Drinfeld modules

Release 10.1

The Sage Development Team

Aug 21, 2023

CONTENTS

SageMath include facilities to manipulate Drinfeld modules and their morphisms. The main entry point is the class *sage.rings.function_field.drinfeld_modules.drinfeld_module.DrinfeldModule*.

CHAPTER

ONE

DRINFELD MODULES

1.1 Drinfeld modules

This module provides the class `sage.rings.function_field.drinfeld_module.drinfeld_module.DrinfeldModule`.

For finite Drinfeld modules and their theory of complex multiplication, see class `sage.rings.function_field.drinfeld_module.finite_drinfeld_module.DrinfeldModule`.

AUTHORS:

- Antoine Leudière (2022-04): initial version
- Xavier Caruso (2022-06): initial version
- David Ayotte (2023-03): added basic j-invariants

class `sage.rings.function_field.drinfeld_modules.drinfeld_module.`**`DrinfeldModule`**(*gen*, *category*)

Bases: `Parent`, `UniqueRepresentation`

This class implements Drinfeld $\mathbb{F}_q[T]$-modules.

Let $\mathbb{F}_q[T]$ be a polynomial ring with coefficients in a finite field $\mathbb{F}_q$ and let K be a field. Fix a ring morphism $\gamma : \mathbb{F}_q[T] \to K$; we say that K is an $\mathbb{F}_q[T]$-*field*. Let $K\{\tau\}$ be the ring of Ore polynomials with coefficients in K, whose multiplication is given by the rule $\tau\lambda = \lambda^q\tau$ for any $\lambda \in K$.

A Drinfeld $\mathbb{F}_q[T]$-module over the base $\mathbb{F}_q[T]$-field K is an $\mathbb{F}_q$-algebra morphism $\phi : \mathbb{F}_q[T] \to K\{\tau\}$ such that $\mathrm{Im}(\phi) \not\subset K$ and ϕ agrees with γ on $\mathbb{F}_q$.

For a in $\mathbb{F}_q[T]$, $\phi(a)$ is denoted ϕ_a.

The Drinfeld $\mathbb{F}_q[T]$-module ϕ is uniquely determined by the image ϕ_T of T; this serves as input of the class.

Note: See also *`sage.categories.drinfeld_modules`*.

The *base morphism* is the morphism $\gamma : \mathbb{F}_q[T] \to K$. The monic polynomial that generates the kernel of γ is called the $\mathbb{F}_q[T]$-*characteristic*, or *function-field characteristic*, of the base field. We say that $\mathbb{F}_q[T]$ is the *function ring* of ϕ; $K\{\tau\}$ is the *Ore polynomial ring*. Further, the *generator* is ϕ_T and the *constant coefficient* is the constant coefficient of ϕ_T.

A Drinfeld module is said to be *finite* if the field K is. Despite an emphasis on this case, the base field can be any extension of $\mathbb{F}_q$:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(6)
sage: phi = DrinfeldModule(A, [z, 4, 1])
sage: phi
Drinfeld module defined by T |--> t^2 + 4*t + z
```

```
sage: Fq = GF(49)
sage: A.<T> = Fq[]
sage: K = Frac(A)
sage: psi = DrinfeldModule(A, [K(T), T+1])
sage: psi
Drinfeld module defined by T |--> (T + 1)*t + T
```

Note: Finite Drinfeld modules are implemented in the class *sage.rings.function_field.drinfeld_modules.finite_drinfeld_module*.

Classical references on Drinfeld modules include [Gos1998], [Rosen2002], [VS06] and [Gek1991].

Note: Drinfeld modules are defined in a larger setting, in which the polynomial ring $\mathbb{F}_q[T]$ is replaced by a more general function ring: the ring of functions in k that are regular outside ∞, where k is a function field over $\mathbb{F}_q$ with transcendence degree 1 and ∞ is a fixed place of k. This is out of the scope of this implementation.

INPUT:

- `function_ring` – a univariate polynomial ring whose base field is a finite field
- `gen` – the generator of the Drinfeld module; as a list of coefficients or an Ore polynomial
- `name` (default: `'t'`) – the name of the Ore polynomial ring generator

Construction

A Drinfeld module object is constructed by giving the function ring and the generator:

```
sage: Fq.<z2> = GF(3^2)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(6)
sage: phi = DrinfeldModule(A, [z, 1, 1])
sage: phi
Drinfeld module defined by T |--> t^2 + t + z
```

Note: Note that the definition of the base field is implicit; it is automatically defined as the compositum of all the parents of the coefficients.

The above Drinfeld module is finite; it can also be infinite:

```
sage: L = Frac(A)
sage: psi = DrinfeldModule(A, [L(T), 1, T^3 + T + 1])
```

(continues on next page)

(continued from previous page)

```
sage: psi
Drinfeld module defined by T |--> (T^3 + T + 1)*t^2 + t + T
```

```
sage: phi.is_finite()
True
sage: psi.is_finite()
False
```

In those examples, we used a list of coefficients (`[z, 1, 1]`) to represent the generator $\phi_T = z + t + t^2$. One can also use regular Ore polynomials:

```
sage: ore_polring = phi.ore_polring()
sage: t = ore_polring.gen()
sage: rho_T = z + t^3
sage: rho = DrinfeldModule(A, rho_T)
sage: rho
Drinfeld module defined by T |--> t^3 + z
sage: rho(T) == rho_T
True
```

Images under the Drinfeld module are computed by calling the object:

```
sage: phi(T)  # phi_T, the generator of the Drinfeld module
t^2 + t + z
sage: phi(T^3 + T + 1)  # phi_(T^3 + T + 1)
t^6 + (z^11 + z^9 + 2*z^6 + 2*z^4 + 2*z + 1)*t^4
+ (2*z^11 + 2*z^10 + z^9 + z^8 + 2*z^7 + 2*z^6 + z^5 + 2*z^3)*t^3
+ (2*z^11 + z^10 + z^9 + 2*z^7 + 2*z^6 + z^5 + z^4 + 2*z^3 + 2*z + 2)*t^2
+ (2*z^11 + 2*z^8 + 2*z^6 + z^5 + z^4 + 2*z^2)*t + z^3 + z + 1
sage: phi(1)  # phi_1
1
```

The category of Drinfeld modules

Drinfeld modules have their own category (see class *sage.categories.drinfeld_modules.DrinfeldModules*):

```
sage: phi.category()
Category of Drinfeld modules over Finite Field in z of size 3^12 over its base
sage: phi.category() is psi.category()
False
sage: phi.category() is rho.category()
True
```

One can use the category to directly create new objects:

```
sage: cat = phi.category()
sage: cat.object([z, 0, 0, 1])
Drinfeld module defined by T |--> t^3 + z
```

The base field of a Drinfeld module

The base field of the Drinfeld module is retrieved using `base()`:

```
sage: phi.base()
Finite Field in z of size 3^12 over its base
```

The base morphism is retrieved using `base_morphism()`:

```
sage: phi.base_morphism()
Ring morphism:
  From: Univariate Polynomial Ring in T over Finite Field in z2 of size 3^2
  To:   Finite Field in z of size 3^12 over its base
  Defn: T |--> z
```

Note that the base field is *not* the field K. Rather, it is a ring extension (see `sage.rings.ring_extension.RingExtension`) whose underlying ring is K and whose base is the base morphism:

```
sage: phi.base() is K
False
```

Getters

One can retrieve basic properties:

```
sage: phi.base_morphism()
Ring morphism:
  From: Univariate Polynomial Ring in T over Finite Field in z2 of size 3^2
  To:   Finite Field in z of size 3^12 over its base
  Defn: T |--> z
```

```
sage: phi.ore_polring()  # K{t}
Ore Polynomial Ring in t over Finite Field in z of size 3^12 over its base
 twisted by Frob^2
```

```
sage: phi.function_ring()  # Fq[T]
Univariate Polynomial Ring in T over Finite Field in z2 of size 3^2
```

```
sage: phi.gen()  # phi_T
t^2 + t + z
sage: phi.gen() == phi(T)
True
```

```
sage: phi.constant_coefficient()  # Constant coefficient of phi_T
z
```

```
sage: phi.morphism()  # The Drinfeld module as a morphism
Ring morphism:
  From: Univariate Polynomial Ring in T over Finite Field in z2 of size 3^2
  To:   Ore Polynomial Ring in t
        over Finite Field in z of size 3^12 over its base
```

(continues on next page)

(continued from previous page)

```
        twisted by Frob^2
  Defn: T |--> t^2 + t + z
```

One can compute the rank and height:

```
sage: phi.rank()
2
sage: phi.height()
1
```

As well as the j-invariant:

```
sage: phi.j_invariant()  # j-invariant
1
```

A Drinfeld $\mathbb{F}_q[T]$-module can be seen as an Ore polynomial with positive degree and constant coefficient $\gamma(T)$, where γ is the base morphism. This analogy is the motivation for the following methods:

```
sage: phi.coefficients()
[z, 1, 1]
```

```
sage: phi.coefficient(1)
1
```

Morphisms and isogenies

A *morphism* of Drinfeld modules $\phi \to \psi$ is an Ore polynomial $f \in K\{\tau\}$ such that $f\phi_a = \psi_a f$ for every a in the function ring. In our case, this is equivalent to $f\phi_T = \psi_T f$. An *isogeny* is a nonzero morphism.

Use the `in` syntax to test if an Ore polynomial defines a morphism:

```
sage: phi(T) in Hom(phi, phi)
True
sage: t^6 in Hom(phi, phi)
True
sage: t^5 + 2*t^3 + 1 in Hom(phi, phi)
False
sage: 1 in Hom(phi, rho)
False
sage: 1 in Hom(phi, phi)
True
sage: 0 in Hom(phi, rho)
True
```

To create a SageMath object representing the morphism, call the homset (`hom`):

```
sage: hom = Hom(phi, phi)
sage: frobenius_endomorphism = hom(t^6)
sage: identity_morphism = hom(1)
sage: zero_morphism = hom(0)
sage: frobenius_endomorphism
Endomorphism of Drinfeld module defined by T |--> t^2 + t + z
```

(continues on next page)

(continued from previous page)

```
  Defn: t^6
sage: identity_morphism
Identity morphism of Drinfeld module defined by T |--> t^2 + t + z
sage: zero_morphism
Endomorphism of Drinfeld module defined by T |--> t^2 + t + z
  Defn: 0
```

The underlying Ore polynomial is retrieved with the method `ore_polynomial()`:

```
sage: frobenius_endomorphism.ore_polynomial()
t^6
sage: identity_morphism.ore_polynomial()
1
```

One checks if a morphism is an isogeny, endomorphism or isomorphism:

```
sage: frobenius_endomorphism.is_isogeny()
True
sage: identity_morphism.is_isogeny()
True
sage: zero_morphism.is_isogeny()
False
sage: frobenius_endomorphism.is_isomorphism()
False
sage: identity_morphism.is_isomorphism()
True
sage: zero_morphism.is_isomorphism()
False
```

The Vélu formula

Let P be a nonzero Ore polynomial. We can decide if P defines an isogeny with a given domain and, if it does, find the codomain:

```
sage: P = (2*z^6 + z^3 + 2*z^2 + z + 2)*t + z^11 + 2*z^10 + 2*z^9 + 2*z^8 + z^7 +␣
↪2*z^6 + z^5 + z^3 + z^2 + z
sage: psi = phi.velu(P)
sage: psi
Drinfeld module defined by T |--> (2*z^11 + 2*z^9 + z^6 + 2*z^5 + 2*z^4 + 2*z^2 +␣
↪1)*t^2
 + (2*z^11 + 2*z^10 + 2*z^9 + z^8 + 2*z^7 + 2*z^6 + z^5 + 2*z^4 + 2*z^2 + 2*z)*t + z
sage: P in Hom(phi, psi)
True
sage: P * phi(T) == psi(T) * P
True
```

If the input does not define an isogeny, an exception is raised:

```
sage: phi.velu(0)
Traceback (most recent call last):
...
ValueError: the input does not define an isogeny
```

(continues on next page)

(continued from previous page)

```
sage: phi.velu(t)
Traceback (most recent call last):
...
ValueError: the input does not define an isogeny
```

The action of a Drinfeld module

The $\mathbb{F}_q[T]$-Drinfeld module ϕ induces a special left $\mathbb{F}_q[T]$-module structure on any field extension L/K. Let $x \in L$ and a be in the function ring; the action is defined as $(a, x) \mapsto \phi_a(x)$. The method *action()* returns a `sage.rings.function_field.drinfeld_modules.action.Action` object representing the Drinfeld module action.

Note: In this implementation, L is K:

```
sage: action = phi.action()
sage: action
Action on Finite Field in z of size 3^12 over its base
 induced by Drinfeld module defined by T |--> t^2 + t + z
```

The action on elements is computed by calling the action object:

```
sage: P = T + 1
sage: a = z
sage: action(P, a)
...
z^9 + 2*z^8 + 2*z^7 + 2*z^6 + 2*z^3 + z^2
sage: action(0, K.random_element())
0
sage: action(A.random_element(), 0)
0
```

Warning: The class `DrinfeldModuleAction` may be replaced later on. See issues #34833 and #34834.

action()

Return the action object (`sage.rings.function_field.drinfeld_modules.action.Action`) that represents the module action, on the base codomain, that is induced by the Drinfeld module.

OUTPUT: a Drinfeld module action object

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 + 2*z12^
↪4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: action = phi.action()
sage: action
```

(continues on next page)

(continued from previous page)

```
Action on Finite Field in z12 of size 5^12 over its base
 induced by Drinfeld module defined by T |--> z12^5*t^2 + z12^3*t + 2*z12^11
  + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 + 2*z12^4 + 3*z12^3 + z12^2 +
↪2*z12
```

The action on elements is computed as follows:

```
sage: P = T^2 + T + 1
sage: a = z12 + 1
sage: action(P, a)
3*z12^11 + 2*z12^10 + 3*z12^9 + 3*z12^7 + 4*z12^5 + z12^4 + z12^3 + 2*z12 + 1
sage: action(0, a)
0
sage: action(P, 0)
0
```

basic_j_invariant_parameters(*coeff_indices=None*, *nonzero=False*)

Return the list of basic j-invariant parameters.

See the method `j_invariant()` for definitions.

INPUT:

- `coeff_indices` (list or tuple, or NoneType; default: `None`) – indices of the Drinfeld module generator coefficients to be considered in the computation. If the parameter is `None` (default), all the coefficients are involved.
- `nonzero` (boolean, default: `False`) – if this flag is set to `True`, then only the parameters for which the corresponding basic j-invariant is nonzero are returned.

> **Warning:** The usage of this method can be computationally expensive e.g. if the rank is greater than four, or if q is large. Setting the `nonzero` flag to `True` can speed up the computation considerably if the Drinfeld module generator possesses multiple zero coefficients.

EXAMPLES:

```
sage: A = GF(5)['T']
sage: K.<T> = Frac(A)
sage: phi = DrinfeldModule(A, [T, 0, T+1, T^2 + 1])
sage: phi.basic_j_invariant_parameters()
[((1,), (31, 1)),
 ((1, 2), (1, 5, 1)),
 ((1, 2), (7, 4, 1)),
 ((1, 2), (8, 9, 2)),
 ((1, 2), (9, 14, 3)),
 ((1, 2), (10, 19, 4)),
 ((1, 2), (11, 24, 5)),
 ((1, 2), (12, 29, 6)),
 ((1, 2), (13, 3, 1)),
 ((1, 2), (15, 13, 3)),
 ((1, 2), (17, 23, 5)),
 ((1, 2), (19, 2, 1)),
 ((1, 2), (20, 7, 2)),
```

(continues on next page)

(continued from previous page)

```
 ((1, 2), (22, 17, 4)),
 ((1, 2), (23, 22, 5)),
 ((1, 2), (25, 1, 1)),
 ((1, 2), (27, 11, 3)),
 ((1, 2), (29, 21, 5)),
 ((1, 2), (31, 31, 7)),
 ((2,), (31, 6))]
```

Use the `nonzero=True` flag to display only the parameters whose j-invariant value is nonzero:

```
sage: phi.basic_j_invariant_parameters(nonzero=True)
[((2,), (31, 6))]
```

One can specify the list of coefficients indices to be considered in the computation:

```
sage: A = GF(2)['T']
sage: K.<T> = Frac(A)
sage: phi = DrinfeldModule(A, [T, T, 1, T])
sage: phi.basic_j_invariant_parameters([1, 2])
[((1,), (7, 1)),
 ((1, 2), (1, 2, 1)),
 ((1, 2), (4, 1, 1)),
 ((1, 2), (5, 3, 2)),
 ((1, 2), (6, 5, 3)),
 ((1, 2), (7, 7, 4)),
 ((2,), (7, 3))]
```

basic_j_invariants(*nonzero=False*)

Return a dictionary whose keys are all the basic j-invariants parameters and values are the corresponding j-invariant.

See the method *j_invariant()* for definitions.

INPUT:

- `nonzero` (boolean, default: `False`) – if this flag is set to `True`, then only the parameters for which the corresponding basic j-invariant is nonzero are returned.

> **Warning:** The usage of this method can be computationally expensive e.g. if the rank is greater than four, or if q is large. Setting the `nonzero` flag to `True` can speed up the computation considerably if the Drinfeld module generator possesses multiple zero coefficients.

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 + 2*z12^
↪4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: phi.basic_j_invariants()
{((1,), (26, 1)): z12^10 + 4*z12^9 + 3*z12^8 + 2*z12^7 + 3*z12^6 + z12^5 + z12^
↪3 + 4*z12^2 + z12 + 2}
```

```
sage: phi = DrinfeldModule(A, [p_root, 0, 1, z12])
sage: phi.basic_j_invariants(nonzero=True)
{((2,), (651, 26)): z12^11 + 3*z12^10 + 4*z12^9 + 3*z12^8 + z12^7 + 2*z12^6 +␣
↪3*z12^4 + 2*z12^3 + z12^2 + 4*z12}
```

```
sage: A = GF(5)['T']
sage: K.<T> = Frac(A)
sage: phi = DrinfeldModule(A, [T, T + 2, T+1, 1])
sage: J_phi = phi.basic_j_invariants(); J_phi
{((1,), (31, 1)): T^31 + 2*T^30 + 2*T^26 + 4*T^25 + 2*T^6 + 4*T^5 + 4*T + 3,
 ((1, 2), (1, 5, 1)): T^6 + 2*T^5 + T + 2,
 ((1, 2), (7, 4, 1)): T^11 + 3*T^10 + T^9 + 4*T^8 + T^7 + 2*T^6 + 2*T^4 + 3*T^3␣
↪+ 2*T^2 + 3,
 ((1, 2), (8, 9, 2)): T^17 + 2*T^15 + T^14 + 4*T^13 + 4*T^11 + 4*T^10 + 3*T^9 +␣
↪2*T^8 + 3*T^7 + 2*T^6 + 3*T^5 + 2*T^4 + 3*T^3 + 4*T^2 + 3*T + 1,
 ((1, 2), (9, 14, 3)): T^23 + 2*T^22 + 2*T^21 + T^19 + 4*T^18 + T^17 + 4*T^16 +␣
↪T^15 + 4*T^14 + 2*T^12 + 4*T^11 + 4*T^10 + 2*T^8 + 4*T^7 + 4*T^6 + 2*T^4 + T^
↪2 + 2*T + 2,
 ((1, 2), (10, 19, 4)): T^29 + 4*T^28 + T^27 + 4*T^26 + T^25 + 2*T^24 + 3*T^23␣
↪+ 2*T^22 + 3*T^21 + 2*T^20 + 4*T^19 + T^18 + 4*T^17 + T^16 + 4*T^15 + T^9 +␣
↪4*T^8 + T^7 + 4*T^6 + T^5 + 4*T^4 + T^3 + 4*T^2 + T + 4,
 ...
 ((2,), (31, 6)): T^31 + T^30 + T^26 + T^25 + T^6 + T^5 + T + 1}
sage: J_phi[((1, 2), (7, 4, 1))]
T^11 + 3*T^10 + T^9 + 4*T^8 + T^7 + 2*T^6 + 2*T^4 + 3*T^3 + 2*T^2 + 3
```

coefficient(*n*)

Return the n-th coefficient of the generator.

INPUT:

- n – a nonnegative integer

OUTPUT: an element in the base codomain

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 + 2*z12^
↪4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: phi.coefficient(0)
2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5
+ 2*z12^4 + 3*z12^3 + z12^2 + 2*z12
sage: phi.coefficient(0) == p_root
True
sage: phi.coefficient(1)
z12^3
sage: phi.coefficient(2)
z12^5
sage: phi.coefficient(5)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: input must be >= 0 and <= rank
```

coefficients(*sparse=True*)

Return the coefficients of the generator, as a list.

If the flag `sparse` is `True` (default), only return the nonzero coefficients; otherwise, return all of them.

INPUT:

- `sparse` – a boolean

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 + 2*z12^
↪4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: phi.coefficients()
[2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7
   + 2*z12^5 + 2*z12^4 + 3*z12^3 + z12^2 + 2*z12,
 z12^3,
 z12^5]
```

Careful, the method only returns the nonzero coefficients, unless otherwise specified:

```
sage: rho = DrinfeldModule(A, [p_root, 0, 0, 0, 1])
sage: rho.coefficients()
[2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7
   + 2*z12^5 + 2*z12^4 + 3*z12^3 + z12^2 + 2*z12,
 1]
sage: rho.coefficients(sparse=False)
[2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7
   + 2*z12^5 + 2*z12^4 + 3*z12^3 + z12^2 + 2*z12,
 0,
 0,
 0,
 1]
```

exponential(*name='z'*)

Return the exponential of this Drinfeld module.

Note that the exponential is only defined when the $\mathbb{F}_q[T]$-characteristic is zero.

INPUT:

- `name` (string, default: `'z'`) – the name of the generator of the lazy power series ring.

OUTPUT:

A lazy power series over the base field.

EXAMPLES:

```
sage: A = GF(2)['T']
sage: K.<T> = Frac(A)
sage: phi = DrinfeldModule(A, [T, 1])
sage: q = A.base_ring().cardinality()
sage: exp = phi.exponential(); exp
z + ((1/(T^2+T))*z^2) + ((1/(T^8+T^6+T^5+T^3))*z^4) + O(z^8)
```

The exponential is returned as a lazy power series, meaning that any of its coefficients can be computed on demands:

```
sage: exp[2^4]
1/(T^64 + T^56 + T^52 + ... + T^27 + T^23 + T^15)
sage: exp[2^5]
1/(T^160 + T^144 + T^136 + ... + T^55 + T^47 + T^31)
```

Example in higher rank:

```
sage: A = GF(5)['T']
sage: K.<T> = Frac(A)
sage: phi = DrinfeldModule(A, [T, T^2, T + T^2 + T^4, 1])
sage: exp = phi.exponential(); exp
z + ((T/(T^4+4))*z^5) + O(z^8)
```

The exponential is the compositional inverse of the logarithm (see *logarithm()*):

```
sage: log = phi.logarithm(); log
z + ((4*T/(T^4+4))*z^5) + O(z^8)
sage: exp.compose(log)
z + O(z^8)
sage: log.compose(exp)
z + O(z^8)
```

```
sage: Fq.<w> = GF(3)
sage: A = Fq['T']
sage: phi = DrinfeldModule(A, [w, 1])
sage: phi.exponential()
Traceback (most recent call last):
...
ValueError: characteristic must be zero (=T + 2)
```

REFERENCE:

See section 4.6 of [Gos1998] for the definition of the exponential.

gen()

Return the generator of the Drinfeld module.

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 + 2*z12^
↪4 + 3*z12^3 + z12^2 + 2*z12
```

(continues on next page)

(continued from previous page)

```
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: phi.gen() == phi(T)
True
```

height()

Return the height of the Drinfeld module if the function field characteristic is a prime ideal; raise ValueError otherwise.

The height of a Drinfeld module is defined when the function field characteristic is a prime ideal. In our case, this ideal is even generated by a monic polynomial $\mathfrak{p}$ in the function field. Write $\phi_\mathfrak{p} = a_s\tau^s + \dots + \tau^{r*\deg(\mathfrak{p})}$. The height of the Drinfeld module is the well-defined positive integer $h = \frac{s}{\deg(\mathfrak{p})}$.

Note: See [Gos1998], Definition 4.5.8 for the general definition.

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 + 2*z12^
↪4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: phi.height() == 1
True
sage: phi.is_ordinary()
True
```

```
sage: Fq = GF(343)
sage: A.<T> = Fq[]
sage: K.<z6> = Fq.extension(2)
sage: phi = DrinfeldModule(A, [1, 0, z6])
sage: phi.height()
2
sage: phi.is_supersingular()
True
```

In characteristic zero, height is not defined:

```
sage: L = A.fraction_field()
sage: phi = DrinfeldModule(A, [L(T), L(1)])
sage: phi.height()
Traceback (most recent call last):
...
ValueError: height is only defined for prime function field characteristic
```

hom(*x*, *codomain=None*)

Return the homomorphism defined by `x` having this Drinfeld module as domain.

We recall that a homomorphism $f : \phi \to \psi$ between two Drinfeld modules is defined by an Ore polynomial u, which is subject to the relation $phi_T u = u\psi_T$.

INPUT:

- `x` – an element of the ring of functions, or an Ore polynomial
- `codomain` – a Drinfeld module or `None` (default: `None`)

EXAMPLES:

```
sage: Fq = GF(5)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(3)
sage: phi = DrinfeldModule(A, [z, 0, 1, z])
sage: phi
Drinfeld module defined by T |--> z*t^3 + t^2 + z
```

An important class of endomorphisms of a Drinfeld module ϕ is given by scalar multiplications, that are endomorphisms corresponding to the Ore polynomials ϕ_a with a in the function ring A. We construct them as follows:

```
sage: phi.hom(T)
Endomorphism of Drinfeld module defined by T |--> z*t^3 + t^2 + z
  Defn: z*t^3 + t^2 + z
```

```
sage: phi.hom(T^2 + 1)
Endomorphism of Drinfeld module defined by T |--> z*t^3 + t^2 + z
  Defn: z^2*t^6 + (3*z^2 + z + 1)*t^5 + t^4 + 2*z^2*t^3 + (3*z^2 + z + 1)*t^2 +␣
↪z^2 + 1
```

We can also define a morphism by passing in the Ore polynomial defining it. For example, below, we construct the Frobenius endomorphism of ϕ:

```
sage: t = phi.ore_variable()
sage: phi.hom(t^3)
Endomorphism of Drinfeld module defined by T |--> z*t^3 + t^2 + z
  Defn: t^3
```

If the input Ore polynomial defines a morphism to another Drinfeld module, the latter is determined automatically:

```
sage: phi.hom(t + 1)
Drinfeld Module morphism:
  From: Drinfeld module defined by T |--> z*t^3 + t^2 + z
  To:   Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^3 + (3*z^2 + 2*z␣
↪+ 2)*t^2 + (2*z^2 + 3*z + 4)*t + z
  Defn: t + 1
```

is_finite()

Return `True` if this Drinfeld module is finite, `False` otherwise.

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 + 2*z12^
↪4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
```

(continues on next page)

(continued from previous page)

```
sage: phi.is_finite()
True
sage: B.<Y> = Fq[]
sage: L = Frac(B)
sage: psi = DrinfeldModule(A, [L(2), L(1)])
sage: psi.is_finite()
False
```

is_isomorphic(*other*, *absolutely=False*)

Return `True` if this Drinfeld module is isomorphic to `other`; return `False` otherwise.

INPUT:

- `absolutely` – a boolean (default: `False`); if `True`, check the existence of an isomorphism defined on the base field; if `False`, check over an algebraic closure.

EXAMPLES:

```
sage: Fq = GF(5)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(3)
sage: phi = DrinfeldModule(A, [z, 0, 1, z])
sage: t = phi.ore_variable()
```

We create a second Drinfeld module, which is isomorphic to ϕ and then check that they are indeed isomorphic:

```
sage: psi = phi.velu(z)
sage: phi.is_isomorphic(psi)
True
```

In the example below, ϕ and ψ are isogenous but not isomorphic:

```
sage: psi = phi.velu(t + 1)
sage: phi.is_isomorphic(psi)
False
```

Here is an example of two Drinfeld modules which are isomorphic on an algebraic closure but not on the base field:

```
sage: phi = DrinfeldModule(A, [z, 1])
sage: psi = DrinfeldModule(A, [z, z])
sage: phi.is_isomorphic(psi)
False
sage: phi.is_isomorphic(psi, absolutely=True)
True
```

On certain fields, testing isomorphisms over the base field may fail:

```
sage: L = A.fraction_field()
sage: T = L.gen()
sage: phi = DrinfeldModule(A, [T, 0, 1])
sage: psi = DrinfeldModule(A, [T, 0, T])
sage: psi.is_isomorphic(phi)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
NotImplementedError: cannot solve the equation u^24 == T
```

However, it never fails over the algebraic closure:

```
sage: psi.is_isomorphic(phi, absolutely=True)
True
```

Note finally that when the constant coefficients of ϕ_T and ψ_T differ, ϕ and ψ do not belong to the same category and checking whether they are isomorphic does not make sense; in this case, an error is raised:

```
sage: phi = DrinfeldModule(A, [z, 0, 1])
sage: psi = DrinfeldModule(A, [z^2, 0, 1])
sage: phi.is_isomorphic(psi)
Traceback (most recent call last):
...
ValueError: Drinfeld modules are not in the same category
```

j_invariant(*parameter=None*, *check=True*)

Return the j-invariant of the Drinfeld $\mathbb{F}_q[T]$-module for the given parameter.

Suppose that $\phi_T = g_0 + g_1\tau + \cdots + g_r\tau^r$ with $g_r \neq 0$. Then the $((k_1, \dots, k_n), (d_1, \dots, d_n, d_r))$-*$j$-invariant* of ϕ is defined by

$$j_{k_1,\dots,k_n}^{d_1,\dots,d_n,d_r}(\phi) := \frac{1}{g_r^{d_q}} \prod_{i=1}^{n} g_{k_i}^{d_i}$$

where $1 \leqslant k_1 < k_2 < \ldots < k_n \leqslant r-1$ and the integers d_i satisfy the *weight-0 condition*:

$$d_1(q^{k_1}-1) + d_2(q^{k_2}-1) + \cdots + d_n(q^{k_n}-1) = d_r(q^r-1).$$

Furthermore, if $\gcd(d_1, \dots, d_n, d_r) = 1$ and

$$0 \leq d_i \leq (q^r-1)/(q^{\gcd(i,r)}-1), \quad 1 \leq i \leq n,$$

then the j-invariant is called *basic*. See the method `basic_j_invariant_parameters()` for computing the list of all basic j-invariant parameters.

INPUT:

- `parameter` (tuple or list, integer or NoneType; default: `None`) – the j-invariant parameter:
  - If `parameter` is a list or a tuple, then it must be of the form: $((k_1, k_2, \dots, k_n), (d_1, d_2, \dots, d_n, d_r))$, where the k_i and d_i are integers satisfying the weight-0 condition described above.
  - If `parameter` is an integer k then the method returns the `j`-invariant associated to the parameter $((k,), (d_k, d_r))$;
  - If `parameter` is `None` and the rank of the Drinfeld module is 2, then the method returns its usual j-invariant, that is the j-invariant for the parameter $((1,), (q+1, 1))$.
- `check` (bool, default: `True`) – if this flag is set to `False` then the code will not check if the given parameter is valid and satisfy the weight-0 condition.

OUTPUT: the j-invariant of `self` for the given parameter.

REFERENCE:

The notion of basic j-invariant was introduced by Potemine in [Pot1998].

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 + 2*z12^
↪4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: phi.j_invariant()
z12^10 + 4*z12^9 + 3*z12^8 + 2*z12^7 + 3*z12^6 + z12^5 + z12^3 + 4*z12^2 + z12␣
↪+ 2
sage: psi = DrinfeldModule(A, [p_root, 1, 1])
sage: psi.j_invariant()
1
sage: rho = DrinfeldModule(A, [p_root, 0, 1])
sage: rho.j_invariant()
0
```

```
sage: A = GF(5)['T']
sage: K.<T> = Frac(A)
sage: phi = DrinfeldModule(A, [T, T^2, 1, T + 1, T^3])
sage: phi.j_invariant(1)
T^309
sage: phi.j_invariant(2)
1/T^3
sage: phi.j_invariant(3)
(T^156 + T^155 + T^151 + T^150 + T^131 + T^130 + T^126 + T^125 + T^31 + T^30 +␣
↪T^26 + T^25 + T^6 + T^5 + T + 1)/T^93
```

The parameter can either be a tuple or a list:

```
sage: Fq.<a> = GF(7)
sage: A.<T> = Fq[]
sage: phi = DrinfeldModule(A, [a, a^2 + a, 0, 3*a, a^2+1])
sage: J = phi.j_invariant(((1, 3), (267, 269, 39))); J
5
sage: J == (phi.coefficient(1)**267)*(phi.coefficient(3)**269)/(phi.
↪coefficient(4)**39)
True
sage: phi.j_invariant([[3], [400, 57]])
4
sage: phi.j_invariant([[3], [400, 57]]) == phi.j_invariant(3)
True
```

The list of all basic j-invariant parameters can be retrieved using the method *basic_j_invariant_parameters()*:

```
sage: A = GF(3)['T']
sage: K.<T> = Frac(A)
```

(continues on next page)

(continued from previous page)

```
sage: phi = DrinfeldModule(A, [T, T^2 + T + 1, 0, T^4 + 1, T - 1])
sage: param = phi.basic_j_invariant_parameters(nonzero=True)
sage: phi.j_invariant(param[1])
T^13 + 2*T^12 + T + 2
sage: phi.j_invariant(param[2])
T^35 + 2*T^31 + T^27 + 2*T^8 + T^4 + 2
```

jk_invariants()

Return a dictionary whose keys are all the integers $1 \leqslant k \leqslant r-1$ and the values are the corresponding j_k-invariants

Recall that the j_k-invariant of self is defined by:

$$j_k := \frac{g_k^{(q^r-1)/(\gcd(k,r)-1)}}{g_r^{(q^k-1)/(\gcd(k,r)-1)}}$$

where g_i is the i-th coefficient of the generator of self.

EXAMPLES:

```
sage: A = GF(3)['T']
sage: K.<T> = Frac(A)
sage: phi = DrinfeldModule(A, [T, 1, T+1, T^3, T^6])
sage: jk_inv = phi.jk_invariants(); jk_inv
{1: 1/T^6, 2: (T^10 + T^9 + T + 1)/T^6, 3: T^42}
sage: jk_inv[2]
(T^10 + T^9 + T + 1)/T^6
```

```
sage: F = GF(7**2)
sage: A = F['T']
sage: E.<z> = F.extension(4)
sage: phi = DrinfeldModule(A, [z^2, 1, z+1, z^2, z, z+1])
sage: phi.jk_invariants()
{1: 5*z^7 + 2*z^6 + 5*z^5 + 2*z^4 + 5*z^3 + z^2 + z + 2,
 2: 3*z^7 + 4*z^6 + 5*z^5 + 6*z^4 + 4*z,
 3: 5*z^7 + 6*z^6 + 6*z^5 + 4*z^3 + z^2 + 2*z + 1,
 4: 3*z^6 + 2*z^5 + 4*z^4 + 2*z^3 + 4*z^2 + 6*z + 2}
```

logarithm(*name='z'*)

Return the logarithm of the given Drinfeld module.

By definition, the logarithm is the compositional inverse of the exponential (see *exponential()*). Note that the logarithm is only defined when the $\mathbb{F}_q[T]$-characteristic is zero.

INPUT:

- `name` (string, default: `'z'`) – the name of the generator of the lazy power series ring.

OUTPUT:

A lazy power series over the base field.

EXAMPLES:

```
sage: A = GF(2)['T']
sage: K.<T> = Frac(A)
sage: phi = DrinfeldModule(A, [T, 1])
sage: log = phi.logarithm(); log
z + ((1/(T^2+T))*z^2) + ((1/(T^6+T^5+T^3+T^2))*z^4) + O(z^8)
```

The logarithm is returned as a lazy power series, meaning that any of its coefficients can be computed on demands:

```
sage: log[2^4]
1/(T^30 + T^29 + T^27 + ... + T^7 + T^5 + T^4)
sage: log[2^5]
1/(T^62 + T^61 + T^59 + ... + T^8 + T^6 + T^5)
```

Example in higher rank:

```
sage: A = GF(5)['T']
sage: K.<T> = Frac(A)
sage: phi = DrinfeldModule(A, [T, T^2, T + T^2 + T^4, 1])
sage: phi.logarithm()
z + ((4*T/(T^4+4))*z^5) + O(z^8)
```

morphism()

Return the morphism object that defines the Drinfeld module.

OUTPUT: a ring morphism from the function ring to the Ore polynomial ring

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 + 2*z12^
↪4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: phi.morphism()
Ring morphism:
  From: Univariate Polynomial Ring in T over Finite Field in z2 of size 5^2
  To:   Ore Polynomial Ring in t over Finite Field in z12 of size 5^12
        over its base twisted by Frob^2
  Defn: T |--> z12^5*t^2 + z12^3*t + 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8
                 + z12^7 + 2*z12^5 + 2*z12^4 + 3*z12^3 + z12^2 + 2*z12
sage: from sage.rings.morphism import RingHomomorphism
sage: isinstance(phi.morphism(), RingHomomorphism)
True
```

Actually, the `DrinfeldModule` method `__call__()` simply class the `__call__` method of this morphism:

```
sage: phi.morphism()(T) == phi(T)
True
sage: a = A.random_element()
sage: phi.morphism()(a) == phi(a)
True
```

And many methods of the Drinfeld module have a counterpart in the morphism object:

```
sage: m = phi.morphism()
sage: m.domain() is phi.function_ring()
True
sage: m.codomain() is phi.ore_polring()
True
sage: m.im_gens()
[z12^5*t^2 + z12^3*t + 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8
 + z12^7 + 2*z12^5 + 2*z12^4 + 3*z12^3 + z12^2 + 2*z12]
sage: phi(T) == m.im_gens()[0]
True
```

rank()

Return the rank of the Drinfeld module.

In our case, the rank is the degree of the generator.

OUTPUT: an integer

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 + 2*z12^
↪4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: phi.rank()
2
sage: psi = DrinfeldModule(A, [p_root, 2])
sage: psi.rank()
1
sage: rho = DrinfeldModule(A, [p_root, 0, 0, 0, 1])
sage: rho.rank()
4
```

scalar_multiplication(x)

Return the endomorphism of this Drinfeld module, which is the multiplication by x, i.e. the isogeny defined by the Ore polynomial ϕ_x.

INPUT:

- `x` – an element in the ring of functions

EXAMPLES:

```
sage: Fq = GF(5)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(3)
sage: phi = DrinfeldModule(A, [z, 0, 1, z])
sage: phi
Drinfeld module defined by T |--> z*t^3 + t^2 + z
sage: phi.hom(T)
Endomorphism of Drinfeld module defined by T |--> z*t^3 + t^2 + z
  Defn: z*t^3 + t^2 + z
```

```
sage: phi.hom(T^2 + 1)
Endomorphism of Drinfeld module defined by T |--> z*t^3 + t^2 + z
  Defn: z^2*t^6 + (3*z^2 + z + 1)*t^5 + t^4 + 2*z^2*t^3 + (3*z^2 + z + 1)*t^2 +␣
↪z^2 + 1
```

velu(*isog*)

Return a new Drinfeld module such that input is an isogeny to this module with domain `self`; if no such isogeny exists, raise an exception.

INPUT:

- `isog` – the Ore polynomial that defines the isogeny

OUTPUT: a Drinfeld module

ALGORITHM:

The input defines an isogeny if only if:

1. The degree of the characteristic divides the height of the input. (The height of an Ore polynomial $P(\tau)$ is the maximum n such that τ^n right-divides $P(\tau)$.)

2. The input right-divides the generator, which can be tested with Euclidean division.

We test if the input is an isogeny, and, if it is, we return the quotient of the Euclidean division.

Height and Euclidean division of Ore polynomials are implemented as methods of class `sage.rings.polynomial.ore_polynomial_element.OrePolynomial`.

Another possible algorithm is to recursively solve a system, see arXiv 2203.06970, Eq. 1.1.

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 + 2*z12^
↪4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: t = phi.ore_polring().gen()
sage: isog = t + 2*z12^11 + 4*z12^9 + 2*z12^8 + 2*z12^6 + 3*z12^5 + z12^4 +␣
↪2*z12^3 + 4*z12^2 + 4*z12 + 4
sage: psi = phi.velu(isog)
sage: psi
Drinfeld module defined by T |-->
 (z12^11 + 3*z12^10 + z12^9 + z12^7 + z12^5 + 4*z12^4 + 4*z12^3 + z12^2 + 1)*t^2
 + (2*z12^11 + 4*z12^10 + 2*z12^8 + z12^6 + 3*z12^5 + z12^4 + 2*z12^3 + z12^2 +␣
↪z12 + 4)*t
 + 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 + 2*z12^4 + 3*z12^3␣
↪+ z12^2 + 2*z12
sage: isog in Hom(phi, psi)
True
```

This method works for endomorphisms as well:

```
sage: phi.velu(phi(T)) is phi
True
sage: phi.velu(t^6) is phi
True
```

The following inputs do not define isogenies, and the method returns None:

```
sage: phi.velu(0)
Traceback (most recent call last):
...
ValueError: the input does not define an isogeny
sage: phi.velu(t)
Traceback (most recent call last):
...
ValueError: the input does not define an isogeny
sage: phi.velu(t^3 + t + 2)
Traceback (most recent call last):
...
ValueError: the input does not define an isogeny
```

1.2 Finite Drinfeld modules

This module provides the class `sage.rings.function_fields.drinfeld_module.finite_drinfeld_module.FiniteDrinfeldModule`, which inherits `sage.rings.function_fields.drinfeld_module.drinfeld_module.DrinfeldModule`.

AUTHORS:

- Antoine Leudière (2022-04)

class `sage.rings.function_field.drinfeld_modules.finite_drinfeld_module.`**`FiniteDrinfeldModule`**(*gen*, *category*)

Bases: *DrinfeldModule*

This class implements finite Drinfeld $\mathbb{F}_q[T]$-modules.

A *finite Drinfeld module* is a Drinfeld module whose base field is finite. In this case, the function field characteristic is a prime ideal.

For general definitions and help on Drinfeld modules, see class `sage.rings.function_fields.drinfeld_module.drinfeld_module.DrinfeldModule`.

Construction:

The user does not ever need to directly call `FiniteDrinfeldModule` — the metaclass `DrinfeldModule` is responsible for instantiating `DrinfeldModule` or `FiniteDrinfeldModule` depending on the input:

```
sage: Fq = GF(343)
sage: A.<T> = Fq[]
sage: K.<z6> = Fq.extension(2)
sage: phi = DrinfeldModule(A, [z6, 0, 5])
sage: phi
Drinfeld module defined by T |--> 5*t^2 + z6
```

```
sage: isinstance(phi, DrinfeldModule)
True
```

(continues on next page)

(continued from previous page)

```
sage: from sage.rings.function_field.drinfeld_modules.finite_drinfeld_module import␣
↪FiniteDrinfeldModule
sage: isinstance(phi, FiniteDrinfeldModule)
True
```

The user should never use `FiniteDrinfeldModule` to test if a Drinfeld module is finite, but rather the `is_finite` method:

```
sage: phi.is_finite()
True
```

Complex multiplication of rank two finite Drinfeld modules

We can handle some aspects of the theory of complex multiplication of finite Drinfeld modules. Apart from the method `frobenius_endomorphism`, we only handle rank two Drinfeld modules.

First of all, it is easy to create the Frobenius endomorphism:

```
sage: frobenius_endomorphism = phi.frobenius_endomorphism()
sage: frobenius_endomorphism
Endomorphism of Drinfeld module defined by T |--> 5*t^2 + z6
  Defn: t^2
```

Its characteristic polynomial can be computed:

```
sage: chi = phi.frobenius_charpoly()
sage: chi
X^2 + (T + 2*z3^2 + 2*z3 + 1)*X + 2*T^2 + (z3^2 + z3 + 4)*T + 2*z3
sage: frob_pol = frobenius_endomorphism.ore_polynomial()
sage: chi(frob_pol, phi(T))
0
```

as well as its trace and norm:

```
sage: phi.frobenius_trace()
6*T + 5*z3^2 + 5*z3 + 6
sage: phi.frobenius_trace() == -chi[1]
True
sage: phi.frobenius_norm()
2*T^2 + (z3^2 + z3 + 4)*T + 2*z3
```

We can decide if a Drinfeld module is ordinary or supersingular:

```
sage: phi.is_ordinary()
True
sage: phi.is_supersingular()
False
```

Inverting the Drinfeld module

The morphism that defines a Drinfeld module is injective (see [Gos1998], cor. 4.5.2). If the Drinfeld module is finite, one can retrieve preimages:

```
sage: a = A.random_element()
sage: phi.invert(phi(a)) == a
True
```

frobenius_charpoly(*var='X'*)

Return the characteristic polynomial of the Frobenius endomorphism if the rank is two. Raise a NotImplementedError otherwise.

Let $\mathbb{F}_q$ be the base field of the function ring. The *characteristic polynomial `chi` of the Frobenius endomorphism* is defined in [Gek1991]. An important feature of this polynomial is that it is a monic univariate polynomial with coefficients in the function ring. As in our case the function ring is a univariate polynomial ring, it is customary to see the characteristic polynomial of the Frobenius endomorphism as a bivariate polynomial.

Let $\chi = X^2 - A(T)X + B(T)$ be the characteristic polynomial of the Frobenius endomorphism, and let t^n be the Ore polynomial that defines the Frobenius endomorphism of ϕ; by definition, n is the degree over of the base field over $\mathbb{F}_q$. We have $\chi(t^n)(\phi(T)) = t^{2n} - \phi_A t^n + \phi_B = 0$, with $\deg(A) \leq \frac{n}{2}$ and $\deg(B) = n$.

Note that the *Frobenius trace* is defined as $A(T)$ and the *Frobenius norm* is defined as $B(T)$.

INPUT:

- `var` (default: `'X'`) – the name of the second variable

EXAMPLES:

```
sage: Fq = GF(343)
sage: A.<T> = Fq[]
sage: K.<z6> = Fq.extension(2)
sage: phi = DrinfeldModule(A, [1, 0, z6])
sage: chi = phi.frobenius_charpoly()
sage: chi
X^2 + ((3*z3^2 + z3 + 4)*T + 4*z3^2 + 6*z3 + 3)*X
 + (5*z3^2 + 2*z3)*T^2 + (4*z3^2 + 3*z3)*T + 5*z3^2 + 2*z3
```

```
sage: frob_pol = phi.frobenius_endomorphism().ore_polynomial()
sage: chi(frob_pol, phi(T))
0
```

```
sage: trace = phi.frobenius_trace()
sage: trace
(4*z3^2 + 6*z3 + 3)*T + 3*z3^2 + z3 + 4
sage: norm = phi.frobenius_norm()
sage: norm
(5*z3^2 + 2*z3)*T^2 + (4*z3^2 + 3*z3)*T + 5*z3^2 + 2*z3
```

```
sage: n = 2  # Degree of the base field over Fq
sage: trace.degree() <= n/2
True
sage: norm.degree() == n
True
```

ALGORITHM:

We compute the Frobenius norm, and with it the Frobenius trace. This gives the Frobenius characteristic polynomial. See [MS2019], Section 4.

See docstrings of methods *frobenius_norm()* and *frobenius_trace()* for further details on the computation of the norm and of the trace.

frobenius_endomorphism()

Return the Frobenius endomorphism of the Drinfeld module as a morphism object.

Let q be the order of the base field of the function ring. The *Frobenius endomorphism* is defined as the endomorphism whose defining Ore polynomial is t^q.

EXAMPLES:

```
sage: Fq = GF(343)
sage: A.<T> = Fq[]
sage: K.<z6> = Fq.extension(2)
sage: phi = DrinfeldModule(A, [1, 0, z6])
sage: phi.frobenius_endomorphism()
Endomorphism of Drinfeld module defined by T |--> z6*t^2 + 1
  Defn: t^2
```

frobenius_norm()

Return Frobenius norm of the Drinfeld module, if the rank is two, raise a NotImplementedError otherwise.

Let $\mathbb{F}_q[T]$ be the function ring, write $\chi = X^2 - A(T)X + B(T) \in \mathbb{F}_q[T][X]$ for the characteristic polynomial of the Frobenius endomorphism. The *Frobenius norm* is defined as the polynomial $B(T) \in \mathbb{F}_q[T]$.

Let n be the degree of the base field over $\mathbb{F}_q$ Then the Frobenius norm has degree n.

EXAMPLES:

```
sage: Fq = GF(343)
sage: A.<T> = Fq[]
sage: K.<z6> = Fq.extension(2)
sage: phi = DrinfeldModule(A, [1, 0, z6])
sage: B = phi.frobenius_norm()
sage: B
(5*z3^2 + 2*z3)*T^2 + (4*z3^2 + 3*z3)*T + 5*z3^2 + 2*z3
```

```
sage: n = 2  # Degree of the base field over Fq
sage: B.degree() == n
True
```

```
sage: B == phi.frobenius_charpoly()[0]
True
```

ALGORITHM:

The Frobenius norm is computed using the formula, by Gekeler, given in [MS2019], Section 4, Proposition 3.

frobenius_trace()

Return Frobenius norm of the Drinfeld module, if the rank is two; raise a NotImplementedError otherwise.

Let $\mathbb{F}_q[T]$ be the function ring, write $\chi = T^2 - A(X)T + B(X) \in \mathbb{F}_q[T][X]$ for the characteristic polynomial of the Frobenius endomorphism. The *Frobenius trace* is defined as the polynomial $A(T) \in \mathbb{F}_q[T]$.

Let n be the degree over $\mathbb{F}_q$ of the base codomain. Then the Frobenius trace has degree at most $\frac{n}{2}$.

EXAMPLES:

```
sage: Fq = GF(343)
sage: A.<T> = Fq[]
sage: K.<z6> = Fq.extension(2)
sage: phi = DrinfeldModule(A, [1, 0, z6])
sage: A = phi.frobenius_trace()
sage: A
(4*z3^2 + 6*z3 + 3)*T + 3*z3^2 + z3 + 4
```

```
sage: n = 2  # Degree over Fq of the base codomain
sage: A.degree() <= n/2
True
```

```
sage: A == -phi.frobenius_charpoly()[1]
True
```

ALGORITHM:

Let $A(T)$ denote the Frobenius trace and $B(T)$ denote the Frobenius norm. We begin by computing $B(T)$, see docstring of method `frobenius_norm()` for details. The characteristic polynomial of the Frobenius yields $t^{2n} - \phi_A t^n + \phi_B = 0$, where t^n is the Frobenius endomorphism. As ϕ_B is now known, we can compute $\phi_A = (t^{2n} + \phi_B)/t^n$. We get $A(T)$ by inverting this quantity, using the method `sage.rings.function_fields.drinfeld_module.drinfeld_module.DrinfeldModule.invert()`, see its docstring for details.

invert(*ore_pol*)

Return the preimage of the input under the Drinfeld module, if it exists.

INPUT:

- `ore_pol` – the Ore polynomial whose preimage we want to compute

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 + 2*z12^
↪4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: a = A.random_element()
sage: phi.invert(phi(a)) == a
True
sage: phi.invert(phi(T)) == T
True
sage: phi.invert(phi(Fq.gen())) == Fq.gen()
True
```

When the input is not in the image of the Drinfeld module, an exception is raised:

```
sage: t = phi.ore_polring().gen()
sage: phi.invert(t + 1)
Traceback (most recent call last):
...
ValueError: input must be in the image of the Drinfeld module
```

```
sage: phi.invert(t^4 + t^2 + 1)
Traceback (most recent call last):
...
ValueError: input must be in the image of the Drinfeld module
```

ALGORITHM:

The algorithm relies on the inversion of a linear algebra system. See [MS2019], 3.2.5 for details.

is_ordinary()

Return `True` if this Drinfeld module is ordinary.

A Drinfeld module is ordinary if and only if its height is one.

EXAMPLES:

```
sage: Fq = GF(343)
sage: A.<T> = Fq[]
sage: K.<z6> = Fq.extension(2)
sage: phi = DrinfeldModule(A, [1, 0, z6])
sage: phi.is_ordinary()
False
```

```
sage: phi = DrinfeldModule(A, [1, z6, 0, z6])
sage: phi.is_ordinary()
True
```

is_supersingular()

Return `True` if this Drinfeld module is supersingular.

A Drinfeld module is supersingular if and only if its height equals its rank.

EXAMPLES:

```
sage: Fq = GF(343)
sage: A.<T> = Fq[]
sage: K.<z6> = Fq.extension(2)
sage: phi = DrinfeldModule(A, [1, 0, z6])
sage: phi.is_supersingular()
True
sage: phi(phi.characteristic())   # Purely inseparable
z6*t^2
```

In rank two, a Drinfeld module is either ordinary or supersinguler. In higher ranks, it could be neither of the two:

```
sage: psi = DrinfeldModule(A, [1, 0, z6, z6])
sage: psi.is_ordinary()
False
```

(continues on next page)

(continued from previous page)

```
sage: psi.is_supersingular()
False
```

CHAPTER

TWO

MORPHISMS AND ISOGENIES

2.1 Drinfeld module morphisms

This module provides the class `sage.rings.function_fields.drinfeld_module.morphism.DrinfeldModuleMorphism`.

AUTHORS: - Antoine Leudière (2022-04)

class `sage.rings.function_field.drinfeld_modules.morphism.`**`DrinfeldModuleMorphism`**(*parent*, *ore_pol*)

Bases: `Morphism`, `UniqueRepresentation`

This class represents Drinfeld $\mathbb{F}_q[T]$-module morphisms.

Let ϕ and ψ be two Drinfeld $\mathbb{F}_q[T]$-modules over a field K. A *morphism of Drinfeld modules* $\phi \to \psi$ is an Ore polynomial $f \in K\{\tau\}$ such that $f\phi_a = \psi_a f$ for every $a \in \mathbb{F}_q[T]$. In our case, this is equivalent to $f\phi_T = \psi_T f$. An *isogeny* is a nonzero morphism.

To create a morphism object, the user should never explicitly instantiate *`DrinfeldModuleMorphism`*, but rather call the parent homset with the defining Ore polynomial:

```
sage: Fq = GF(4)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(3)
sage: phi = DrinfeldModule(A, [z, z^2 + z, z^2 + z])
sage: t = phi.ore_polring().gen()
sage: ore_pol = t + z^5 + z^3 + z + 1
sage: psi = phi.velu(ore_pol)
sage: morphism = Hom(phi, psi)(ore_pol)
sage: morphism
Drinfeld Module morphism:
  From: Drinfeld module defined by T |--> (z^2 + z)*t^2 + (z^2 + z)*t + z
  To:   Drinfeld module defined by T |--> (z^5 + z^2 + z + 1)*t^2 + (z^4 + z + 1)*t␣
↪+ z
  Defn: t + z^5 + z^3 + z + 1
```

The given Ore polynomial must indeed define a morphism:

```
sage: morphism = Hom(phi, psi)(1)
Traceback (most recent call last):
...
ValueError: Ore polynomial does not define a morphism
```

One can get basic data on the morphism:

```
sage: morphism.domain()
Drinfeld module defined by T |--> (z^2 + z)*t^2 + (z^2 + z)*t + z
sage: morphism.domain() is phi
True

sage: morphism.codomain()
Drinfeld module defined by T |--> (z^5 + z^2 + z + 1)*t^2 + (z^4 + z + 1)*t + z
sage: morphism.codomain() is psi
True
```

```
sage: morphism.ore_polynomial()
t + z^5 + z^3 + z + 1
sage: morphism.ore_polynomial() is ore_pol
True
```

One can check various properties:

```
sage: morphism.is_zero()
False
sage: morphism.is_isogeny()
True
sage: morphism.is_endomorphism()
False
sage: morphism.is_isomorphism()
False
```

characteristic_polynomial(*var='X'*)

Return the characteristic polynomial of this endomorphism.

INPUT:

- var – a string (default: X), the name of the variable of the characteristic polynomial

EXAMPLES:

```
sage: Fq = GF(5)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(3)
sage: phi = DrinfeldModule(A, [z, 0, 1, z])

sage: f = phi.frobenius_endomorphism()
sage: f.characteristic_polynomial()
X^3 + (T + 1)*X^2 + (2*T + 3)*X + 2*T^3 + T + 1
```

We verify, on an example, that the caracteristic polynomial of a morphism corresponding to ϕ_a is $(X - a)^r$ where r is the rank:

```
sage: g = phi.hom(T^2 + 1)
sage: chi = g.characteristic_polynomial()
sage: chi.factor()
(X + 4*T^2 + 4)^3
```

An example with another variable name:

```
sage: f.characteristic_polynomial(var='Y')
Y^3 + (T + 1)*Y^2 + (2*T + 3)*Y + 2*T^3 + T + 1
```

charpoly(*var='X'*)

Return the characteristic polynomial of this endomorphism.

INPUT:

- `var` – a string (default: `X`), the name of the variable of the characteristic polynomial

EXAMPLES:

```
sage: Fq = GF(5)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(3)
sage: phi = DrinfeldModule(A, [z, 0, 1, z])

sage: f = phi.frobenius_endomorphism()
sage: chi = f.charpoly()
sage: chi
X^3 + (T + 1)*X^2 + (2*T + 3)*X + 2*T^3 + T + 1
```

We check that the characteristic polynomial annihilates the morphism (Cayley-Hamilton's theorem):

```
sage: chi(f)
Endomorphism of Drinfeld module defined by T |--> z*t^3 + t^2 + z
  Defn: 0
```

We verify, on an example, that the caracteristic polynomial of the morphism corresponding to ϕ_a is $(X-a)^r$ where r is the rank:

```
sage: g = phi.hom(T^2 + 1)
sage: g.charpoly().factor()
(X + 4*T^2 + 4)^3
```

An example with another variable name:

```
sage: f.charpoly(var='Y')
Y^3 + (T + 1)*Y^2 + (2*T + 3)*Y + 2*T^3 + T + 1
```

dual_isogeny()

Return a dual isogeny to this morphism.

By definition, a dual isogeny of $f : \phi \to \psi$ is an isogeny $g : \psi \to \phi$ such that the composite $g \circ f$ is the multiplication by a generator of the norm of f.

EXAMPLES:

```
sage: Fq = GF(5)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(3)
sage: phi = DrinfeldModule(A, [z, 0, 1, z])
sage: t = phi.ore_variable()
sage: f = phi.hom(t + 1)
sage: f
Drinfeld Module morphism:
```

(continues on next page)

(continued from previous page)

```
  From: Drinfeld module defined by T |--> z*t^3 + t^2 + z
  To:   Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^3 + (3*z^2 + 2*z␣
↪+ 2)*t^2 + (2*z^2 + 3*z + 4)*t + z
  Defn: t + 1
sage: g = f.dual_isogeny()
sage: g
Drinfeld Module morphism:
  From: Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^3 + (3*z^2 + 2*z␣
↪+ 2)*t^2 + (2*z^2 + 3*z + 4)*t + z
  To:   Drinfeld module defined by T |--> z*t^3 + t^2 + z
  Defn: z*t^2 + (4*z + 1)*t + z + 4
```

We check that $f \circ g$ (resp. $g \circ f$) is the multiplication by the norm of f:

```
sage: a = f.norm().gen(); a
T + 4
sage: g * f == phi.hom(a)
True

sage: psi = f.codomain()
sage: f * g == psi.hom(a)
True
```

inverse()

Return the inverse of this morphism.

Only morphisms defined by constant nonzero Ore polynomials are invertible.

EXAMPLES:

```
sage: Fq = GF(5)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(3)
sage: phi = DrinfeldModule(A, [z, 1, z, z^2])
sage: f = phi.hom(2); f
Endomorphism of Drinfeld module defined by T |--> z^2*t^3 + z*t^2 + t + z
  Defn: 2
sage: f.inverse()
Endomorphism of Drinfeld module defined by T |--> z^2*t^3 + z*t^2 + t + z
  Defn: 3
```

Inversion of general isomorphisms between different Drinfeld modules also works:

```
sage: g = phi.hom(z); g
Drinfeld Module morphism:
  From: Drinfeld module defined by T |--> z^2*t^3 + z*t^2 + t + z
  To:   Drinfeld module defined by T |--> z^2*t^3 + (z^2 + 2*z + 3)*t^2 + (z^2␣
↪+ 3*z)*t + z
  Defn: z
sage: g.inverse()
Drinfeld Module morphism:
  From: Drinfeld module defined by T |--> z^2*t^3 + (z^2 + 2*z + 3)*t^2 + (z^2␣
↪+ 3*z)*t + z
```

(continues on next page)

(continued from previous page)

```
  To:   Drinfeld module defined by T |--> z^2*t^3 + z*t^2 + t + z
  Defn: 3*z^2 + 4
```

When the morphism is not invertible, an error is raised:

```
sage: F = phi.frobenius_endomorphism()
sage: F.inverse()
Traceback (most recent call last):
...
ZeroDivisionError: this morphism is not invertible
```

is_identity()

Return True whether the morphism is the identity morphism.

EXAMPLES:

```
sage: Fq = GF(2)
sage: A.<T> = Fq[]
sage: K.<z6> = Fq.extension(6)
sage: phi = DrinfeldModule(A, [z6, 1, 1])
sage: morphism = End(phi)(1)
sage: morphism.is_identity()
True
```

```
sage: psi = DrinfeldModule(A, [z6, z6^4 + z6^2 + 1, 1])
sage: t = phi.ore_polring().gen()
sage: morphism = Hom(phi, psi)(t + z6^5 + z6^2 + 1)
sage: morphism.is_identity()
False
```

is_isogeny()

Return True whether the morphism is an isogeny.

EXAMPLES:

```
sage: Fq = GF(2)
sage: A.<T> = Fq[]
sage: K.<z6> = Fq.extension(6)
sage: phi = DrinfeldModule(A, [z6, 1, 1])
sage: psi = DrinfeldModule(A, [z6, z6^4 + z6^2 + 1, 1])
sage: t = phi.ore_polring().gen()
sage: morphism = Hom(phi, psi)(t + z6^5 + z6^2 + 1)
sage: morphism.is_isogeny()
True
```

```
sage: zero_morphism = End(phi)(0)
sage: zero_morphism.is_isogeny()
False
```

```
sage: identity_morphism = End(phi)(1)
sage: identity_morphism.is_isogeny()
True
```

```
sage: frobenius_endomorphism = phi.frobenius_endomorphism()
sage: frobenius_endomorphism.is_isogeny()
True
```

is_isomorphism()

Return True whether the morphism is an isomorphism.

EXAMPLES:

```
sage: Fq = GF(2)
sage: A.<T> = Fq[]
sage: K.<z6> = Fq.extension(6)
sage: phi = DrinfeldModule(A, [z6, 1, 1])
sage: psi = DrinfeldModule(A, [z6, z6^4 + z6^2 + 1, 1])
sage: t = phi.ore_polring().gen()
sage: morphism = Hom(phi, psi)(t + z6^5 + z6^2 + 1)
sage: morphism.is_isomorphism()
False
```

```
sage: zero_morphism = End(phi)(0)
sage: zero_morphism.is_isomorphism()
False
```

```
sage: identity_morphism = End(phi)(1)
sage: identity_morphism.is_isomorphism()
True
```

```
sage: frobenius_endomorphism = phi.frobenius_endomorphism()
sage: frobenius_endomorphism.is_isomorphism()
False
```

is_zero()

Return True whether the morphism is the zero morphism.

EXAMPLES:

```
sage: Fq = GF(2)
sage: A.<T> = Fq[]
sage: K.<z6> = Fq.extension(6)
sage: phi = DrinfeldModule(A, [z6, 1, 1])
sage: psi = DrinfeldModule(A, [z6, z6^4 + z6^2 + 1, 1])
sage: t = phi.ore_polring().gen()
sage: morphism = Hom(phi, psi)(t + z6^5 + z6^2 + 1)
sage: morphism.is_zero()
False
```

```
sage: zero_morphism = End(phi)(0)
sage: zero_morphism.is_zero()
True
```

norm(*ideal=True*)

Return the norm of this isogeny.

INPUT:

- `ideal` – a boolean (default: `True`); if `True`, return the norm as an ideal in the function ring of the Drinfeld modules; if `False`, return the norm as an element in this function ring (only relevant for endomorphisms)

EXAMPLES:

```
sage: Fq = GF(5)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(3)
sage: phi = DrinfeldModule(A, [z, 0, 1, z])
sage: t = phi.ore_variable()
sage: f = phi.hom(t + 1)
sage: f.norm()
Principal ideal (T + 4) of Univariate Polynomial Ring in T over Finite Field of␣
↪size 5
```

The norm of the Frobenius endomorphism is equal to the characteristic:

```
sage: F = phi.frobenius_endomorphism()
sage: F.norm()
Principal ideal (T^3 + 3*T + 3) of Univariate Polynomial Ring in T over Finite␣
↪Field of size 5
sage: phi.characteristic()
T^3 + 3*T + 3
```

For a in the underlying function ring, the norm of the endomorphism given by ϕ_a is a^r where r is the rank:

```
sage: g = phi.hom(T)
sage: g.norm()
Principal ideal (T^3) of Univariate Polynomial Ring in T over Finite Field of␣
↪size 5

sage: h = phi.hom(T+1)
sage: h.norm()
Principal ideal (T^3 + 3*T^2 + 3*T + 1) of Univariate Polynomial Ring in T over␣
↪Finite Field of size 5
```

For endomorphisms, the norm is not an ideal of A but it makes sense as an actual element of A. We can get this element by passing in the argument `ideal=False`:

```
sage: phi.hom(2*T).norm(ideal=False)
3*T^3

sage: f.norm(ideal=False)
Traceback (most recent call last):
...
ValueError: norm is defined as an actual element only for endomorphisms
```

ore_polynomial()

Return the Ore polynomial that defines the morphism.

EXAMPLES:

```
sage: Fq = GF(2)
sage: A.<T> = Fq[]
```

(continues on next page)

(continued from previous page)

```
sage: K.<z6> = Fq.extension(6)
sage: phi = DrinfeldModule(A, [z6, 1, 1])
sage: psi = DrinfeldModule(A, [z6, z6^4 + z6^2 + 1, 1])
sage: t = phi.ore_polring().gen()
sage: morphism = Hom(phi, psi)(t + z6^5 + z6^2 + 1)
sage: ore_pol = morphism.ore_polynomial()
sage: ore_pol
t + z6^5 + z6^2 + 1
```

```
sage: ore_pol * phi(T) == psi(T) * ore_pol
True
```

2.2 Set of morphisms between two Drinfeld modules

This module provides the class `sage.rings.function_field.drinfeld_module.homset.DrinfeldModuleHomset`.

AUTHORS:

- Antoine Leudière (2022-04)

class `sage.rings.function_field.drinfeld_modules.homset.`**`DrinfeldModuleHomset`**(*X*, *Y*, *category=None*, *check=True*)

Bases: `Homset`

This class implements the set of morphisms between two Drinfeld $\mathbb{F}_q[T]$-modules.

INPUT:

- `X` – the domain
- `Y` – the codomain

EXAMPLES:

```
sage: Fq = GF(27)
sage: A.<T> = Fq[]
sage: K.<z6> = Fq.extension(2)
sage: phi = DrinfeldModule(A, [z6, z6, 2])
sage: psi = DrinfeldModule(A, [z6, 2*z6^5 + 2*z6^4 + 2*z6 + 1, 2])
sage: H = Hom(phi, psi)
sage: H
Set of Drinfeld module morphisms
 from (gen) 2*t^2 + z6*t + z6
   to (gen) 2*t^2 + (2*z6^5 + 2*z6^4 + 2*z6 + 1)*t + z6
```

```
sage: from sage.rings.function_field.drinfeld_modules.homset import
↪DrinfeldModuleHomset
sage: isinstance(H, DrinfeldModuleHomset)
True
```

There is a simpler syntax for endomorphisms sets:

```
sage: E = End(phi)
sage: E
Set of Drinfeld module morphisms from (gen) 2*t^2 + z6*t + z6 to (gen) 2*t^2 + z6*t␣
↪+ z6
sage: E is Hom(phi, phi)
True
```

The domain and codomain must have the same Drinfeld modules category:

```
sage: rho = DrinfeldModule(A, [Frac(A)(T), 1])
sage: Hom(phi, rho)
Traceback (most recent call last):
...
ValueError: Drinfeld modules must be in the same category
```

```
sage: sigma = DrinfeldModule(A, [1, z6, 2])
sage: Hom(phi, sigma)
Traceback (most recent call last):
...
ValueError: Drinfeld modules must be in the same category
```

One can create morphism objects by calling the homset:

```
sage: identity_morphism = E(1)
sage: identity_morphism
Identity morphism of Drinfeld module defined by T |--> 2*t^2 + z6*t + z6
```

```
sage: t = phi.ore_polring().gen()
sage: frobenius_endomorphism = E(t^6)
sage: frobenius_endomorphism
Endomorphism of Drinfeld module defined by T |--> 2*t^2 + z6*t + z6
  Defn: t^6
```

```
sage: isogeny = H(t + 1)
sage: isogeny
Drinfeld Module morphism:
  From: Drinfeld module defined by T |--> 2*t^2 + z6*t + z6
  To:   Drinfeld module defined by T |--> 2*t^2 + (2*z6^5 + 2*z6^4 + 2*z6 + 1)*t +␣
↪z6
  Defn: t + 1
```

And one can test if an Ore polynomial defines a morphism using the `in` syntax:

```
sage: 1 in H
False
sage: t^6 in H
False
sage: t + 1 in H
True
sage: 1 in E
True
sage: t^6 in E
```

(continues on next page)

(continued from previous page)

```
True
sage: t + 1 in E
False
```

This also works if the candidate is a morphism object:

```
sage: isogeny in H
True
sage: E(0) in E
True
sage: identity_morphism in H
False
sage: frobenius_endomorphism in H
False
```

Element

alias of *DrinfeldModuleMorphism*

class sage.rings.function_field.drinfeld_modules.homset.**DrinfeldModuleMorphismAction**(*A*, *H*, *is_left*, *op*)

Bases: Action

Action of the function ring on the homset of a Drinfeld module.

EXAMPLES:

```
sage: Fq = GF(5)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(3)
sage: phi = DrinfeldModule(A, [z, 1, z])
sage: psi = DrinfeldModule(A, [z, z^2 + 4*z + 3, 2*z^2 + 4*z + 4])
sage: H = Hom(phi, psi)
sage: t = phi.ore_variable()
sage: f = H(t + 2)
```

Left action:

```
sage: (T + 1) * f
Drinfeld Module morphism:
  From: Drinfeld module defined by T |--> z*t^2 + t + z
  To:   Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^2 + (z^2 + 4*z + 3)*t␣
↪+ z
  Defn: (2*z^2 + 4*z + 4)*t^3 + (2*z + 1)*t^2 + (2*z^2 + 4*z + 2)*t + 2*z + 2
```

Right action currently does not work (it is a known bug, due to an incompatibility between multiplication of morphisms and the coercion system):

```
sage: f * (T + 1)
Traceback (most recent call last):
...
TypeError: right (=T + 1) must be a map to multiply it by Drinfeld Module morphism:
  From: Drinfeld module defined by T |--> z*t^2 + t + z
  To:   Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^2 + (z^2 + 4*z + 3)*t␣
```

(continues on next page)

(continued from previous page)

```
↪+ z
  Defn: t + 2
```

CHAPTER

THREE

THE MODULE ACTION INDUCED BY A DRINFELD MODULE

3.1 The module action induced by a Drinfeld module

This module provides the class `sage.rings.function_field.drinfeld_module.action.DrinfeldModuleAction`.

AUTHORS:

- Antoine Leudière (2022-04)

class `sage.rings.function_field.drinfeld_modules.action.`**`DrinfeldModuleAction`**(*drinfeld_module*)

Bases: `Action`

This class implements the module action induced by a Drinfeld $\mathbb{F}_q[T]$-module.

Let ϕ be a Drinfeld $\mathbb{F}_q[T]$-module over a field K and let L/K be a field extension. Let $x \in L$ and let a be a function ring element; the action is defined as $(a, x) \mapsto \phi_a(x)$.

Note: In this implementation, L is K.

Note: The user should never explicitly instantiate the class $DrinfeldModuleAction$.

Warning: This class may be replaced later on. See issues #34833 and #34834.

INPUT: the Drinfeld module

EXAMPLES:

```
sage: Fq.<z2> = GF(11)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(2)
sage: phi = DrinfeldModule(A, [z, 0, 0, 1])
sage: action = phi.action()
sage: action
Action on Finite Field in z of size 11^2 over its base
 induced by Drinfeld module defined by T |--> t^3 + z
```

The action on elements is computed as follows:

```
sage: P = T + 1
sage: a = z
sage: action(P, a)
...
4*z + 2
sage: action(0, K.random_element())
0
sage: action(A.random_element(), 0)
0
```

Finally, given a Drinfeld module action, it is easy to recover the corresponding Drinfeld module:

```
sage: action.drinfeld_module() is phi
True
```

drinfeld_module()

Return the Drinfeld module defining the action.

OUTPUT: a Drinfeld module

EXAMPLES:

```
sage: Fq.<z2> = GF(11)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(2)
sage: phi = DrinfeldModule(A, [z, 0, 0, 1])
sage: action = phi.action()
sage: action.drinfeld_module() is phi
True
```

CHAPTER

FOUR

THE CATEGORY OF DRINFELD MODULES

4.1 Drinfeld modules over a base

This module provides the class `sage.category.drinfeld_modules.DrinfeldModules`.

AUTHORS:

- Antoine Leudière (2022-04)
- Xavier Caruso (2022-06)

class `sage.categories.drinfeld_modules.`**`DrinfeldModules`**(*base_field*, *name='t'*)

Bases: `Category_over_base_ring`

This class implements the category of Drinfeld $\mathbb{F}_q[T]$-modules on a given base field.

Let $\mathbb{F}_q[T]$ be a polynomial ring with coefficients in a finite field $\mathbb{F}_q$ and let K be a field. Fix a ring morphism $\gamma : \mathbb{F}_q[T] \to K$; we say that K is an $\mathbb{F}_q[T]`*-field*.Let`K\{\tau\}$ be the ring of Ore polynomials with coefficients in K, whose multiplication is given by the rule $\tau\lambda = \lambda^q\tau$ for any $\lambda \in K$.

The extension $K/\mathbb{F}_q[T]$ (represented as an instance of the class `sage.rings.ring_extension.RingExtension`) is the *base field* of the category; its defining morphism γ is called the *base morphism*.

The monic polynomial that generates the kernel of γ is called the $\mathbb{F}_q[T]$-*characteristic*, or *function-field characteristic*, of the base field. We say that $\mathbb{F}_q[T]$ is the *function ring* of the category; $K\{\tau\}$ is the *Ore polynomial ring*. The constant coefficient of the category is the image of T under the base morphism.

Construction

Generally, Drinfeld modules objects are created before their category, and the category is retrieved as an attribute of the Drinfeld module:

```
sage: Fq = GF(11)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(4)
sage: p_root = z^3 + 7*z^2 + 6*z + 10
sage: phi = DrinfeldModule(A, [p_root, 0, 0, 1])
sage: C = phi.category()
sage: C
Category of Drinfeld modules over Finite Field in z of size 11^4 over its base
```

The output tells the user that the category is only defined by its base.

Properties of the category

The base field is retrieved using the method `base()`.

sage: C.base() Finite Field in z of size 11^4 over its base

Equivalently, one can use `base_morphism()` to retrieve the base morphism:

```
sage: C.base_morphism()
Ring morphism:
  From: Univariate Polynomial Ring in T over Finite Field of size 11
  To:   Finite Field in z of size 11^4 over its base
  Defn: T |--> z^3 + 7*z^2 + 6*z + 10
```

The so-called constant coefficient — which is the same for all Drinfeld modules in the category — is simply the image of T by the base morphism:

```
sage: C.constant_coefficient()
z^3 + 7*z^2 + 6*z + 10
sage: C.base_morphism()(T) == C.constant_coefficient()
True
```

Similarly, the function ring-characteristic of the category is either 0 or the unique monic polynomial in $\mathbb{F}_q[T]$ that generates the kernel of the base:

```
sage: C.characteristic()
T^2 + 7*T + 2
sage: C.base_morphism()(C.characteristic())
0
```

The base field, base morphism, function ring and Ore polynomial ring are the same for the category and its objects:

```
sage: C.base() is phi.base()
True
sage: C.base_morphism() is phi.base_morphism()
True

sage: C.function_ring()
Univariate Polynomial Ring in T over Finite Field of size 11
sage: C.function_ring() is phi.function_ring()
True

sage: C.ore_polring()
Ore Polynomial Ring in t over Finite Field in z of size 11^4 over its base twisted␣
↪by Frob
sage: C.ore_polring() is phi.ore_polring()
True
```

Creating Drinfeld module objects from the category

Calling *object()* with an Ore polynomial creates a Drinfeld module object in the category whose generator is the input:

```
sage: psi = C.object([p_root, 1])
sage: psi
Drinfeld module defined by T |--> t + z^3 + 7*z^2 + 6*z + 10
sage: psi.category() is C
True
```

Of course, the constant coefficient of the input must be the same as the category:

```
sage: C.object([z, 1])
Traceback (most recent call last):
...
ValueError: constant coefficient must equal that of the category
```

It is also possible to create a random object in the category. The input is the desired rank:

```
sage: rho = C.random_object(2)
sage: rho  # random
Drinfeld module defined by T |--> (7*z^3 + 7*z^2 + 10*z + 2)*t^2 + (9*z^3 + 5*z^2 +␣
↪2*z + 7)*t + z^3 + 7*z^2 + 6*z + 10
sage: rho.rank() == 2
True
sage: rho.category() is C
True
```

Endsets()

Return the category of endsets.

EXAMPLES:

```
sage: Fq = GF(11)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(4)
sage: p_root = z^3 + 7*z^2 + 6*z + 10
sage: phi = DrinfeldModule(A, [p_root, 0, 0, 1])
sage: C = phi.category()

sage: from sage.categories.homsets import Homsets
sage: C.Endsets() is Homsets().Endsets()
True
```

Homsets()

Return the category of homsets.

EXAMPLES:

```
sage: Fq = GF(11)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(4)
sage: p_root = z^3 + 7*z^2 + 6*z + 10
sage: phi = DrinfeldModule(A, [p_root, 0, 0, 1])
```

(continues on next page)

(continued from previous page)

```
sage: C = phi.category()

sage: from sage.categories.homsets import Homsets
sage: C.Homsets() is Homsets()
True
```

class ParentMethods

Bases: `object`

base()

Return the base field of this Drinfeld module, viewed as an algebra over the function ring.

This is an instance of the class `sage.rings.ring_extension.RingExtension`.

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 +␣
↪2*z12^4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: phi.base()
Finite Field in z12 of size 5^12 over its base
```

The base can be infinite:

```
sage: sigma = DrinfeldModule(A, [Frac(A).gen(), 1])
sage: sigma.base()
Fraction Field of Univariate Polynomial Ring in T over Finite Field in z2␣
↪of size 5^2 over its base
```

base_morphism()

Return the base morphism of this Drinfeld module.

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 +␣
↪2*z12^4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: phi.base_morphism()
Ring morphism:
  From: Univariate Polynomial Ring in T over Finite Field in z2 of size 5^2
  To:   Finite Field in z12 of size 5^12 over its base
  Defn: T |--> 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 +␣
↪2*z12^4 + 3*z12^3 + z12^2 + 2*z12
```

The base field can be infinite:

```
sage: sigma = DrinfeldModule(A, [Frac(A).gen(), 1])
sage: sigma.base_morphism()
```

(continues on next page)

(continued from previous page)

```
Ring morphism:
  From: Univariate Polynomial Ring in T over Finite Field in z2 of size 5^2
  To:   Fraction Field of Univariate Polynomial Ring in T over Finite Field␣
↪in z2 of size 5^2 over its base
  Defn: T |--> T
```

base_over_constants_field()

Return the base field, seen as an extension over the constants field $\mathbb{F}_q$.

This is an instance of the class `sage.rings.ring_extension.RingExtension`.

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 +␣
↪2*z12^4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: phi.base_over_constants_field()
Field in z12 with defining polynomial x^6 + (4*z2 + 3)*x^5 + x^4 + (3*z2 +␣
↪1)*x^3 + x^2 + (4*z2 + 1)*x + z2 over its base
```

characteristic()

Return the function ring-characteristic.

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 +␣
↪2*z12^4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: phi.characteristic()
T^2 + (4*z2 + 2)*T + 2
sage: phi.base_morphism()(phi.characteristic())
0
```

```
sage: B.<Y> = Fq[]
sage: L = Frac(B)
sage: psi = DrinfeldModule(A, [L(1), 0, 0, L(1)])
sage: psi.characteristic()
Traceback (most recent call last):
...
NotImplementedError: function ring characteristic not implemented in this␣
↪case
```

constant_coefficient()

Return the constant coefficient of the generator of this Drinfeld module.

OUTPUT: an element in the base field

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 +␣
↪2*z12^4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: phi.constant_coefficient() == p_root
True
```

Let $\mathbb{F}_q[T]$ be the function ring, and let γ be the base of the Drinfeld module. The constant coefficient is $\gamma(T)$:

```
sage: C = phi.category()
sage: base = C.base()
sage: base(T) == phi.constant_coefficient()
True
```

Naturally, two Drinfeld modules in the same category have the same constant coefficient:

```
sage: t = phi.ore_polring().gen()
sage: psi = C.object(phi.constant_coefficient() + t^3)
sage: psi
Drinfeld module defined by T |--> t^3 + 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^
↪8 + z12^7 + 2*z12^5 + 2*z12^4 + 3*z12^3 + z12^2 + 2*z12
```

Reciprocally, it is impossible to create two Drinfeld modules in this category if they do not share the same constant coefficient:

```
sage: rho = C.object(phi.constant_coefficient() + 1 + t^3)
Traceback (most recent call last):
...
ValueError: constant coefficient must equal that of the category
```

function_ring()

Return the function ring of this Drinfeld module.

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 +␣
↪2*z12^4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: phi.function_ring()
Univariate Polynomial Ring in T over Finite Field in z2 of size 5^2
sage: phi.function_ring() is A
True
```

ore_polring()

Return the Ore polynomial ring of this Drinfeld module.

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 +␣
↪2*z12^4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])
sage: S = phi.ore_polring()
sage: S
Ore Polynomial Ring in t over Finite Field in z12 of size 5^12 over its␣
↪base twisted by Frob^2
```

The Ore polynomial ring can also be retrieved from the category of the Drinfeld module:

```
sage: S is phi.category().ore_polring()
True
```

The generator of the Drinfeld module is in the Ore polynomial ring:

```
sage: phi(T) in S
True
```

ore_variable()

Return the variable of the Ore polynomial ring of this Drinfeld module.

EXAMPLES:

```
sage: Fq = GF(25)
sage: A.<T> = Fq[]
sage: K.<z12> = Fq.extension(6)
sage: p_root = 2*z12^11 + 2*z12^10 + z12^9 + 3*z12^8 + z12^7 + 2*z12^5 +␣
↪2*z12^4 + 3*z12^3 + z12^2 + 2*z12
sage: phi = DrinfeldModule(A, [p_root, z12^3, z12^5])

sage: phi.ore_polring()
Ore Polynomial Ring in t over Finite Field in z12 of size 5^12 over its␣
↪base twisted by Frob^2
sage: phi.ore_variable()
t
```

base_morphism()

Return the base morphism of the category.

EXAMPLES:

```
sage: Fq = GF(11)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(4)
sage: p_root = z^3 + 7*z^2 + 6*z + 10
sage: phi = DrinfeldModule(A, [p_root, 0, 0, 1])
sage: C = phi.category()
sage: C.base_morphism()
Ring morphism:
  From: Univariate Polynomial Ring in T over Finite Field of size 11
  To:   Finite Field in z of size 11^4 over its base
```

(continues on next page)

(continued from previous page)

```
  Defn: T |--> z^3 + 7*z^2 + 6*z + 10

sage: C.constant_coefficient() == C.base_morphism()(T)
True
```

base_over_constants_field()

Return the base field, seen as an extension over the constants field $\mathbb{F}_q$.

EXAMPLES:

```
sage: Fq = GF(11)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(4)
sage: p_root = z^3 + 7*z^2 + 6*z + 10
sage: phi = DrinfeldModule(A, [p_root, 0, 0, 1])
sage: C = phi.category()
sage: C.base_over_constants_field()
Field in z with defining polynomial x^4 + 8*x^2 + 10*x + 2 over its base
```

characteristic()

Return the function ring-characteristic.

EXAMPLES:

```
sage: Fq = GF(11)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(4)
sage: p_root = z^3 + 7*z^2 + 6*z + 10
sage: phi = DrinfeldModule(A, [p_root, 0, 0, 1])
sage: C = phi.category()
sage: C.characteristic()
T^2 + 7*T + 2
```

```
sage: psi = DrinfeldModule(A, [Frac(A).gen(), 1])
sage: C = psi.category()
sage: C.characteristic()
0
```

constant_coefficient()

Return the constant coefficient of the category.

EXAMPLES:

```
sage: Fq = GF(11)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(4)
sage: p_root = z^3 + 7*z^2 + 6*z + 10
sage: phi = DrinfeldModule(A, [p_root, 0, 0, 1])
sage: C = phi.category()
sage: C.constant_coefficient()
z^3 + 7*z^2 + 6*z + 10
sage: C.constant_coefficient() == C.base()(T)
True
```

function_ring()

Return the function ring of the category.

EXAMPLES:

```
sage: Fq = GF(11)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(4)
sage: p_root = z^3 + 7*z^2 + 6*z + 10
sage: phi = DrinfeldModule(A, [p_root, 0, 0, 1])
sage: C = phi.category()
sage: C.function_ring()
Univariate Polynomial Ring in T over Finite Field of size 11
sage: C.function_ring() is A
True
```

object(*gen*)

Return a Drinfeld module object in the category whose generator is the input.

INPUT:

- gen – the generator of the Drinfeld module, given as an Ore polynomial or a list of coefficients

EXAMPLES:

```
sage: Fq = GF(11)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(4)
sage: p_root = z^3 + 7*z^2 + 6*z + 10
sage: psi = DrinfeldModule(A, [p_root, 1])
sage: C = psi.category()

sage: phi = C.object([p_root, 0, 1])
sage: phi
Drinfeld module defined by T |--> t^2 + z^3 + 7*z^2 + 6*z + 10
sage: t = phi.ore_polring().gen()
sage: C.object(t^2 + z^3 + 7*z^2 + 6*z + 10) is phi
True
```

ore_polring()

Return the Ore polynomial ring of the category

EXAMPLES:

```
sage: Fq = GF(11)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(4)
sage: p_root = z^3 + 7*z^2 + 6*z + 10
sage: phi = DrinfeldModule(A, [p_root, 0, 0, 1])
sage: C = phi.category()
sage: C.ore_polring()
Ore Polynomial Ring in t over Finite Field in z of size 11^4 over its base␣
↪twisted by Frob
```

random_object(*rank*)

Return a random Drinfeld module in the category with given rank.

INPUT:

- rank – an integer, the rank of the Drinfeld module

EXAMPLES:

```
sage: Fq = GF(11)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(4)
sage: p_root = z^3 + 7*z^2 + 6*z + 10
sage: phi = DrinfeldModule(A, [p_root, 0, 0, 1])
sage: C = phi.category()

sage: psi = C.random_object(3)  # random
Drinfeld module defined by T |--> (6*z^3 + 4*z^2 + 10*z + 9)*t^3 + (4*z^3 + 8*z^
↪2 + 8*z)*t^2 + (10*z^3 + 3*z^2 + 6*z)*t + z^3 + 7*z^2 + 6*z + 10
sage: psi.rank() == 3
True
```

super_categories()

EXAMPLES:

```
sage: Fq = GF(11)
sage: A.<T> = Fq[]
sage: K.<z> = Fq.extension(4)
sage: p_root = z^3 + 7*z^2 + 6*z + 10
sage: phi = DrinfeldModule(A, [p_root, 0, 0, 1])
sage: C = phi.category()
sage: C.super_categories()
[Category of objects]
```

CHAPTER

FIVE

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

INDEX

V