defining x, y, z

sage: f = 3*x^1*y^2 + 2*y^3*z^4
sage: f.lt()
3*x*y^2

sage: f = 5*x^3*y^2*z^4 + 4*x^3*y^2*z^1
sage: f.lt()
-2*x^3*y^2*z^4

```

monomial_coefficient(mon)

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

This function contrasts with the function `coefficient()` which returns the coefficient of a monomial viewing this polynomial in a polynomial ring over a base ring having fewer variables.

INPUT:

- `mon` - a monomial

OUTPUT:

coefficient in base ring

See also:

For coefficients in a base ring of fewer variables, look at [`coefficient\(\)`](#)

EXAMPLES:

```
sage: A.<x,z,y> = FreeAlgebra(GF(389), 3)
sage: P = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: P.inject_variables()
Defining x, z, y

The parent of the return is a member of the base ring.
sage: f = 2 * x * y
sage: c = f.monomial_coefficient(x*y); c
2
sage: c.parent()
Finite Field of size 389

sage: f = y^2 + y^2*x - x^9 - 7*x + 5*x*y
sage: f.monomial_coefficient(y^2)
1
sage: f.monomial_coefficient(x*y)
5
sage: f.monomial_coefficient(x^9)
388
sage: f.monomial_coefficient(x^10)
0
```

monomials()

Return the list of monomials in `self`

The returned list is decreasingly ordered by the term ordering of `self.parent()`.

EXAMPLES:

```
sage: A.<x,z,y> = FreeAlgebra(GF(389), 3)
sage: P = A.g_algebra(relations={y*x:-x*y + z}, order='lex')
sage: P.inject_variables()
Defining x, z, y
sage: f = x + (3^2)*y*z^2 + (2+3)
sage: f.monomials()
[x, z^2*y, 1]
sage: f = P(3^2)
sage: f.monomials()
[1]
```

reduce()

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False)
```

The result of reduction is not the normal form, if one reduces by a list of polynomials:

```
sage: (x*z).reduce(I.gens())
x*z
```

However, if the argument is an ideal, then a normal form (reduction with respect to a two-sided Groebner basis) is returned:

```
sage: (x*z).reduce(I)
-x
```

The Groebner basis shows that the result is correct:

```
sage: I.std() #random
Left Ideal (z^2 - 1, y*z - y, x*z + x, y^2, 2*x*y - z - 1, x^2) of
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational
Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z}
sage: sorted(I.std().gens(),key=str)
[z*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
sage: f.total_degree()
10
sage: f=z^9+10*x^4+y^8*x^2
sage: f.total_degree()
10
```

`sage.rings.polynomial.plural.SCA(base_ring, names, alt_vars, order='degrevlex')`

Return a free graded-commutative algebra

This is also known as a free super-commutative algebra.

INPUT:

- `base_ring` – the ground field
- `names` – a list of variable names
- `alt_vars` – a list of indices of to be anti-commutative variables (odd variables)
- `order` – ordering to be used for the constructed algebra

EXAMPLES:

```
sage: from sage.rings.polynomial.plural import SCA
sage: E = SCA(QQ, ['x', 'y', 'z'], [0, 1], order = 'degrevlex')
sage: 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)
sage: E.inject_variables()
Defining xbar, ybar, zbar
sage: x,y,z = (xbar,ybar,zbar)
sage: y*x
-x*y
sage: z*x
x*z
sage: x^2
0
sage: y^2
0
sage: z^2
z^2
sage: E.one()
1
```

`sage.rings.polynomial.plural.new_CRing(rw, base_ring)`

Construct MPolynomialRing_libsingular from ringWrap, assuming the ground field to be `base_ring`

EXAMPLES:

```
sage: H.<x,y,z> = PolynomialRing(QQ, 3)
sage: from sage.libs.singular.function import singular_function

sage: ringlist = singular_function('ringlist')
sage: ring = singular_function("ring")

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())
```

(continues on next page)

(continued from previous page)

```
sage: R # indirect doctest
Multivariate Polynomial Ring in x, y, z over Rational Field
```

Check that [github issue #13145](#) has been resolved:

```
sage: h = hash(R.gen() + 1) # sets currRing
sage: from sage.libs.singular.ring import ring_refcount_dict, currRing_wrapper
sage: curcnt = ring_refcount_dict[currRing_wrapper()]
sage: newR = new_CRing(W, H.base_ring())
sage: ring_refcount_dict[currRing_wrapper()] - curcnt
2
```

Check that [github issue #29311](#) is fixed:

```
sage: R.<x,y,z> = QQ[]
sage: from sage.libs.singular.function_factory import ff
sage: W = ff.ring(ff.ringlist(R), ring=R)
sage: C = sage.rings.polynomial.plural.new_CRing(W, R.base_ring())
sage: C.one()
1
```

`sage.rings.polynomial.plural.new_NRing(rw, base_ring)`

Construct NCPolynomialRing_plural from ringWrap, assuming the ground field to be base_ring

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-1})
sage: H.inject_variables()
Defining x, y, z
sage: z*x
x*z
sage: z*y
y*z
sage: y*x
x*y - 1
sage: I = H.ideal([y^2, x^2, z^2-1])
sage: I._groebner_basis_libsingular()
[1]

sage: from sage.libs.singular.function import singular_function

sage: ringlist = singular_function('ringlist')
sage: ring = singular_function("ring")

sage: L = ringlist(H, ring=H); L
[
[0 1 1]
[0 0 1]
0, ['x', 'y', 'z'], [['dp', (1, 1, 1)], ['C', (0,)], [0], [0 0 0],
[ 0 -1 0]
[ 0 0 0]
```

(continues on next page)

(continued from previous page)

```

[ 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}

```

`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: 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

```

(continues on next page)

(continued from previous page)

```
sage: W = ring(L, ring=H); W
<noncommutative RingWrap>

sage: from sage.rings.polynomial.plural import new_Ring
sage: R = new_Ring(W, H.base_ring()); R
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-
↪relations: {y*x: x*y - 1}
```

`sage.rings.polynomial.plural.unpickle_NCPolynomial_plural(R, d)`

Auxiliary function to unpickle a non-commutative polynomial.

INDICES AND TABLES

- [Index](#)
- [Module Index](#)
- [Search Page](#)

PYTHON MODULE INDEX

r

`sage.rings.polynomial.ore_function_element`,
72
`sage.rings.polynomial.ore_function_field`, 62
`sage.rings.polynomial.ore_polynomial_element`,
11
`sage.rings.polynomial.ore_polynomial_ring`, 1
`sage.rings.polynomial.plural`, 79
`sage.rings.polynomial.skew_polynomial_element`,
46
`sage.rings.polynomial.skew_polynomial_finite_field`,
55
`sage.rings.polynomial.skew_polynomial_finite_order`,
49
`sage.rings.polynomial.skew_polynomial_ring`,
40

INDEX

A

`an_element()` (*sage.rings.polynomial.ore_function_element.OreFunctionElement*),
method), 76

`an_element()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*),
method), 37

B

`base_ring()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*),
method), 15

`bound()` (*sage.rings.polynomial.skew_polynomial_finite_order.SkewPolynomialFiniteOrder*),
method), 50

C

`center()` (*sage.rings.polynomial.ore_function_field.OreFunctionField*),
method), 70

`center()` (*sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing*),
method), 44

`change_var()` (*sage.rings.polynomial.ore_function_field.OreFunctionField*),
method), 65

`change_var()` (*sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing*),
method), 4

`change_variable_name()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*),
method), 15

`characteristic()` (*sage.rings.polynomial.ore_function_field.OreFunctionField*),
method), 65

`characteristic()` (*sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing*),
method), 4

`coefficient()` (*sage.rings.polynomial.plural.NCPolynomial_plural*),
method), 87

`coefficients()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*),
method), 15

`coefficients()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*),
method), 37

`conjugate()` (*sage.rings.polynomial.skew_polynomial_element.SkewPolynomialElement*),
method), 46

`constant_coefficient()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*),
method), 16

`constant_coefficient()` (*sage.rings.polynomial.plural.NCPolynomial_plural*),
method), 88

`ConstantOreFunctionSection` (class in *sage.rings.polynomial.ore_function_element*),
72

`ConstantOrePolynomialSection` (class in *sage.rings.polynomial.ore_polynomial_element*),
11

`count_factorizations()` (*sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomialFiniteField*),
method), 55

`count_irreducible_divisors()` (*sage.rings.polynomial.skew_polynomial_finite_order_dense.SkewPolynomialFiniteOrderDense*),
method), 55

`create_key_and_extra_args()` (*sage.rings.polynomial.plural.G_AlgebraFactory*),
method), 81

`create_object()` (*sage.rings.polynomial.plural.G_AlgebraFactory*),
method), 81

`degree()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*),
method), 16

`degree()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*),
method), 38

`degree()` (*sage.rings.polynomial.plural.NCPolynomial_plural*),
method), 88

`degrees()` (*sage.rings.polynomial.plural.NCPolynomial_plural*),
method), 89

`dict()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*),
method), 38

`dict()` (*sage.rings.polynomial.plural.NCPolynomial_plural*),
method), 89

E

`Element` (*sage.rings.polynomial.ore_function_field.OreFunctionField*),
attribute), 65

`Element` (*sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing*),
attribute), 4

`exponents()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*),
method), 16

`exponents()` (*sage.rings.polynomial.plural.NCPolynomial_plural*),
method), 89

```

ExteriorAlgebra() (in sage.rings.polynomial.plural), 80
ExteriorAlgebra_plural (class in sage.rings.polynomial.plural), 81
F
factor() (sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomialFiniteField), 56
factorizations() (sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomialFiniteField), 57
fraction_field() (sage.rings.polynomial.ore_function_field.OreFunctionField), 65
fraction_field() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing), 5
free_algebra() (sage.rings.polynomial.plural.NCPolynomialRing_plural), 82
G
G_AlgebraFactory (class in sage.rings.polynomial.plural), 81
gen() (sage.rings.polynomial.ore_function_field.OreFunctionField), 66
gen() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing), 5
gen() (sage.rings.polynomial.plural.NCPolynomialRing_plural), 82
gens_dict() (sage.rings.polynomial.ore_function_field.OreFunctionField), 66
gens_dict() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing), 6
H
hamming_weight() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement), 16
hilbert_shift() (sage.rings.polynomial.ore_function_element.OreFunctionElement), 72
hilbert_shift() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement), 38
I
ideal() (sage.rings.polynomial.plural.NCPolynomialRing_plural), 82
is_central() (sage.rings.polynomial.skew_polynomial_finite_order.SkewPolynomialFiniteOrderDense), 51
is_commutative() (sage.rings.polynomial.ore_function_field.OreFunctionField), 66
is_commutative() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing), 6
is_commutative() (sage.rings.polynomial.plural.NCPolynomialRing_plural), 83
is_constant() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement), 17
is_constant() (sage.rings.polynomial.plural.NCPolynomialRing_plural), 90
is_exact() (sage.rings.polynomial.ore_function_field.OreFunctionField), 67
is_exact() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing), 7
is_field() (sage.rings.polynomial.ore_function_field.OreFunctionField), 67
is_finite() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing), 7
is_finite() (sage.rings.polynomial.plural.NCPolynomialRing_plural), 83
is_finite() (sage.rings.polynomial.ore_function_field.OreFunctionField), 67
is_finite() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing), 7
is_homogeneous() (sage.rings.polynomial.plural.NCPolynomialRing_plural), 90
is_irreducible() (sage.rings.polynomial.skew_polynomial_finite_order.SkewPolynomialFiniteOrderDense), 57
is_left_divisible_by() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement), 17
is_monic() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement), 18
is_monomial() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement), 18
is_monomial() (sage.rings.polynomial.plural.NCPolynomialRing_plural), 90
is_nilpotent() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement), 19
is_one() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement), 19
is_right_divisible_by() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement), 19
is_sparse() (sage.rings.polynomial.ore_function_field.OreFunctionField), 67
is_sparse() (sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing), 7
is_term() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement), 20
is_unit() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement), 20
is_zero() (sage.rings.polynomial.ore_function_element.OreFunctionElement), 73
is_zero() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement), 21
is_zero() (sage.rings.polynomial.plural.NCPolynomialRing_plural), 91
lagrange_polynomial() (sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing), 42

```

`lc()` (*sage.rings.polynomial.plural.NCPolynomial_plural*
method), 91
`leading_coefficient()`
(*sage.rings.polynomial.ore_polynomial_element.OrePolynomial*
method), 21
`left_denominator()` (*sage.rings.polynomial.ore_function_element.OreFunction*
method), 73
`left_divides()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomial*
method), 21
`left_gcd()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomial*
method), 22
`left_irreducible_divisor()`
(*sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field_dense*
method), 58
`left_irreducible_divisors()`
(*sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field_dense*
method), 59
`left_lcm()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomial*
method), 23
`left_mod()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomial*
method), 24
`left_monic()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomial*
method), 24
`left_numerator()` (*sage.rings.polynomial.ore_function_element.OreFunction*
method), 74
`left_power_mod()` (*sage.rings.polynomial.skew_polynomial_element.SkewPolynomial_generic_dense*
method), 47
`left_quo_rem()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomial*
method), 25
`left_xgcd()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomial*
method), 26
`left_xlcm()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomial*
method), 27
`list()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomial_generic_dense*
method), 39
`lm()` (*sage.rings.polynomial.plural.NCPolynomial_plural*
method), 91
`lt()` (*sage.rings.polynomial.plural.NCPolynomial_plural*
method), 92
M
`minimal_vanishing_polynomial()`
(*sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialRing*
method), 42
module
`sage.rings.polynomial.ore_function_element`
72
`sage.rings.polynomial.ore_function_field`
62
`sage.rings.polynomial.ore_polynomial_element`
11
`sage.rings.polynomial.ore_polynomial_ring`
1
`sage.rings.polynomial.plural`, 79
`sage.rings.polynomial.skew_polynomial_element`,
46
`sage.rings.polynomial.skew_polynomial_finite_field`,
55
`sage.rings.polynomial.skew_polynomial_finite_order`,
54
`sage.rings.polynomial.skew_polynomial_ring`,
40
`monomial_all_divisors()`
(*sage.rings.polynomial.plural.NCPolynomialRing_plural*
method), 83
`monomial_coefficient()`
(*sage.rings.polynomial.plural.NCPolynomialRing_plural*
method), 92
`monomial_divides()` (*sage.rings.polynomial.plural.NCPolynomialRing_plural*
method), 84
`monomial_lcm()` (*sage.rings.polynomial.plural.NCPolynomialRing_plural*
method), 84
`monomial_pairwise_prime()`
(*sage.rings.polynomial.plural.NCPolynomialRing_plural*
method), 85
`monomial_reduce()` (*sage.rings.polynomial.plural.NCPolynomialRing_plural*
method), 85
`monomials()` (*sage.rings.polynomial.plural.NCPolynomial_plural*
method), 62
`multi_point_evaluation()`
(*sage.rings.polynomial.skew_polynomial_element.SkewPolynomial*
method), 47
N
`NCPolynomialRing_plural` (class in
sage.rings.polynomial.plural), 87
`NCPolynomialRing_plural` (class in
sage.rings.polynomial.plural), 81
`new_CRing()` (in module *sage.rings.polynomial.plural*),
95
`new_NRing()` (in module *sage.rings.polynomial.plural*),
96
`new_Ring()` (in module *sage.rings.polynomial.plural*),
97
`ngens()` (*sage.rings.polynomial.ore_function_field.OreFunctionField*
method), 68
`ngens()` (*sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing*
method), 8
`ngens()` (*sage.rings.polynomial.plural.NCPolynomialRing_plural*
method), 86
`number_of_terms()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomial*
method), 28
O
`operator_eval()` (*sage.rings.polynomial.skew_polynomial_element.SkewPolynomial*
method), 47

`method`), 48
`optimal_bound()` (*sage.rings.polynomial.skew_polynomial_finite_order_dense*
`method`), 51
`OreFunction` (class in *sage.rings.polynomial.ore_function_element*),
72
`OreFunction_with_large_center` (class in *sage.rings.polynomial.ore_function_element*),
76
`OreFunctionBasingInjection` (class in *sage.rings.polynomial.ore_function_element*),
76
`OreFunctionCenterInjection` (class in *sage.rings.polynomial.ore_function_field*),
64
`OreFunctionField` (class in *sage.rings.polynomial.ore_function_field*),
65
`OreFunctionField_with_large_center` (class in *sage.rings.polynomial.ore_function_field*), 70
`OrePolynomial` (class in *sage.rings.polynomial.ore_polynomial_element*),
12
`OrePolynomial_generic_dense` (class in *sage.rings.polynomial.ore_polynomial_element*),
37
`OrePolynomialBasingInjection` (class in *sage.rings.polynomial.ore_polynomial_element*),
36
`OrePolynomialRing` (class in *sage.rings.polynomial.ore_polynomial_ring*), 1

P

`padded_list()` (*sage.rings.polynomial.ore_polynomial_element*), *OrePolynomial*
`method`), 28
`parameter()` (*sage.rings.polynomial.ore_function_field.OreFunctionField*
`method`), 68
`parameter()` (*sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing*
`method`), 8
`prec()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomial*
`method`), 29

R

`random_element()` (*sage.rings.polynomial.ore_function_field.OreFunctionField*
`method`), 68
`random_element()` (*sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing*
`method`), 8
`random_irreducible()`
(*sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing*
`method`), 9
`reduce()` (*sage.rings.polynomial.plural.NCPolynomial_plural*
`method`), 93
`reduced_charpoly()` (*sage.rings.polynomial.skew_polynomial_finite_order_dense*
`method`), 52
`reduced_norm()` (*sage.rings.polynomial.ore_function_element.OreFunctionElement*
`method`), 53
`reduced_norm()` (*sage.rings.polynomial.skew_polynomial_finite_order_skew_polynomial_finite_order_dense*
`method`), 53
`reduced_trace()` (*sage.rings.polynomial.ore_function_element.OreFunctionElement*
`method`), 77
`reduced_trace()` (*sage.rings.polynomial.skew_polynomial_finite_order_skew_polynomial_finite_order_dense*
`method`), 54
`relations()` (*sage.rings.polynomial.plural.NCPolynomialRing_plural*
`method`), 86
`right_denominator()`
(*sage.rings.polynomial.ore_function_element.OreFunctionElement*
`method`), 75
`right_divides()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*
`method`), 29
`right_gcd()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*
`method`), 30
`right_irreducible_divisor()`
(*sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field*
`method`), 59
`right_irreducible_divisors()`
(*sage.rings.polynomial.skew_polynomial_finite_field.SkewPolynomial_finite_field*
`method`), 60
`right_lcm()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*
`method`), 31
`right_mod()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*
`method`), 32
`right_monic()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*
`method`), 32
`right_numerator()` (*sage.rings.polynomial.ore_function_element.OreFunctionElement*
`method`), 75
`right_power_mod()` (*sage.rings.polynomial.skew_polynomial_element.SkewPolynomial_finite_order_dense*
`method`), 48
`right_rem()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*
`method`), 33
`right_xgcd()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*
`method`), 34
`right_xlcm()` (*sage.rings.polynomial.ore_polynomial_element.OrePolynomialElement*
`method`), 35

S

sage.rings.polynomial.ore_function_element
module, 72
sage.rings.polynomial.ore_function_field
module, 62
sage.rings.polynomial.ore_polynomial_element
module, 11
sage.rings.polynomial.ore_polynomial_ring
module, 1
sage.rings.polynomial.plural
module, 79
sage.rings.polynomial.skew_polynomial_element
module, 46
sage.rings.polynomial.skew_polynomial_finite_order_dense
module, 52
sage.rings.polynomial.skew_polynomial_finite_field
module, 59

module, 55
 sage.rings.polynomial.skew_polynomial_finite_order
 module, 49
 sage.rings.polynomial.skew_polynomial_ring
 module, 40
 SCA() (in module sage.rings.polynomial.plural), 94
 section() (sage.rings.polynomial.ore_function_element.OreFunctionCenterInjection
 method), 76
 section() (sage.rings.polynomial.ore_function_field.OreFunctionCenterInjection
 method), 64
 section() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialRing
 method), 37
 section() (sage.rings.polynomial.skew_polynomial_ring.SkewPolynomialCenterInjection
 method), 41
 SectionOreFunctionCenterInjection (class in
 sage.rings.polynomial.ore_function_field), 71
 SectionSkewPolynomialCenterInjection (class in
 sage.rings.polynomial.skew_polynomial_ring),
 41
 shift() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialRing
 method), 35
 SkewPolynomial_finite_field_dense (class in
 sage.rings.polynomial.skew_polynomial_finite_field),
 55
 SkewPolynomial_finite_order_dense (class in
 sage.rings.polynomial.skew_polynomial_finite_order),
 49
 SkewPolynomial_generic_dense (class in
 sage.rings.polynomial.skew_polynomial_element),
 46
 SkewPolynomialCenterInjection (class in
 sage.rings.polynomial.skew_polynomial_ring),
 41
 SkewPolynomialRing (class in
 sage.rings.polynomial.skew_polynomial_ring),
 41
 SkewPolynomialRing_finite_field (class in
 sage.rings.polynomial.skew_polynomial_ring),
 43
 SkewPolynomialRing_finite_order (class in
 sage.rings.polynomial.skew_polynomial_ring),
 43
 square() (sage.rings.polynomial.ore_polynomial_element.OrePolynomialRing
 method), 36

T
 term_order() (sage.rings.polynomial.plural.NCPolynomialRing_plural
 method), 86
 total_degree() (sage.rings.polynomial.plural.NCPolynomialRing_plural
 method), 94
 truncate() (sage.rings.polynomial.ore_polynomial_element.OrePolynomial_generic_dense
 method), 40
 twisting_derivation()
 (sage.rings.polynomial.ore_function_field.OreFunctionField