Sage Reference Manual: Finite Rings

Release 6.7

The Sage Development Team

June 24, 2015

CHAPTER

ONE

ROUTINES FOR CONWAY AND PSEUDO-CONWAY POLYNOMIALS.

AUTHORS:

- David Roe
- Jean-Pierre Flori
- Peter Bruin

class `sage.rings.finite_rings.conway_polynomials.`**`PseudoConwayLattice`**(*p*, *use_database=True*)

Bases: `sage.structure.sage_object.SageObject`

A pseudo-Conway lattice over a given finite prime field.

The Conway polynomial f_n of degree n over $\mathbf{F}_p$ is defined by the following four conditions:

- f_n is irreducible.
- In the quotient field $\mathbf{F}_p[x]/(f_n)$, the element $x \bmod f_n$ generates the multiplicative group.
- The minimal polynomial of $(x \bmod f_n)^{\frac{p^n-1}{p^m-1}}$ equals the Conway polynomial f_m, for every divisor m of n.
- f_n is lexicographically least among all such polynomials, under a certain ordering.

The final condition is needed only in order to make the Conway polynomial unique. We define a pseudo-Conway lattice to be any family of polynomials, indexed by the positive integers, satisfying the first three conditions.

INPUT:

- `p` – prime number
- `use_database` – boolean. If `True`, use actual Conway polynomials whenever they are available in the database. If `False`, always compute pseudo-Conway polynomials.

EXAMPLES:

```
sage: from sage.rings.finite_rings.conway_polynomials import PseudoConwayLattice
sage: PCL = PseudoConwayLattice(2, use_database=False)
sage: PCL.polynomial(3)
x^3 + x + 1
```

`check_consistency`(*n*)

Check that the pseudo-Conway polynomials of degree dividing n in this lattice satisfy the required compatibility conditions.

EXAMPLES:

```
sage: from sage.rings.finite_rings.conway_polynomials import PseudoConwayLattice
sage: PCL = PseudoConwayLattice(2, use_database=False)
```

```
sage: PCL.check_consistency(6)
sage: PCL.check_consistency(60)  # long
```

polynomial(*n*)

Return the pseudo-Conway polynomial of degree n in this lattice.

INPUT:

- `n` – positive integer

OUTPUT:

- a pseudo-Conway polynomial of degree n for the prime p.

ALGORITHM:

Uses an algorithm described in [HL99], modified to find pseudo-Conway polynomials rather than Conway polynomials. The major difference is that we stop as soon as we find a primitive polynomial.

REFERENCE:

EXAMPLES:

```
sage: from sage.rings.finite_rings.conway_polynomials import PseudoConwayLattice
sage: PCL = PseudoConwayLattice(2, use_database=False)
sage: PCL.polynomial(3)
x^3 + x + 1
sage: PCL.polynomial(4)
x^4 + x^3 + 1
sage: PCL.polynomial(60)
x^60 + x^59 + x^58 + x^55 + x^54 + x^53 + x^52 + x^51 + x^48 + x^46 + x^45 + x^42 + x^41 + x
```

`sage.rings.finite_rings.conway_polynomials.`**conway_polynomial**(*p*, *n*)

Return the Conway polynomial of degree n over `GF(p)`.

If the requested polynomial is not known, this function raises a `RuntimeError` exception.

INPUT:

- `p` – prime number
- `n` – positive integer

OUTPUT:

- the Conway polynomial of degree n over the finite field `GF(p)`, loaded from a table.

Note: The first time this function is called a table is read from disk, which takes a fraction of a second. Subsequent calls do not require reloading the table.

See also the `ConwayPolynomials()` object, which is the table of Conway polynomials used by this function.

EXAMPLES:

```
sage: conway_polynomial(2,5)
x^5 + x^2 + 1
sage: conway_polynomial(101,5)
x^5 + 2*x + 99
sage: conway_polynomial(97,101)
Traceback (most recent call last):
...
RuntimeError: requested Conway polynomial not in database.
```

sage.rings.finite_rings.conway_polynomials.**exists_conway_polynomial**(*p*, *n*)

Check whether the Conway polynomial of degree n over `GF(p)` is known.

INPUT:

- `p` – prime number
- `n` – positive integer

OUTPUT:

- boolean: `True` if the Conway polynomial of degree n over `GF(p)` is in the database, `False` otherwise.

If the Conway polynomial is in the database, it can be obtained using the command `conway_polynomial(p,n)`.

EXAMPLES:

```
sage: exists_conway_polynomial(2,3)
True
sage: exists_conway_polynomial(2,-1)
False
sage: exists_conway_polynomial(97,200)
False
sage: exists_conway_polynomial(6,6)
False
```

CHAPTER

TWO

GIVARO FIELD ELEMENTS

Sage includes the Givaro finite field library, for highly optimized arithmetic in finite fields.

Note: The arithmetic is performed by the Givaro C++ library which uses Zech logs internally to represent finite field elements. This implementation is the default finite extension field implementation in Sage for the cardinality less than 2^{16}, as it is a lot faster than the PARI implementation. Some functionality in this class however is implemented using PARI.

EXAMPLES:

```
sage: k = GF(5); type(k)
<class 'sage.rings.finite_rings.finite_field_prime_modn.FiniteField_prime_modn_with_category'>
sage: k = GF(5^2,'c'); type(k)
<class 'sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro_with_category'>
sage: k = GF(2^16,'c'); type(k)
<class 'sage.rings.finite_rings.finite_field_ntl_gf2e.FiniteField_ntl_gf2e_with_category'>
sage: k = GF(3^16,'c'); type(k)
<class 'sage.rings.finite_rings.finite_field_pari_ffelt.FiniteField_pari_ffelt_with_category'>

sage: n = previous_prime_power(2^16 - 1)
sage: while is_prime(n):
...     n = previous_prime_power(n)
sage: factor(n)
251^2
sage: k = GF(n,'c'); type(k)
<class 'sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro_with_category'>
```

AUTHORS:

- Martin Albrecht <malb@informatik.uni-bremen.de> (2006-06-05)
- William Stein (2006-12-07): editing, lots of docs, etc.
- Robert Bradshaw (2007-05-23): is_square/sqrt, pow.

class `sage.rings.finite_rings.element_givaro.`**`Cache_givaro`**

Bases: `sage.structure.sage_object.SageObject`

Finite Field.

These are implemented using Zech logs and the cardinality must be less than 2^{16}. By default conway polynomials are used as minimal polynomial.

INPUT:

- `q` – p^n (must be prime power)
- `name` – variable used for poly_repr (default: `'a'`)

- `modulus` – a polynomial to use as modulus.
- `repr` – (default: 'poly') controls the way elements are printed to the user:
  - 'log': repr is `log_repr()`
  - 'int': repr is `int_repr()`
  - 'poly': repr is `poly_repr()`
- `cache` – (default: `False`) if `True` a cache of all elements of this field is created. Thus, arithmetic does not create new elements which speeds calculations up. Also, if many elements are needed during a calculation this cache reduces the memory requirement as at most `order()` elements are created.

OUTPUT:

Givaro finite field with characteristic p and cardinality p^n.

EXAMPLES:

By default Conway polynomials are used:

```
sage: k.<a> = GF(2**8)
sage: -a ^ k.degree()
a^4 + a^3 + a^2 + 1
sage: f = k.modulus(); f
x^8 + x^4 + x^3 + x^2 + 1
```

You may enforce a modulus:

```
sage: P.<x> = PolynomialRing(GF(2))
sage: f = x^8 + x^4 + x^3 + x + 1 # Rijndael polynomial
sage: k.<a> = GF(2^8, modulus=f)
sage: k.modulus()
x^8 + x^4 + x^3 + x + 1
sage: a^(2^8)
a
```

You may enforce a random modulus:

```
sage: k = GF(3**5, 'a', modulus='random')
sage: k.modulus() # random polynomial
x^5 + 2*x^4 + 2*x^3 + x^2 + 2
```

For binary fields, you may ask for a minimal weight polynomial:

```
sage: k = GF(2**10, 'a', modulus='minimal_weight')
sage: k.modulus()
x^10 + x^3 + 1
```

a_times_b_minus_c(*a*, *b*, *c*)

Return `a*b - c`.

INPUT:

- `a,b,c` – `FiniteField_givaroElement`

EXAMPLES:

```
sage: k.<a> = GF(3**3)
sage: k._cache.a_times_b_minus_c(a,a,k(1))
a^2 + 2
```

a_times_b_plus_c(*a*, *b*, *c*)

Return `a*b + c`. This is faster than multiplying `a` and `b` first and adding `c` to the result.

INPUT:

- a,b,c – FiniteField_givaroElement

EXAMPLES:

```
sage: k.<a> = GF(2**8)
sage: k._cache.a_times_b_plus_c(a,a,k(1))
a^2 + 1
```

c_minus_a_times_b(*a*, *b*, *c*)

Return c - a*b.

INPUT:

- a,b,c – FiniteField_givaroElement

EXAMPLES:

```
sage: k.<a> = GF(3**3)
sage: k._cache.c_minus_a_times_b(a,a,k(1))
2*a^2 + 1
```

characteristic()

Return the characteristic of this field.

EXAMPLES:

```
sage: p = GF(19^3,'a')._cache.characteristic(); p
19
```

element_from_data(*e*)

Coerces several data types to self.

INPUT:

- e – data to coerce in.

EXAMPLES:

```
sage: k = GF(3^8, 'a')
sage: type(k)
<class 'sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro_with_category'>
sage: e = k.vector_space().gen(1); e
(0, 1, 0, 0, 0, 0, 0, 0)
sage: k(e) #indirect doctest
a
```

TESTS:

Check coercion of large integers:

```
sage: k(-5^13)
1
sage: k(2^31)
2
sage: k(int(10^19))
1
sage: k(2^63)
2
sage: k(2^100)
1
sage: k(int(2^100))
1
```

```
sage: k(long(2^100))
1
sage: k(-2^100)
2
```

For more examples, see `finite_field_givaro.FiniteField_givaro._element_constructor_`

exponent()

Returns the degree of this field over $\mathbf{F}_p$.

EXAMPLES:

```
sage: K.<a> = GF(9); K._cache.exponent()
2
```

fetch_int(*n*)

Given an integer `n` return a finite field element in `self` which equals `n` under the condition that `gen()` is set to `characteristic()`.

EXAMPLES:

```
sage: k.<a> = GF(2^8)
sage: k._cache.fetch_int(8)
a^3
sage: e = k._cache.fetch_int(151); e
a^7 + a^4 + a^2 + a + 1
sage: 2^7 + 2^4 + 2^2 + 2 + 1
151
```

gen()

Returns a generator of the field.

EXAMPLES:

```
sage: K.<a> = GF(625)
sage: K._cache.gen()
a
```

int_to_log(*n*)

Given an integer n this method returns i where i satisfies $g^i = n \mod p$ where g is the generator and p is the characteristic of `self`.

INPUT:

- `n` – integer representation of an finite field element

OUTPUT:

log representation of `n`

EXAMPLES:

```
sage: k = GF(7**3, 'a')
sage: k._cache.int_to_log(4)
228
sage: k._cache.int_to_log(3)
57
sage: k.gen()^57
3
```

log_to_int(*n*)

Given an integer n this method returns i where i satisfies $g^n = i$ where g is the generator of `self`; the result is interpreted as an integer.

INPUT:

•n – log representation of a finite field element

OUTPUT:

integer representation of a finite field element.

EXAMPLES:

```
sage: k = GF(2**8, 'a')
sage: k._cache.log_to_int(4)
16
sage: k._cache.log_to_int(20)
180
```

order()

Returns the order of this field.

EXAMPLES:

```
sage: K.<a> = GF(9)
sage: K._cache.order()
9
```

order_c()

Returns the order of this field.

EXAMPLES:

```
sage: K.<a> = GF(9)
sage: K._cache.order_c()
9
```

random_element(**args*, ***kwds*)

Return a random element of self.

EXAMPLES:

```
sage: k = GF(23**3, 'a')
sage: e = k._cache.random_element(); e
2*a^2 + 14*a + 21
sage: type(e)
<type 'sage.rings.finite_rings.element_givaro.FiniteField_givaroElement'>

sage: P.<x> = PowerSeriesRing(GF(3^3, 'a'))
sage: P.random_element(5)
2*a + 2 + (a^2 + a + 2)*x + (2*a + 1)*x^2 + (2*a^2 + a)*x^3 + 2*a^2*x^4 + O(x^5)
```

repr

class sage.rings.finite_rings.element_givaro.**FiniteField_givaroElement**

Bases: sage.rings.finite_rings.element_base.FinitePolyExtElement

An element of a (Givaro) finite field.

int_repr(**args*, ***kwds*)

Deprecated: Use _int_repr() instead. See trac ticket #11295 for details.

integer_representation()

Return the integer representation of self. When self is in the prime subfield, the integer returned is equal to self.

Elements of this field are represented as integers as follows: given the element $e \in \mathbf{F}_p[x]$ with $e = a_0 + a_1 x + a_2 x^2 + \cdots$, the integer representation is $a_0 + a_1 p + a_2 p^2 + \cdots$.

OUTPUT: A Python `int`.

EXAMPLES:

```
sage: k.<b> = GF(5^2); k
Finite Field in b of size 5^2
sage: k(4).integer_representation()
4
sage: b.integer_representation()
5
sage: type(b.integer_representation())
<type 'int'>
```

is_one()

Return `True` if `self == k(1)`.

EXAMPLES:

```
sage: k.<a> = GF(3^4); k
Finite Field in a of size 3^4
sage: a.is_one()
False
sage: k(1).is_one()
True
```

is_square()

Return `True` if `self` is a square in `self.parent()`

ALGORITHM:

Elements are stored as powers of generators, so we simply check to see if it is an even power of a generator.

EXAMPLES:

```
sage: k.<a> = GF(9); k
Finite Field in a of size 3^2
sage: a.is_square()
False
sage: v = set([x^2 for x in k])
sage: [x.is_square() for x in v]
[True, True, True, True, True]
sage: [x.is_square() for x in k if not x in v]
[False, False, False, False]
```

TESTS:

```
sage: K = GF(27, 'a')
sage: set([a*a for a in K]) == set([a for a in K if a.is_square()])
True
sage: K = GF(25, 'a')
sage: set([a*a for a in K]) == set([a for a in K if a.is_square()])
True
sage: K = GF(16, 'a')
sage: set([a*a for a in K]) == set([a for a in K if a.is_square()])
True
```

is_unit()

Return `True` if self is nonzero, so it is a unit as an element of the finite field.

EXAMPLES:

```
sage: k.<a> = GF(3^4); k
Finite Field in a of size 3^4
sage: a.is_unit()
True
sage: k(0).is_unit()
False
```

log (*base*)

Return the log to the base b of `self`, i.e., an integer n such that $b^n =$ `self`.

> **Warning:** TODO – This is currently implemented by solving the discrete log problem – which shouldn't be needed because of how finite field elements are represented.

EXAMPLES:

```
sage: k.<b> = GF(5^2); k
Finite Field in b of size 5^2
sage: a = b^7
sage: a.log(b)
7
```

log_repr (**args*, ***kwds*)

Deprecated: Use `_log_repr()` instead. See trac ticket #11295 for details.

log_to_int (**args*, ***kwds*)

Deprecated: Use `_log_to_int()` instead. See trac ticket #11295 for details.

multiplicative_order ()

Return the multiplicative order of this field element.

EXAMPLES:

```
sage: S.<b> = GF(5^2); S
Finite Field in b of size 5^2
sage: b.multiplicative_order()
24
sage: (b^6).multiplicative_order()
4
```

poly_repr (**args*, ***kwds*)

Deprecated: Use `_poly_repr()` instead. See trac ticket #11295 for details.

polynomial (*name=None*)

Return self viewed as a polynomial over `self.parent().prime_subfield()`.

EXAMPLES:

```
sage: k.<b> = GF(5^2); k
Finite Field in b of size 5^2
sage: f = (b^2+1).polynomial(); f
b + 4
sage: type(f)
<type 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>
sage: parent(f)
Univariate Polynomial Ring in b over Finite Field of size 5
```

sqrt (*extend=False*, *all=False*)

Return a square root of this finite field element in its parent, if there is one. Otherwise, raise a `ValueError`.

INPUT:

•`extend` – bool (default: `True`); if `True`, return a square root in an extension ring, if necessary. Otherwise, raise a `ValueError` if the root is not in the base ring.

> **Warning:** this option is not implemented!

•`all` – bool (default: `False`); if `True`, return all square roots of `self`, instead of just one.

> **Warning:** The `extend` option is not implemented (yet).

ALGORITHM:

`self` is stored as a^k for some generator a. Return $a^{k/2}$ for even k.

EXAMPLES:

```
sage: k.<a> = GF(7^2)
sage: k(2).sqrt()
3
sage: k(3).sqrt()
2*a + 6
sage: k(3).sqrt()**2
3
sage: k(4).sqrt()
2
sage: k.<a> = GF(7^3)
sage: k(3).sqrt()
Traceback (most recent call last):
...
ValueError: must be a perfect square.
```

TESTS:

```
sage: K = GF(49, 'a')
sage: all([a.sqrt()*a.sqrt() == a for a in K if a.is_square()])
True
sage: K = GF(27, 'a')
sage: all([a.sqrt()*a.sqrt() == a for a in K if a.is_square()])
True
sage: K = GF(8, 'a')
sage: all([a.sqrt()*a.sqrt() == a for a in K if a.is_square()])
True
sage: K.<a>=FiniteField(9)
sage: a.sqrt(extend = False, all = True)
[]
```

class `sage.rings.finite_rings.element_givaro.`**`FiniteField_givaro_iterator`**

Bases: `object`

Iterator over `FiniteField_givaro` elements. We iterate multiplicatively, as powers of a fixed internal generator.

EXAMPLES:

```
sage: for x in GF(2^2,'a'): print x
0
a
a + 1
1
```

next()

x.next() -> the next value, or raise StopIteration

`sage.rings.finite_rings.element_givaro.`**unpickle_Cache_givaro**(*parent*, *p*, *k*, *modulus*, *rep*, *cache*)

EXAMPLES:

```
sage: k = GF(3**7, 'a')
sage: loads(dumps(k)) == k # indirect doctest
True
```

`sage.rings.finite_rings.element_givaro.`**unpickle_FiniteField_givaroElement**(*parent*, *x*)

TESTS:

```
sage: k = GF(3**4, 'a')
sage: e = k.random_element()
sage: TestSuite(e).run() # indirect doctest
```

CHAPTER

THREE

FINITE FIELDS OF CHARACTERISTIC 2.

This implementation uses NTL's GF2E class to perform the arithmetic and is the standard implementation for `GF(2^n)` for `n >= 16`.

AUTHORS:

- Martin Albrecht <malb@informatik.uni-bremen.de> (2007-10)

class `sage.rings.finite_rings.element_ntl_gf2e.`**`Cache_ntl_gf2e`**

Bases: `sage.structure.sage_object.SageObject`

This class stores information for an NTL finite field in a Cython class so that elements can access it quickly.

It's modeled on `NativeIntStruct`, but includes many functions that were previously included in the parent (see trac ticket #12062).

`degree`()

If the field has cardinality 2^n this method returns n.

EXAMPLES:

```
sage: k.<a> = GF(2^64)
sage: k._cache.degree()
64
```

`fetch_int`(*number*)

Given an integer less than p^n with base 2 representation $a_0 + a_1 \cdot 2 + \cdots + a_k 2^k$, this returns $a_0 + a_1 x + \cdots + a_k x^k$, where x is the generator of this finite field.

INPUT:

- `number` – an integer, of size less than the cardinality

EXAMPLES:

```
sage: k.<a> = GF(2^48)
sage: k._cache.fetch_int(2^33 + 2 + 1)
a^33 + a + 1
```

TESTS:

We test that #17027 is fixed:

```
sage: K.<a> = GF(2^16)
sage: K._cache.fetch_int(0r)
0
```

`import_data`(*e*)

EXAMPLES:

```
sage: k.<a> = GF(2^17)
sage: V = k.vector_space()
sage: v = [1,0,0,0,0,1,0,0,1,0,0,0,0,1,0,0,0]
sage: k._cache.import_data(v)
a^13 + a^8 + a^5 + 1
sage: k._cache.import_data(V(v))
a^13 + a^8 + a^5 + 1
```

TESTS:

We check that trac ticket #12584 is fixed:

```
sage: k(2^63)
0
```

We can coerce from PARI finite field implementations:

```
sage: K.<a> = GF(2^19, impl="ntl")
sage: a^20
a^6 + a^3 + a^2 + a
sage: M.<c> = GF(2^19, impl="pari_ffelt")
sage: K(c^20)
a^6 + a^3 + a^2 + a
```

order()

Return the cardinality of the field.

EXAMPLES:

```
sage: k.<a> = GF(2^64)
sage: k._cache.order()
18446744073709551616
```

polynomial()

Returns the list of 0's and 1's giving the defining polynomial of the field.

EXAMPLES:

```
sage: k.<a> = GF(2^20,modulus="minimal_weight")
sage: k._cache.polynomial()
[1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

class `sage.rings.finite_rings.element_ntl_gf2e.`**FiniteField_ntl_gf2eElement**

Bases: `sage.rings.finite_rings.element_base.FinitePolyExtElement`

An element of an NTL:GF2E finite field.

charpoly(*var='x'*)

Return the characteristic polynomial of `self` as a polynomial in var over the prime subfield.

INPUT:

- `var` – string (default: `'x'`)

OUTPUT:

polynomial

EXAMPLES:

```
sage: k.<a> = GF(2^8, impl="ntl")
sage: b = a^3 + a
sage: b.minpoly()
```

```
x^4 + x^3 + x^2 + x + 1
sage: b.charpoly()
x^8 + x^6 + x^4 + x^2 + 1
sage: b.charpoly().factor()
(x^4 + x^3 + x^2 + x + 1)^2
sage: b.charpoly('Z')
Z^8 + Z^6 + Z^4 + Z^2 + 1
```

integer_representation()

Return the int representation of `self`. When `self` is in the prime subfield, the integer returned is equal to `self` and not to `log_repr`.

Elements of this field are represented as ints in as follows: for $e \in \mathbf{F}_p[x]$ with $e = a_0 + a_1 x + a_2 x^2 + \cdots$, e is represented as: $n = a_0 + a_1 p + a_2 p^2 + \cdots$.

EXAMPLES:

```
sage: k.<a> = GF(2^20)
sage: a.integer_representation()
2
sage: (a^2 + 1).integer_representation()
5
sage: k.<a> = GF(2^70)
sage: (a^65 + a^64 + 1).integer_representation()
55340232221128654849L
```

is_one()

Return `True` if `self == k(1)`.

Equivalent to `self != k(0)`.

EXAMPLES:

```
sage: k.<a> = GF(2^20)
sage: a.is_one() # indirect doctest
False
sage: k(1).is_one()
True
```

is_square()

Return `True` as every element in $\mathbf{F}_{2^n}$ is a square.

EXAMPLES:

```
sage: k.<a> = GF(2^18)
sage: e = k.random_element()
sage: e
a^15 + a^14 + a^13 + a^11 + a^10 + a^9 + a^6 + a^5 + a^4 + 1
sage: e.is_square()
True
sage: e.sqrt()
a^16 + a^15 + a^14 + a^11 + a^9 + a^8 + a^7 + a^6 + a^4 + a^3 + 1
sage: e.sqrt()^2 == e
True
```

is_unit()

Return `True` if `self` is nonzero, so it is a unit as an element of the finite field.

EXAMPLES:

```
sage: k.<a> = GF(2^17)
sage: a.is_unit()
True
sage: k(0).is_unit()
False
```

log (*base*)

Return x such that $b^x = a$, where x is a and b is the base.

INPUT:

•`base` – finite field element that generates the multiplicative group.

OUTPUT:

Integer x such that $a^x = b$, if it exists. Raises a `ValueError` exception if no such x exists.

EXAMPLES:

```
sage: F = GF(17)
sage: F(3^11).log(F(3))
11
sage: F = GF(113)
sage: F(3^19).log(F(3))
19
sage: F = GF(next_prime(10000))
sage: F(23^997).log(F(23))
997
```

```
sage: F = FiniteField(2^10, 'a')
sage: g = F.gen()
sage: b = g; a = g^37
sage: a.log(b)
37
sage: b^37; a
a^8 + a^7 + a^4 + a + 1
a^8 + a^7 + a^4 + a + 1
```

AUTHOR: David Joyner and William Stein (2005-11)

minpoly (*var='x'*)

Return the minimal polynomial of `self`, which is the smallest degree polynomial $f \in \mathbf{F}_2[x]$ such that `f(self) == 0`.

INPUT:

•`var` – string (default: `'x'`)

OUTPUT:

polynomial

EXAMPLES:

```
sage: K.<a> = GF(2^100)
sage: f = a.minpoly(); f
x^100 + x^57 + x^56 + x^55 + x^52 + x^48 + x^47 + x^46 + x^45 + x^44 + x^43 + x^41 + x^37 +
sage: f(a)
0
sage: g = K.random_element()
sage: g.minpoly()(g)
0
```

polynomial (*name=None*)

Return `self` viewed as a polynomial over `self.parent().prime_subfield()`.

INPUT:

- `name` – (optional) variable name

EXAMPLES:

```
sage: k.<a> = GF(2^17)
sage: e = a^15 + a^13 + a^11 + a^10 + a^9 + a^8 + a^7 + a^6 + a^4 + a + 1
sage: e.polynomial()
a^15 + a^13 + a^11 + a^10 + a^9 + a^8 + a^7 + a^6 + a^4 + a + 1

sage: from sage.rings.polynomial.polynomial_element import is_Polynomial
sage: is_Polynomial(e.polynomial())
True

sage: e.polynomial('x')
x^15 + x^13 + x^11 + x^10 + x^9 + x^8 + x^7 + x^6 + x^4 + x + 1
```

sqrt (*all=False*, *extend=False*)

Return a square root of this finite field element in its parent.

EXAMPLES:

```
sage: k.<a> = GF(2^20)
sage: a.is_square()
True
sage: a.sqrt()
a^19 + a^15 + a^14 + a^12 + a^9 + a^7 + a^4 + a^3 + a + 1
sage: a.sqrt()^2 == a
True
```

This failed before trac ticket #4899:

```
sage: GF(2^16,'a')(1).sqrt()
1
```

trace ()

Return the trace of `self`.

EXAMPLES:

```
sage: K.<a> = GF(2^25)
sage: a.trace()
0
sage: a.charpoly()
x^25 + x^8 + x^6 + x^2 + 1
sage: parent(a.trace())
Finite Field of size 2

sage: b = a+1
sage: b.trace()
1
sage: b.charpoly()[1]
1
```

weight ()

Returns the number of non-zero coefficients in the polynomial representation of `self`.

EXAMPLES:

```
sage: K.<a> = GF(2^21)
sage: a.weight()
1
sage: (a^5+a^2+1).weight()
3
sage: b = 1/(a+1); b
a^20 + a^19 + a^18 + a^17 + a^16 + a^15 + a^14 + a^13 + a^12 + a^11 + a^10 + a^9 + a^8 + a^7
sage: b.weight()
18
```

`sage.rings.finite_rings.element_ntl_gf2e.`**`unpickleFiniteField_ntl_gf2eElement`**(*parent*, *elem*)

EXAMPLES:

```
sage: k.<a> = GF(2^20)
sage: e = k.random_element()
sage: f = loads(dumps(e)) # indirect doctest
sage: e == f
True
```

CHAPTER

FOUR

FINITE FIELD ELEMENTS IMPLEMENTED VIA PARI'S FFELT TYPE

AUTHORS:

- Peter Bruin (June 2013): initial version, based on element_ext_pari.py by William Stein et al. and element_ntl_gf2e.pyx by Martin Albrecht.

class sage.rings.finite_rings.element_pari_ffelt.**FiniteFieldElement_pari_ffelt**

Bases: sage.rings.finite_rings.element_base.FinitePolyExtElement

An element of a finite field.

EXAMPLE:

```
sage: K = FiniteField(10007^10, 'a', impl='pari_ffelt')
sage: a = K.gen(); a
a
sage: type(a)
<type 'sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt'>
```

TESTS:

```
sage: n = 63
sage: m = 3;
sage: K.<a> = GF(2^n, impl='pari_ffelt')
sage: f = conway_polynomial(2, n)
sage: f(a) == 0
True
sage: e = (2^n - 1) / (2^m - 1)
sage: conway_polynomial(2, m)(a^e) == 0
True

sage: K.<a> = FiniteField(2^16, impl='pari_ffelt')
sage: K(0).is_zero()
True
sage: (a - a).is_zero()
True
sage: a - a
0
sage: a == a
True
sage: a - a == 0
True
sage: a - a == K(0)
True
sage: TestSuite(a).run()
```

Test creating elements from basic Python types:

```
sage: K.<a> = FiniteField(7^20, impl='pari_ffelt')
sage: K(int(8))
1
sage: K(long(-2^300))
6
```

charpoly(*var='x'*)

Return the characteristic polynomial of `self`.

INPUT:

- `var` – string (default: 'x'): variable name to use.

EXAMPLE:

```
sage: R.<x> = PolynomialRing(FiniteField(3))
sage: F.<a> = FiniteField(3^2, modulus=x^2 + 1)
sage: a.charpoly('y')
y^2 + 1
```

is_one()

Return `True` if `self` equals 1.

EXAMPLE:

```
sage: F.<a> = FiniteField(5^3, impl='pari_ffelt')
sage: a.is_one()
False
sage: (a/a).is_one()
True
```

is_square()

Return `True` if and only if `self` is a square in the finite field.

EXAMPLES:

```
sage: k.<a> = FiniteField(3^2, impl='pari_ffelt')
sage: a.is_square()
False
sage: (a**2).is_square()
True

sage: k.<a> = FiniteField(2^2, impl='pari_ffelt')
sage: (a**2).is_square()
True

sage: k.<a> = FiniteField(17^5, impl='pari_ffelt')
sage: (a**2).is_square()
True
sage: a.is_square()
False
sage: k(0).is_square()
True
```

is_unit()

Return `True` if `self` is non-zero.

EXAMPLE:

```
sage: F.<a> = FiniteField(5^3, impl='pari_ffelt')
sage: a.is_unit()
True
```

is_zero()

Return `True` if `self` equals 0.

EXAMPLE:

```
sage: F.<a> = FiniteField(5^3, impl='pari_ffelt')
sage: a.is_zero()
False
sage: (a - a).is_zero()
True
```

lift()

If `self` is an element of the prime field, return a lift of this element to an integer.

EXAMPLE:

```
sage: k = FiniteField(next_prime(10^10)^2, 'u', impl='pari_ffelt')
sage: a = k(17)/k(19)
sage: b = a.lift(); b
7894736858
sage: b.parent()
Integer Ring
```

log(*base*)

Return a discrete logarithm of `self` with respect to the given base.

INPUT:

- `base` – non-zero field element

OUTPUT:

An integer x such that `self` equals `base` raised to the power x. If no such x exists, a `ValueError` is raised.

EXAMPLES:

```
sage: F.<g> = FiniteField(2^10, impl='pari_ffelt')
sage: b = g; a = g^37
sage: a.log(b)
37
sage: b^37; a
g^8 + g^7 + g^4 + g + 1
g^8 + g^7 + g^4 + g + 1
```

```
sage: F.<a> = FiniteField(5^2, impl='pari_ffelt')
sage: F(-1).log(F(2))
2
sage: F(1).log(a)
0
```

Some cases where the logarithm is not defined or does not exist:

```
sage: F.<a> = GF(3^10, impl='pari_ffelt')
sage: a.log(-1)
Traceback (most recent call last):
...
ArithmeticError: element a does not lie in group generated by 2
sage: a.log(0)
Traceback (most recent call last):
...
```

```
ArithmeticError: discrete logarithm with base 0 is not defined
sage: F(0).log(1)
Traceback (most recent call last):
...
ArithmeticError: discrete logarithm of 0 is not defined
```

multiplicative_order()

Returns the order of `self` in the multiplicative group.

EXAMPLE:

```
sage: a = FiniteField(5^3, 'a', impl='pari_ffelt').0
sage: a.multiplicative_order()
124
sage: a**124
1
```

polynomial()

Return the unique representative of `self` as a polynomial over the prime field whose degree is less than the degree of the finite field over its prime field.

EXAMPLES:

```
sage: k.<a> = FiniteField(3^2, impl='pari_ffelt')
sage: pol = a.polynomial()
sage: pol
a
sage: parent(pol)
Univariate Polynomial Ring in a over Finite Field of size 3

sage: k = FiniteField(3^4, 'alpha', impl='pari_ffelt')
sage: a = k.gen()
sage: a.polynomial()
alpha
sage: (a**2 + 1).polynomial()
alpha^2 + 1
sage: (a**2 + 1).polynomial().parent()
Univariate Polynomial Ring in alpha over Finite Field of size 3
```

sqrt(*extend=False*, *all=False*)

Return a square root of `self`, if it exists.

INPUT:

- `extend` – bool (default: `False`)

  Warning: This option is not implemented.

- `all` - bool (default: `False`)

OUTPUT:

A square root of `self`, if it exists. If `all` is `True`, a list containing all square roots of `self` (of length zero, one or two) is returned instead.

If `extend` is `True`, a square root is chosen in an extension field if necessary. If `extend` is `False`, a ValueError is raised if the element is not a square in the base field.

Warning: The `extend` option is not implemented (yet).

EXAMPLES:

```
sage: F = FiniteField(7^2, 'a', impl='pari_ffelt')
sage: F(2).sqrt()
4
sage: F(3).sqrt()
5*a + 1
sage: F(3).sqrt()**2
3
sage: F(4).sqrt(all=True)
[2, 5]

sage: K = FiniteField(7^3, 'alpha', impl='pari_ffelt')
sage: K(3).sqrt()
Traceback (most recent call last):
...
ValueError: element is not a square
sage: K(3).sqrt(all=True)
[]

sage: K.<a> = GF(3^17, impl='pari_ffelt')
sage: (a^3 - a - 1).sqrt()
a^16 + 2*a^15 + a^13 + 2*a^12 + a^10 + 2*a^9 + 2*a^8 + a^7 + a^6 + 2*a^5 + a^4 + 2*a^2 + 2*a
```

sage.rings.finite_rings.element_pari_ffelt.**unpickle_FiniteFieldElement_pari_ffelt**(*parent*, *elem*)

EXAMPLE:

```
sage: k.<a> = GF(2^20, impl='pari_ffelt')
sage: e = k.random_element()
sage: f = loads(dumps(e)) # indirect doctest
sage: e == f
True
```

CHAPTER

FIVE

BASE CLASSES FOR FINITE FIELDS

TESTS:

```
sage: K.<a> = NumberField(x^2 + 1)
sage: F = K.factor(3)[0][0].residue_field()
sage: loads(dumps(F)) == F
True
```

AUTHORS:

- Adrien Brochard, David Roe, Jeroen Demeyer, Julian Rueth, Niles Johnson, Peter Bruin, Travis Scrimshaw, Xavier Caruso: initial version

class `sage.rings.finite_rings.finite_field_base.`**`FiniteField`**

Bases: `sage.rings.ring.Field`

Abstract base class for finite fields.

`algebraic_closure`(*name='z'*, ***kwds*)

Return an algebraic closure of `self`.

INPUT:

- `name` – string (default: 'z'): prefix to use for variable names of subfields
- `implementation` – string (optional): specifies how to construct the algebraic closure. The only value supported at the moment is `'pseudo_conway'`. For more details, see `algebraic_closure_finite_field`.

OUTPUT:

An algebraic closure of `self`. Note that mathematically speaking, this is only unique up to *non-unique* isomorphism. To obtain canonically defined algebraic closures, one needs an algorithm that also provides a canonical isomorphism between any two algebraic closures constructed using the algorithm.

This non-uniqueness problem can in principle be solved by using *Conway polynomials*; see for example [CP]. These have the drawback that computing them takes a long time. Therefore Sage implements a variant called *pseudo-Conway polynomials*, which are easier to compute but do not determine an algebraic closure up to unique isomorphism.

The output of this method is cached, so that within the same Sage session, calling it multiple times will return the same algebraic closure (i.e. the same Sage object). Despite this, the non-uniqueness of the current implementation means that coercion and pickling cannot work as one might expect. See below for an example.

EXAMPLE:

```
sage: F = GF(5).algebraic_closure()
sage: F
```

```
Algebraic closure of Finite Field of size 5
sage: F.gen(3)
z3
```

The default name is 'z' but you can change it through the option `name`:

```
sage: Ft = GF(5).algebraic_closure('t')
sage: Ft.gen(3)
t3
```

Because Sage currently only implements algebraic closures using a non-unique definition (see above), it is currently impossible to implement pickling in such a way that a pickled and unpickled element compares equal to the original:

```
sage: F = GF(7).algebraic_closure()
sage: x = F.gen(2)
sage: loads(dumps(x)) == x
False
```

Note: This is currently only implemented for prime fields.

REFERENCE:

TEST:

```
sage: GF(5).algebraic_closure() is GF(5).algebraic_closure()
True
```

cardinality()

Return the cardinality of `self`.

Same as `order()`.

EXAMPLES:

```
sage: GF(997).cardinality()
997
```

construction()

Return the construction of this finite field, as a `ConstructionFunctor` and the base field.

EXAMPLES:

```
sage: v = GF(3^3, conway=True, prefix='z').construction(); v
(AlgebraicExtensionFunctor, Finite Field of size 3)
sage: v[0].polys[0]
3
sage: v = GF(2^1000, 'a').construction(); v[0].polys[0]
a^1000 + a^5 + a^4 + a^3 + 1
```

extension(*modulus*, *name=None*, *names=None*, *map=False*, *embedding=None*, ***kwds*)

Return an extension of this finite field.

INPUT:

- `modulus` – a polynomial with coefficients in `self`, or an integer.
- `name` – string: the name of the generator in the new extension
- `map` – boolean (default: `False`): if `False`, return just the extension E; if `True`, return a pair (E, f), where f is an embedding of `self` into E.

- embedding – currently not used; for compatibility with other AlgebraicExtensionFunctor calls.
- **kwds: further keywords, passed to the finite field constructor.

OUTPUT:

An extension of the given modulus, or pseudo-Conway of the given degree if modulus is an integer.

EXAMPLES:

```
sage: k = GF(2)
sage: R.<x> = k[]
sage: k.extension(x^1000 + x^5 + x^4 + x^3 + 1, 'a')
Finite Field in a of size 2^1000
sage: k = GF(3^4, conway=True, prefix='z')
sage: R.<x> = k[]
sage: k.extension(3, conway=True, prefix='z')
Finite Field in z12 of size 3^12
```

An example using the map argument:

```
sage: F = GF(5)
sage: E, f = F.extension(2, 'b', map=True)
sage: E
Finite Field in b of size 5^2
sage: f
Ring morphism:
  From: Finite Field of size 5
  To:   Finite Field in b of size 5^2
  Defn: 1 |--> 1
sage: f.parent()
Set of field embeddings from Finite Field of size 5 to Finite Field in b of size 5^2
```

Extensions of non-prime finite fields by polynomials are not yet supported: we fall back to generic code:

```
sage: k.extension(x^5 + x^2 + x - 1)
Univariate Quotient Polynomial Ring in x over Finite Field in z4 of size 3^4 with modulus x^
```

factored_order()

Returns the factored order of this field. For compatibility with integer_mod_ring.

EXAMPLES:

```
sage: GF(7^2,'a').factored_order()
7^2
```

factored_unit_order()

Returns the factorization of self.order()-1, as a 1-element list.

The format is for compatibility with integer_mod_ring.

EXAMPLES:

```
sage: GF(7^2,'a').factored_unit_order()
[2^4 * 3]
```

frobenius_endomorphism(*n=1*)

INPUT:

- n – an integer (default: 1)

OUTPUT:

The n-th power of the absolute arithmetic Frobenius endomorphism on this finite field.

EXAMPLES:

```
sage: k.<t> = GF(3^5)
sage: Frob = k.frobenius_endomorphism(); Frob
Frobenius endomorphism t |--> t^3 on Finite Field in t of size 3^5

sage: a = k.random_element()
sage: Frob(a) == a^3
True
```

We can specify a power:

```
sage: k.frobenius_endomorphism(2)
Frobenius endomorphism t |--> t^(3^2) on Finite Field in t of size 3^5
```

The result is simplified if possible:

```
sage: k.frobenius_endomorphism(6)
Frobenius endomorphism t |--> t^3 on Finite Field in t of size 3^5
sage: k.frobenius_endomorphism(5)
Identity endomorphism of Finite Field in t of size 3^5
```

Comparisons work:

```
sage: k.frobenius_endomorphism(6) == Frob
True
sage: from sage.categories.morphism import IdentityMorphism
sage: k.frobenius_endomorphism(5) == IdentityMorphism(k)
True
```

AUTHOR:

- Xavier Caruso (2012-06-29)

gen()

Return a generator of this field (over its prime field). As this is an abstract base class, this just raises a `NotImplementedError`.

EXAMPLES:

```
sage: K = GF(17)
sage: sage.rings.finite_rings.finite_field_base.FiniteField.gen(K)
Traceback (most recent call last):
...
NotImplementedError
```

is_conway()

Return `True` if self is defined by a Conway polynomial.

EXAMPLES:

sage: GF(5^3, 'a').is_conway() True sage: GF(5^3, 'a', modulus='adleman-lenstra').is_conway() False sage: GF(next_prime(2^16, 2), 'a').is_conway() False

is_field(*proof=True*)

Returns whether or not the finite field is a field, i.e., always returns `True`.

EXAMPLES:

```
sage: k.<a> = FiniteField(3^4)
sage: k.is_field()
True
```

is_finite()

Return True since a finite field is finite.

EXAMPLES:

```
sage: GF(997).is_finite()
True
```

is_perfect()

Return whether this field is perfect, i.e., every element has a p-th root. Always returns True since finite fields are perfect.

EXAMPLES:

```
sage: GF(2).is_perfect()
True
```

is_prime_field()

Return True if self is a prime field, i.e., has degree 1.

EXAMPLES:

```
sage: GF(3^7, 'a').is_prime_field()
False
sage: GF(3, 'a').is_prime_field()
True
```

modulus()

Return the minimal polynomial of the generator of self over the prime finite field.

The minimal polynomial of an element a in a field is the unique monic irreducible polynomial of smallest degree with coefficients in the base field that has a as a root. In finite field extensions, $\mathbf{F}_{p^n}$, the base field is $\mathbf{F}_p$.

OUTPUT:

- a monic polynomial over $\mathbf{F}_p$ in the variable x.

EXAMPLES:

```
sage: F.<a> = GF(7^2); F
Finite Field in a of size 7^2
sage: F.polynomial_ring()
Univariate Polynomial Ring in a over Finite Field of size 7
sage: f = F.modulus(); f
x^2 + 6*x + 3
sage: f(a)
0
```

Although f is irreducible over the base field, we can double-check whether or not f factors in F as follows. The command F['x'](f) coerces f as a polynomial with coefficients in F. (Instead of a polynomial with coefficients over the base field.)

```
sage: f.factor()
x^2 + 6*x + 3
sage: F['x'](f).factor()
(x + a + 6) * (x + 6*a)
```

Here is an example with a degree 3 extension:

```
sage: G.<b> = GF(7^3); G
Finite Field in b of size 7^3
sage: g = G.modulus(); g
x^3 + 6*x^2 + 4
sage: g.degree(); G.degree()
3
3
```

For prime fields, this returns $x - 1$ unless a custom modulus was given when constructing this field:

```
sage: k = GF(199)
sage: k.modulus()
x + 198
sage: var('x')
x
sage: k = GF(199, modulus=x+1)
sage: k.modulus()
x + 1
```

The given modulus is always made monic:

```
sage: k.<a> = GF(7^2, modulus=2*x^2-3, impl="pari_ffelt")
sage: k.modulus()
x^2 + 2
```

TESTS:

We test the various finite field implementations:

```
sage: GF(2, impl="modn").modulus()
x + 1
sage: GF(2, impl="givaro").modulus()
x + 1
sage: GF(2, impl="ntl").modulus()
x + 1
sage: GF(2, impl="modn", modulus=x).modulus()
x
sage: GF(2, impl="givaro", modulus=x).modulus()
x
sage: GF(2, impl="ntl", modulus=x).modulus()
x
sage: GF(13^2, 'a', impl="givaro", modulus=x^2+2).modulus()
x^2 + 2
sage: GF(13^2, 'a', impl="pari_ffelt", modulus=x^2+2).modulus()
x^2 + 2
```

multiplicative_generator()

Return a primitive element of this finite field, i.e. a generator of the multiplicative group.

You can use `multiplicative_generator()` or `primitive_element()`, these mean the same thing.

Warning: This generator might change from one version of Sage to another.

EXAMPLES:

```
sage: k = GF(997)
sage: k.multiplicative_generator()
7
sage: k.<a> = GF(11^3)
```

```
sage: k.primitive_element()
a
sage: k.<b> = GF(19^32)
sage: k.multiplicative_generator()
b + 4
```

TESTS:

Check that large characteristics work (trac ticket #11946):

```
sage: p = 10^20 + 39
sage: x = polygen(GF(p))
sage: K.<a> = GF(p^2, modulus=x^2+1)
sage: K.multiplicative_generator()
a + 12
```

ngens()

The number of generators of the finite field. Always 1.

EXAMPLES:

```
sage: k = FiniteField(3^4, 'b')
sage: k.ngens()
1
```

order()

Return the order of this finite field.

EXAMPLES:

```
sage: GF(997).order()
997
```

polynomial(*name=None*)

Return the minimal polynomial of the generator of `self` over the prime finite field.

INPUT:

- `name` – a variable name to use for the polynomial. By default, use the name given when constructing this field.

OUTPUT:

- a monic polynomial over $\mathbf{F}_p$ in the variable `name`.

See also:

Except for the `name` argument, this is identical to the `modulus()` method.

EXAMPLES:

```
sage: k.<a> = FiniteField(9)
sage: k.polynomial('x')
x^2 + 2*x + 2
sage: k.polynomial()
a^2 + 2*a + 2

sage: F = FiniteField(9, 'a', impl='pari_ffelt')
sage: F.polynomial()
a^2 + 2*a + 2

sage: F = FiniteField(7^20, 'a', impl='pari_ffelt')
sage: f = F.polynomial(); f
```

```
a^20 + a^12 + 6*a^11 + 2*a^10 + 5*a^9 + 2*a^8 + 3*a^7 + a^6 + 3*a^5 + 3*a^3 + a + 3
sage: f(F.gen())
0

sage: k.<a> = GF(2^20, impl='ntl')
sage: k.polynomial()
a^20 + a^10 + a^9 + a^7 + a^6 + a^5 + a^4 + a + 1
sage: k.polynomial('FOO')
FOO^20 + FOO^10 + FOO^9 + FOO^7 + FOO^6 + FOO^5 + FOO^4 + FOO + 1
sage: a^20
a^10 + a^9 + a^7 + a^6 + a^5 + a^4 + a + 1
```

polynomial_ring(*variable_name=None*)

Returns the polynomial ring over the prime subfield in the same variable as this finite field.

EXAMPLES:

```
sage: k.<alpha> = FiniteField(3^4)
sage: k.polynomial_ring()
Univariate Polynomial Ring in alpha over Finite Field of size 3
```

primitive_element()

Return a primitive element of this finite field, i.e. a generator of the multiplicative group.

You can use `multiplicative_generator()` or `primitive_element()`, these mean the same thing.

Warning: This generator might change from one version of Sage to another.

EXAMPLES:

```
sage: k = GF(997)
sage: k.multiplicative_generator()
7
sage: k.<a> = GF(11^3)
sage: k.primitive_element()
a
sage: k.<b> = GF(19^32)
sage: k.multiplicative_generator()
b + 4
```

TESTS:

Check that large characteristics work (trac ticket #11946):

```
sage: p = 10^20 + 39
sage: x = polygen(GF(p))
sage: K.<a> = GF(p^2, modulus=x^2+1)
sage: K.multiplicative_generator()
a + 12
```

random_element(**args, **kwds*)

A random element of the finite field. Passes arguments to `random_element()` function of underlying vector space.

EXAMPLES:

```
sage: k = GF(19^4, 'a')
sage: k.random_element()
a^3 + 3*a^2 + 6*a + 9
```

Passes extra positional or keyword arguments through:

```
sage: k.random_element(prob=0)
0
```

some_elements()

Returns a collection of elements of this finite field for use in unit testing.

EXAMPLES:

```
sage: k = GF(2^8,'a')
sage: k.some_elements() # random output
[a^4 + a^3 + 1, a^6 + a^4 + a^3, a^5 + a^4 + a, a^2 + a]
```

subfields(*degree=0*, *name=None*)

Return all subfields of `self` of the given `degree`, or all possible degrees if `degree` is 0.

The subfields are returned as absolute fields together with an embedding into `self`.

INPUT:

- `degree` – (default: 0) an integer
- `name` – a string, a dictionary or `None`:
  - If `degree` is nonzero, then `name` must be a string (or `None`, if this is a pseudo-Conway extension), and will be the variable name of the returned field.
  - If `degree` is zero, the dictionary should have keys the divisors of the degree of this field, with the desired variable name for the field of that degree as an entry.
  - As a shortcut, you can provide a string and the degree of each subfield will be appended for the variable name of that subfield.
  - If `None`, uses the prefix of this field.

OUTPUT:

A list of pairs `(K, e)`, where `K` ranges over the subfields of this field and `e` gives an embedding of `K` into `self`.

EXAMPLES:

```
sage: k.<a> = GF(2^21, conway=True, prefix='z')
sage: k.subfields()
[(Finite Field of size 2,
  Ring morphism:
      From: Finite Field of size 2
      To:   Finite Field in a of size 2^21
      Defn: 1 |--> 1),
 (Finite Field in z3 of size 2^3,
  Ring morphism:
      From: Finite Field in z3 of size 2^3
      To:   Finite Field in a of size 2^21
      Defn: z3 |--> a^20 + a^19 + a^17 + a^15 + a^11 + a^9 + a^8 + a^6 + a^2),
 (Finite Field in z7 of size 2^7,
  Ring morphism:
      From: Finite Field in z7 of size 2^7
      To:   Finite Field in a of size 2^21
      Defn: z7 |--> a^20 + a^19 + a^17 + a^15 + a^14 + a^6 + a^4 + a^3 + a),
 (Finite Field in z21 of size 2^21,
  Ring morphism:
```

```
From: Finite Field in z21 of size 2^21
To:   Finite Field in a of size 2^21
Defn: z21 |--> a)]
```

unit_group_exponent()

The exponent of the unit group of the finite field. For a finite field, this is always the order minus 1.

EXAMPLES:

```
sage: k = GF(2^10, 'a')
sage: k.order()
1024
sage: k.unit_group_exponent()
1023
```

vector_space()

Return the vector space over the prime subfield isomorphic to this finite field as a vector space.

EXAMPLES:

```
sage: GF(27,'a').vector_space()
Vector space of dimension 3 over Finite Field of size 3
```

zeta(*n=None*)

Returns an element of multiplicative order `n` in this finite field, if there is one. Raises a `ValueError` if there is not.

EXAMPLES:

```
sage: k = GF(7)
sage: k.zeta()
3
sage: k.zeta().multiplicative_order()
6
sage: k.zeta(3)
2
sage: k.zeta(3).multiplicative_order()
3
sage: k = GF(49, 'a')
sage: k.zeta().multiplicative_order()
48
sage: k.zeta(6)
3
```

Even more examples:

```
sage: GF(9,'a').zeta_order()
8
sage: GF(9,'a').zeta()
a
sage: GF(9,'a').zeta(4)
a + 1
sage: GF(9,'a').zeta()^2
a + 1
```

zeta_order()

Return the order of the distinguished root of unity in `self`.

EXAMPLES:

```
sage: GF(9,'a').zeta_order()
8
sage: GF(9,'a').zeta()
a
sage: GF(9,'a').zeta().multiplicative_order()
8
```

class sage.rings.finite_rings.finite_field_base.**FiniteFieldIterator**

Bases: object

An iterator over a finite field. This should only be used when the field is an extension of a smaller field which already has a separate iterator function.

next()

x.next() -> the next value, or raise StopIteration

sage.rings.finite_rings.finite_field_base.**is_FiniteField**(*x*)

Return True if x is of type finite field, and False otherwise.

EXAMPLES:

```
sage: from sage.rings.finite_rings.finite_field_base import is_FiniteField
sage: is_FiniteField(GF(9,'a'))
True
sage: is_FiniteField(GF(next_prime(10^10)))
True
```

Note that the integers modulo n are not of type finite field, so this function returns False:

```
sage: is_FiniteField(Integers(7))
False
```

sage.rings.finite_rings.finite_field_base.**unpickle_FiniteField_ext**(*_type*, *order*, *variable_name*, *modulus*, *kwargs*)

Used to unpickle extensions of finite fields. Now superseded (hence no doctest), but kept around for backward compatibility.

EXAMPLES:

```
sage: # not tested
```

sage.rings.finite_rings.finite_field_base.**unpickle_FiniteField_prm**(*_type*, *order*, *variable_name*, *kwargs*)

Used to unpickle finite prime fields. Now superseded (hence no doctest), but kept around for backward compatibility.

EXAMPLE:

```
sage: # not tested
```

CHAPTER

SIX

FINITE EXTENSION FIELDS IMPLEMENTED VIA PARI POLMODS (DEPRECATED).

AUTHORS:

- William Stein: initial version
- Jeroen Demeyer (2010-12-16): fix formatting of docstrings (trac ticket #10487)

class `sage.rings.finite_rings.finite_field_ext_pari.`**`FiniteField_ext_pari`**(*q*, *name*, *modulus=None*)

Bases: `sage.rings.finite_rings.finite_field_base.FiniteField`

Finite Field of order q, where q is a prime power (not a prime), implemented using PARI `POLMOD`. This implementation is the default implementation for $q \geq 2^{16}$.

INPUT:

- `q` – integer, size of the finite field, not prime
- `name` – variable name used for printing elements of the finite field
- `modulus` – an irreducible polynomial to construct this field.

OUTPUT:

A finite field of order q with the given variable name

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(3))
sage: from sage.rings.finite_rings.finite_field_ext_pari import FiniteField_ext_pari
sage: k = FiniteField_ext_pari(9, 'a', modulus=(x^2 + 2*x + 2))
doctest:...: DeprecationWarning: The "pari_mod" finite field implementation is deprecated
See http://trac.sagemath.org/17297 for details.
sage: k
Finite Field in a of size 3^2
sage: k.is_field()
True
sage: k.characteristic()
3
sage: a = k.gen()
sage: a
a
sage: a.parent()
Finite Field in a of size 3^2
sage: a.charpoly('x')
```

```
x^2 + 2*x + 2
sage: [a^i for i in range(8)]
[1, a, a + 1, 2*a + 1, 2, 2*a, 2*a + 2, a + 2]
```

Fields can be coerced into sets or list and iterated over:

```
sage: list(k)
[0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2]
```

The following is a native Python set:

```
sage: set(k)
{0, 1, 2, a, 2*a, a + 1, 2*a + 1, a + 2, 2*a + 2}
```

And the following is a Sage set:

```
sage: Set(k)
{0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2}

We can also make a list via comprehension:
sage: [x for x in k]
[0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2]
```

Next we compute with the finite field of order 16, where the name is named `b`:

```
sage: P.<x> = PolynomialRing(GF(2))
sage: from sage.rings.finite_rings.finite_field_ext_pari import FiniteField_ext_pari
sage: k16 = FiniteField_ext_pari(16, "b", modulus=(x^4 + x + 1))
sage: z = k16.gen()
sage: z
b
sage: z.charpoly('x')
x^4 + x + 1
sage: k16.is_field()
True
sage: k16.characteristic()
2
sage: z.multiplicative_order()
15
```

Of course one can also make prime finite fields:

```
sage: k = FiniteField(7)
```

Note that the generator is 1:

```
sage: k.gen()
1
sage: k.gen().multiplicative_order()
1
```

Prime finite fields are implemented elsewhere, they cannot be constructed using `FiniteField_ext_pari`:

```
sage: k = FiniteField_ext_pari(7, 'a', modulus=polygen(GF(7)))
Traceback (most recent call last):
...
ValueError: The size of the finite field must not be prime.
```

Illustration of dumping and loading:

```
sage: K = FiniteField(7)
sage: loads(K.dumps()) == K
True
sage: K = FiniteField(7^10, 'b', impl='pari_mod')
doctest:...: DeprecationWarning: The "pari_mod" finite field implementation is deprecated
See http://trac.sagemath.org/17297 for details.
sage: loads(K.dumps()) == K
True
sage: K = FiniteField(7^10, 'a', impl='pari_mod')
sage: loads(K.dumps()) == K
True
```

In this example K is large enough that Conway polynomials are not used. Note that when the field is dumped the defining polynomial f is also dumped. Since f is determined by a random algorithm, it's important that f is dumped as part of K. If you quit Sage and restart and remake a finite field of the same order (and the order is large enough so that there is no Conway polynomial), then defining polynomial is probably different. However, if you load a previously saved field, that will have the same defining polynomial.

```
sage: K = GF(10007^10, 'a', impl='pari_mod')
sage: loads(K.dumps()) == K
True
```

Note: We do NOT yet define natural consistent inclusion maps between different finite fields.

characteristic()

Returns the characteristic of the finite field, which is a prime number.

EXAMPLES:

```
sage: k = FiniteField(3^4, 'a', impl='pari_mod')
sage: k.characteristic()
3
```

degree()

Returns the degree of the finite field, which is a positive integer.

EXAMPLES:

```
sage: FiniteField(3^20, 'a', impl='pari_mod').degree()
20
```

gen(*n=0*)

Return a generator of `self` over its prime field, which is a root of `self.modulus()`.

INPUT:

- `n` – must be 0

OUTPUT:

An element a of `self` such that `self.modulus()(a) == 0`.

Warning: This generator is not guaranteed to be a generator for the multiplicative group. To obtain the latter, use `multiplicative_generator()` or use the `modulus="primitive"` option when constructing the field.

EXAMPLES:

```
sage: FiniteField(2^4, "b", impl='pari_mod').gen()
b
```

```
sage: k = FiniteField(3^4, "alpha", impl='pari_mod')
sage: a = k.gen()
sage: a
alpha
sage: a^4
alpha^3 + 1
```

order()

The number of elements of the finite field.

EXAMPLES:

```
sage: k = FiniteField(2^10, 'a', impl='pari_mod')
sage: k
Finite Field in a of size 2^10
sage: k.order()
1024
```

CHAPTER

SEVEN

GIVARO FINITE FIELD

Finite fields that are implemented using Zech logs and the cardinality must be less than 2^{16}. By default, conway polynomials are used as minimal polynomial.

TESTS:

Test backwards compatibility:

```
sage: from sage.rings.finite_rings.finite_field_givaro import FiniteField_givaro
sage: FiniteField_givaro(9, 'a')
doctest:...: DeprecationWarning: constructing a FiniteField_givaro without giving a polynomial as moc
See http://trac.sagemath.org/16930 for details.
Finite Field in a of size 3^2
```

class sage.rings.finite_rings.finite_field_givaro.**FiniteField_givaro**(*q*, *name='a'*, *modulus=None*, *repr='poly'*, *cache=False*)

Bases: sage.rings.finite_rings.finite_field_base.FiniteField

Finite field implemented using Zech logs and the cardinality must be less than 2^{16}. By default, conway polynomials are used as minimal polynomials.

INPUT:

- q – p^n (must be prime power)
- name – (default: 'a') variable used for poly_repr()
- modulus – A minimal polynomial to use for reduction.
- repr – (default: 'poly') controls the way elements are printed to the user:
  - 'log': repr is log_repr()
  - 'int': repr is int_repr()
  - 'poly': repr is poly_repr()
- cache – (default: False) if True a cache of all elements of this field is created. Thus, arithmetic does not create new elements which speeds calculations up. Also, if many elements are needed during a calculation this cache reduces the memory requirement as at most order() elements are created.

OUTPUT:

Givaro finite field with characteristic p and cardinality p^n.

EXAMPLES:

By default conway polynomials are used for extension fields:

```
sage: k.<a> = GF(2**8)
sage: -a ^ k.degree()
a^4 + a^3 + a^2 + 1
sage: f = k.modulus(); f
x^8 + x^4 + x^3 + x^2 + 1
```

You may enforce a modulus:

```
sage: P.<x> = PolynomialRing(GF(2))
sage: f = x^8 + x^4 + x^3 + x + 1 # Rijndael Polynomial
sage: k.<a> = GF(2^8, modulus=f)
sage: k.modulus()
x^8 + x^4 + x^3 + x + 1
sage: a^(2^8)
a
```

You may enforce a random modulus:

```
sage: k = GF(3**5, 'a', modulus='random')
sage: k.modulus() # random polynomial
x^5 + 2*x^4 + 2*x^3 + x^2 + 2
```

Three different representations are possible:

```
sage: FiniteField(9, 'a', impl='givaro', repr='poly').gen()
a
sage: FiniteField(9, 'a', impl='givaro', repr='int').gen()
3
sage: FiniteField(9, 'a', impl='givaro', repr='log').gen()
1
```

For prime fields, the default modulus is the polynomial $x - 1$, but you can ask for a different modulus:

```
sage: GF(1009, impl='givaro').modulus()
x + 1008
sage: GF(1009, impl='givaro', modulus='conway').modulus()
x + 998
```

a_times_b_minus_c (*a*, *b*, *c*)

Return `a*b - c`.

INPUT:

- `a,b,c` – `FiniteField_givaroElement`

EXAMPLES:

```
sage: k.<a> = GF(3**3)
sage: k.a_times_b_minus_c(a,a,k(1))
a^2 + 2
```

a_times_b_plus_c (*a*, *b*, *c*)

Return `a*b + c`. This is faster than multiplying `a` and `b` first and adding `c` to the result.

INPUT:

- `a,b,c` – `FiniteField_givaroElement`

EXAMPLES:

```
sage: k.<a> = GF(2**8)
sage: k.a_times_b_plus_c(a,a,k(1))
a^2 + 1
```

c_minus_a_times_b (*a*, *b*, *c*)

Return `c - a*b`.

INPUT:

- `a,b,c` – `FiniteField_givaroElement`

EXAMPLES:

```
sage: k.<a> = GF(3**3)
sage: k.c_minus_a_times_b(a,a,k(1))
2*a^2 + 1
```

characteristic ()

Return the characteristic of this field.

EXAMPLES:

```
sage: p = GF(19^5,'a').characteristic(); p
19
sage: type(p)
<type 'sage.rings.integer.Integer'>
```

degree ()

If the cardinality of `self` is p^n, then this returns n.

OUTPUT:

Integer – the degree

EXAMPLES:

```
sage: GF(3^4,'a').degree()
4
```

fetch_int (*n*)

Given an integer n return a finite field element in `self` which equals n under the condition that `gen()` is set to `characteristic()`.

EXAMPLES:

```
sage: k.<a> = GF(2^8)
sage: k.fetch_int(8)
a^3
sage: e = k.fetch_int(151); e
a^7 + a^4 + a^2 + a + 1
sage: 2^7 + 2^4 + 2^2 + 2 + 1
151
```

frobenius_endomorphism (*n=1*)

INPUT:

- `n` – an integer (default: 1)

OUTPUT:

The n-th power of the absolute arithmetic Frobenius endomorphism on this finite field.

EXAMPLES:

```
sage: k.<t> = GF(3^5)
sage: Frob = k.frobenius_endomorphism(); Frob
Frobenius endomorphism t |--> t^3 on Finite Field in t of size 3^5

sage: a = k.random_element()
sage: Frob(a) == a^3
True
```

We can specify a power:

```
sage: k.frobenius_endomorphism(2)
Frobenius endomorphism t |--> t^(3^2) on Finite Field in t of size 3^5
```

The result is simplified if possible:

```
sage: k.frobenius_endomorphism(6)
Frobenius endomorphism t |--> t^3 on Finite Field in t of size 3^5
sage: k.frobenius_endomorphism(5)
Identity endomorphism of Finite Field in t of size 3^5
```

Comparisons work:

```
sage: k.frobenius_endomorphism(6) == Frob
True
sage: from sage.categories.morphism import IdentityMorphism
sage: k.frobenius_endomorphism(5) == IdentityMorphism(k)
True
```

AUTHOR:

- Xavier Caruso (2012-06-29)

gen (*n=0*)

Return a generator of `self` over its prime field, which is a root of `self.modulus()`.

INPUT:

- `n` – must be 0

OUTPUT:

An element a of `self` such that `self.modulus()(a) == 0`.

> **Warning:** This generator is not guaranteed to be a generator for the multiplicative group. To obtain the latter, use `multiplicative_generator()` or use the `modulus="primitive"` option when constructing the field.

EXAMPLES:

```
sage: k = GF(3^4, 'b'); k.gen()
b
sage: k.gen(1)
Traceback (most recent call last):
...
IndexError: only one generator
sage: F = FiniteField(31, impl='givaro')
sage: F.gen()
1
```

int_to_log (*n*)

Given an integer n this method returns i where i satisfies $g^i = n \mod p$ where g is the generator and p is the characteristic of `self`.

INPUT:

- `n` – integer representation of an finite field element

OUTPUT:

log representation of `n`

EXAMPLES:

```
sage: k = GF(7**3, 'a')
sage: k.int_to_log(4)
228
sage: k.int_to_log(3)
57
sage: k.gen()^57
3
```

log_to_int (*n*)

Given an integer n this method returns `i` where `i` satisfies $g^n = i$ where g is the generator of `self`; the result is interpreted as an integer.

INPUT:

- `n` – log representation of a finite field element

OUTPUT:

integer representation of a finite field element.

EXAMPLES:

```
sage: k = GF(2**8, 'a')
sage: k.log_to_int(4)
16
sage: k.log_to_int(20)
180
```

order ()

Return the cardinality of this field.

OUTPUT:

Integer – the number of elements in `self`.

EXAMPLES:

```
sage: n = GF(19^5,'a').order(); n
2476099
sage: type(n)
<type 'sage.rings.integer.Integer'>
```

prime_subfield ()

Return the prime subfield $\mathbf{F}_p$ of self if `self` is $\mathbf{F}_{p^n}$.

EXAMPLES:

```
sage: GF(3^4, 'b').prime_subfield()
Finite Field of size 3

sage: S.<b> = GF(5^2); S
Finite Field in b of size 5^2
sage: S.prime_subfield()
```

```
Finite Field of size 5
sage: type(S.prime_subfield())
<class 'sage.rings.finite_rings.finite_field_prime_modn.FiniteField_prime_modn_with_category
```

random_element (**args*, ***kwds*)

Return a random element of `self`.

EXAMPLES:

```
sage: k = GF(23**3, 'a')
sage: e = k.random_element(); e
2*a^2 + 14*a + 21
sage: type(e)
<type 'sage.rings.finite_rings.element_givaro.FiniteField_givaroElement'>

sage: P.<x> = PowerSeriesRing(GF(3^3, 'a'))
sage: P.random_element(5)
2*a + 2 + (a^2 + a + 2)*x + (2*a + 1)*x^2 + (2*a^2 + a)*x^3 + 2*a^2*x^4 + O(x^5)
```

CHAPTER

EIGHT

FINITE FIELDS OF CHARACTERISTIC 2

TESTS:

Test backwards compatibility:

```
sage: from sage.rings.finite_rings.finite_field_ntl_gf2e import FiniteField_ntl_gf2e
sage: FiniteField_ntl_gf2e(16, 'a')
doctest:...: DeprecationWarning: constructing a FiniteField_ntl_gf2e without giving a polynomial as r
See http://trac.sagemath.org/16983 for details.
Finite Field in a of size 2^4
```

class sage.rings.finite_rings.finite_field_ntl_gf2e.**FiniteField_ntl_gf2e**(*q*, *names='a'*, *modulus=None*, *repr='poly'*)

Bases: sage.rings.finite_rings.finite_field_base.FiniteField

Finite Field of characteristic 2 and order 2^n.

INPUT:

- q – 2^n (must be 2 power)
- names – variable used for poly_repr (default: 'a')
- modulus – A minimal polynomial to use for reduction.
- **repr – controls the way elements are printed to the user:** (default: 'poly')
  - 'poly': polynomial representation

OUTPUT:

Finite field with characteristic 2 and cardinality 2^n.

EXAMPLES:

```
sage: k.<a> = GF(2^16)
sage: type(k)
<class 'sage.rings.finite_rings.finite_field_ntl_gf2e.FiniteField_ntl_gf2e_with_category'>
sage: k.<a> = GF(2^1024)
sage: k.modulus()
x^1024 + x^19 + x^6 + x + 1
sage: set_random_seed(0)
sage: k.<a> = GF(2^17, modulus='random')
sage: k.modulus()
x^17 + x^16 + x^15 + x^10 + x^8 + x^6 + x^4 + x^3 + x^2 + x + 1
```

```
sage: k.modulus().is_irreducible()
True
sage: k.<a> = GF(2^211, modulus='minimal_weight')
sage: k.modulus()
x^211 + x^11 + x^10 + x^8 + 1
sage: k.<a> = GF(2^211, modulus='conway')
sage: k.modulus()
x^211 + x^9 + x^6 + x^5 + x^3 + x + 1
sage: k.<a> = GF(2^23, modulus='conway')
sage: a.multiplicative_order() == k.order() - 1
True
```

characteristic()

Return the characteristic of `self` which is 2.

EXAMPLES:

```
sage: k.<a> = GF(2^16,modulus='random')
sage: k.characteristic()
2
```

degree()

If this field has cardinality 2^n this method returns n.

EXAMPLES:

```
sage: k.<a> = GF(2^64)
sage: k.degree()
64
```

fetch_int(*number*)

Given an integer n less than `cardinality()` with base 2 representation $a_0 + 2 \cdot a_1 + \cdots + 2^k a_k$, returns $a_0 + a_1 \cdot x + \cdots + a_k x^k$, where x is the generator of this finite field.

INPUT:

- `number` – an integer

EXAMPLES:

```
sage: k.<a> = GF(2^48)
sage: k.fetch_int(2^43 + 2^15 + 1)
a^43 + a^15 + 1
sage: k.fetch_int(33793)
a^15 + a^10 + 1
sage: 33793.digits(2) # little endian
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1]
```

gen(*n=0*)

Return a generator of `self` over its prime field, which is a root of `self.modulus()`.

INPUT:

- `n` – must be 0

OUTPUT:

An element a of `self` such that `self.modulus()(a) == 0`.

> **Warning:** This generator is not guaranteed to be a generator for the multiplicative group. To obtain the latter, use `multiplicative_generator()` or use the `modulus="primitive"` option when constructing the field.

EXAMPLES:

```
sage: k.<a> = GF(2^19)
sage: k.gen() == a
True
sage: a
a
```

TESTS:

```
sage: GF(2, impl='ntl').gen()
1
sage: GF(2, impl='ntl', modulus=polygen(GF(2)) ).gen()
0
sage: GF(2^19, 'a').gen(1)
Traceback (most recent call last):
...
IndexError: only one generator
```

order()

Return the cardinality of this field.

EXAMPLES:

```
sage: k.<a> = GF(2^64)
sage: k.order()
18446744073709551616
```

prime_subfield()

Return the prime subfield $\mathbf{F}_p$ of `self` if `self` is $\mathbf{F}_{p^n}$.

EXAMPLES:

```
sage: F.<a> = GF(2^16)
sage: F.prime_subfield()
Finite Field of size 2
```

sage.rings.finite_rings.finite_field_ntl_gf2e.**late_import**()

Imports various modules after startup.

EXAMPLES:

```
sage: sage.rings.finite_rings.finite_field_ntl_gf2e.late_import()
sage: sage.rings.finite_rings.finite_field_ntl_gf2e.GF2 is None # indirect doctest
False
```

CHAPTER NINE

FINITE FIELDS IMPLEMENTED VIA PARI'S FFELT TYPE

AUTHORS:

- Peter Bruin (June 2013): initial version, based on finite_field_ext_pari.py by William Stein et al.

class `sage.rings.finite_rings.finite_field_pari_ffelt.FiniteField_pari_ffelt`(*p*, *modulus*, *name=None*)

Bases: `sage.rings.finite_rings.finite_field_base.FiniteField`

Finite fields whose cardinality is a prime power (not a prime), implemented using PARI's `FFELT` type.

INPUT:

- `p` – prime number
- `modulus` – an irreducible polynomial of degree at least 2 over the field of p elements
- `name` – string: name of the distinguished generator (default: variable name of `modulus`)

OUTPUT:

A finite field of order $q = p^n$, generated by a distinguished element with minimal polynomial `modulus`. Elements are represented as polynomials in `name` of degree less than n.

Note: Direct construction of `FiniteField_pari_ffelt` objects requires specifying a characteristic and a modulus. To construct a finite field by specifying a cardinality and an algorithm for finding an irreducible polynomial, use the `FiniteField` constructor with `impl='pari_ffelt'`.

EXAMPLES:

Some computations with a finite field of order 9:

```
sage: k = FiniteField(9, 'a', impl='pari_ffelt')
sage: k
Finite Field in a of size 3^2
sage: k.is_field()
True
sage: k.characteristic()
3
sage: a = k.gen()
sage: a
a
sage: a.parent()
Finite Field in a of size 3^2
sage: a.charpoly('x')
x^2 + 2*x + 2
```

```
sage: [a^i for i in range(8)]
[1, a, a + 1, 2*a + 1, 2, 2*a, 2*a + 2, a + 2]
sage: TestSuite(k).run()
```

Next we compute with a finite field of order 16:

```
sage: k16 = FiniteField(16, 'b', impl='pari_ffelt')
sage: z = k16.gen()
sage: z
b
sage: z.charpoly('x')
x^4 + x + 1
sage: k16.is_field()
True
sage: k16.characteristic()
2
sage: z.multiplicative_order()
15
```

Illustration of dumping and loading:

```
sage: K = FiniteField(7^10, 'b', impl='pari_ffelt')
sage: loads(K.dumps()) == K
True

sage: K = FiniteField(10007^10, 'a', impl='pari_ffelt')
sage: loads(K.dumps()) == K
True
```

Element

alias of `FiniteFieldElement_pari_ffelt`

characteristic()

Return the characteristic of `self`.

EXAMPLE:

```
sage: F = FiniteField(3^4, 'a', impl='pari_ffelt')
sage: F.characteristic()
3
```

degree()

Returns the degree of `self` over its prime field.

EXAMPLE:

```
sage: F = FiniteField(3^20, 'a', impl='pari_ffelt')
sage: F.degree()
20
```

gen(*n=0*)

Return a generator of `self` over its prime field, which is a root of `self.modulus()`.

INPUT:

- `n` – must be 0

OUTPUT:

An element a of `self` such that `self.modulus()(a) == 0`.

> **Warning:** This generator is not guaranteed to be a generator for the multiplicative group. To obtain the latter, use `multiplicative_generator()` or use the `modulus="primitive"` option when constructing the field.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(GF(2))
sage: FiniteField(2^4, 'b', impl='pari_ffelt').gen()
b
sage: k = FiniteField(3^4, 'alpha', impl='pari_ffelt')
sage: a = k.gen()
sage: a
alpha
sage: a^4
alpha^3 + 1
```

CHAPTER

TEN

FINITE PRIME FIELDS

AUTHORS:

- William Stein: initial version
- Martin Albrecht (2008-01): refactoring

TESTS:

```
sage: k = GF(3)
sage: TestSuite(k).run()
```

class sage.rings.finite_rings.finite_field_prime_modn.**FiniteField_prime_modn**(*p*, *check=True*, *modulus=None*)

Bases: sage.rings.finite_rings.finite_field_base.FiniteField, sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic

Finite field of order p where p is prime.

EXAMPLES:

```
sage: FiniteField(3)
Finite Field of size 3

sage: FiniteField(next_prime(1000))
Finite Field of size 1009
```

characteristic()

Return the characteristic of code{self}.

EXAMPLES:

```
sage: k = GF(7)
sage: k.characteristic()
7
```

construction()

Returns the construction of this finite field (for use by sage.categories.pushout)

EXAMPLES:

```
sage: GF(3).construction()
(QuotientFunctor, Integer Ring)
```

degree()

Return the degree of `self` over its prime field.

This always returns 1.

EXAMPLES:

```
sage: FiniteField(3).degree()
1
```

gen (*n=0*)

Return a generator of `self` over its prime field, which is a root of `self.modulus()`.

Unless a custom modulus was given when constructing this prime field, this returns 1.

INPUT:

- `n` – must be 0

OUTPUT:

An element a of `self` such that `self.modulus()(a) == 0`.

> **Warning:** This generator is not guaranteed to be a generator for the multiplicative group. To obtain the latter, use `multiplicative_generator()` or use the `modulus="primitive"` option when constructing the field.

EXAMPLES:

```
sage: k = GF(13)
sage: k.gen()
1
sage: k = GF(1009, modulus="primitive")
sage: k.gen()  # this gives a primitive element
11
sage: k.gen(1)
Traceback (most recent call last):
...
IndexError: only one generator
```

is_prime_field ()

Return `True` since this is a prime field.

EXAMPLES:

```
sage: k.<a> = GF(3)
sage: k.is_prime_field()
True

sage: k.<a> = GF(3^2)
sage: k.is_prime_field()
False
```

order ()

Return the order of this finite field.

EXAMPLES:

```
sage: k = GF(5)
sage: k.order()
5
```

polynomial (*name=None*)

Returns the polynomial `name`.

EXAMPLES:

```
sage: k.<a> = GF(3)
sage: k.polynomial()
x
```

CHAPTER ELEVEN

FINITE FIELD MORPHISMS

This file provides several classes implementing:

- embeddings between finite fields
- Frobenius isomorphism on finite fields

EXAMPLES:

```
sage: from sage.rings.finite_rings.hom_finite_field import FiniteFieldHomomorphism_generic
```

Construction of an embedding:

```
sage: k.<t> = GF(3^7)
sage: K.<T> = GF(3^21)
sage: f = FiniteFieldHomomorphism_generic(Hom(k, K)); f
Ring morphism:
  From: Finite Field in t of size 3^7
  To:   Finite Field in T of size 3^21
  Defn: t |--> T^20 + 2*T^18 + T^16 + 2*T^13 + T^9 + 2*T^8 + T^7 + T^6 + T^5 + T^3 + 2*T^2 + T

sage: f(t)
T^20 + 2*T^18 + T^16 + 2*T^13 + T^9 + 2*T^8 + T^7 + T^6 + T^5 + T^3 + 2*T^2 + T
```

The map f has a method `section` which returns a partially defined map which is the inverse of f on the image of f:

```
sage: g = f.section(); g
Section of Ring morphism:
  From: Finite Field in t of size 3^7
  To:   Finite Field in T of size 3^21
  Defn: t |--> T^20 + 2*T^18 + T^16 + 2*T^13 + T^9 + 2*T^8 + T^7 + T^6 + T^5 + T^3 + 2*T^2 + T
sage: g(f(t^3+t^2+1))
t^3 + t^2 + 1
sage: g(T)
Traceback (most recent call last):
...
ValueError: T is not in the image of Ring morphism:
  From: Finite Field in t of size 3^7
  To:   Finite Field in T of size 3^21
  Defn: t |--> T^20 + 2*T^18 + T^16 + 2*T^13 + T^9 + 2*T^8 + T^7 + T^6 + T^5 + T^3 + 2*T^2 + T
```

There is no embedding of $GF(5^6)$ into $GF(5^11)$:

```
sage: k.<t> = GF(5^6)
sage: K.<T> = GF(5^11)
sage: FiniteFieldHomomorphism_generic(Hom(k, K))
Traceback (most recent call last):
```

```
...
ValueError: No embedding of Finite Field in t of size 5^6 into Finite Field in T of size 5^11
```

Construction of Frobenius endomorphisms:

```
sage: k.<t> = GF(7^14)
sage: Frob = k.frobenius_endomorphism(); Frob
Frobenius endomorphism t |--> t^7 on Finite Field in t of size 7^14
sage: Frob(t)
t^7
```

Some basic arithmetics is supported:

```
sage: Frob^2
Frobenius endomorphism t |--> t^(7^2) on Finite Field in t of size 7^14
sage: f = k.frobenius_endomorphism(7); f
Frobenius endomorphism t |--> t^(7^7) on Finite Field in t of size 7^14
sage: f*Frob
Frobenius endomorphism t |--> t^(7^8) on Finite Field in t of size 7^14

sage: Frob.order()
14
sage: f.order()
2
```

Note that simplifications are made automatically:

```
sage: Frob^16
Frobenius endomorphism t |--> t^(7^2) on Finite Field in t of size 7^14
sage: Frob^28
Identity endomorphism of Finite Field in t of size 7^14
```

And that comparisons work:

```
sage: Frob == Frob^15
True
sage: Frob^14 == Hom(k, k).identity()
True
```

AUTHOR:

- Xavier Caruso (2012-06-29)

class sage.rings.finite_rings.hom_finite_field.**FiniteFieldHomomorphism_generic**

Bases: sage.rings.morphism.RingHomomorphism_im_gens

A class implementing embeddings between finite fields.

is_injective()

Return True since a embedding between finite fields is always injective.

EXAMPLES:

```
sage: from sage.rings.finite_rings.hom_finite_field import FiniteFieldHomomorphism_generic
sage: k.<t> = GF(3^3)
sage: K.<T> = GF(3^9)
sage: f = FiniteFieldHomomorphism_generic(Hom(k, K))
sage: f.is_injective()
True
```

is_surjective()

Return true if this embedding is surjective (and hence an isomorphism.

EXAMPLES:

```
sage: from sage.rings.finite_rings.hom_finite_field import FiniteFieldHomomorphism_generic
sage: k.<t> = GF(3^3)
sage: K.<T> = GF(3^9)
sage: f = FiniteFieldHomomorphism_generic(Hom(k, K))
sage: f.is_surjective()
False
sage: g = FiniteFieldHomomorphism_generic(Hom(k, k))
sage: g.is_surjective()
True
```

section()

Return the `inverse` of this embedding.

It is a partially defined map whose domain is the codomain of the embedding, but which is only defined on the image of the embedding.

EXAMPLES:

```
sage: from sage.rings.finite_rings.hom_finite_field import FiniteFieldHomomorphism_generic
sage: k.<t> = GF(3^7)
sage: K.<T> = GF(3^21)
sage: f = FiniteFieldHomomorphism_generic(Hom(k, K));
sage: g = f.section(); g
Section of Ring morphism:
  From: Finite Field in t of size 3^7
  To:   Finite Field in T of size 3^21
  Defn: t |--> T^20 + 2*T^18 + T^16 + 2*T^13 + T^9 + 2*T^8 + T^7 + T^6 + T^5 + T^3 + 2*T^2 +
sage: g(f(t^3+t^2+1))
t^3 + t^2 + 1
sage: g(T)
Traceback (most recent call last):
...
ValueError: T is not in the image of Ring morphism:
  From: Finite Field in t of size 3^7
  To:   Finite Field in T of size 3^21
  Defn: t |--> T^20 + 2*T^18 + T^16 + 2*T^13 + T^9 + 2*T^8 + T^7 + T^6 + T^5 + T^3 + 2*T^2 +
```

class sage.rings.finite_rings.hom_finite_field.**FrobeniusEndomorphism_finite_field**

Bases: sage.rings.morphism.FrobeniusEndomorphism_generic

A class implementing Frobenius endomorphisms on finite fields.

fixed_field()

Return the fixed field of `self`.

OUTPUT:

- a tuple (K, e), where K is the subfield of the domain consisting of elements fixed by `self` and e is an embedding of K into the domain.

Note: The name of the variable used for the subfield (if it is not a prime subfield) is suffixed by `_fixed`.

EXAMPLES:

```
sage: k.<t> = GF(5^6)
sage: f = k.frobenius_endomorphism(2)
```

```
sage: kfixed, embed = f.fixed_field()
sage: kfixed
Finite Field in t_fixed of size 5^2
sage: embed
Ring morphism:
  From: Finite Field in t_fixed of size 5^2
  To:   Finite Field in t of size 5^6
  Defn: t_fixed |--> 4*t^5 + 2*t^4 + 4*t^2 + t

sage: tfixed = kfixed.gen()
sage: embed(tfixed)
4*t^5 + 2*t^4 + 4*t^2 + t
```

is_identity()

Return true if this morphism is the identity morphism.

EXAMPLES:

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: Frob.is_identity()
False
sage: (Frob^3).is_identity()
True
```

is_injective()

Return true since any power of the Frobenius endomorphism over a finite field is always injective.

EXAMPLES:

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: Frob.is_injective()
True
```

is_surjective()

Return true since any power of the Frobenius endomorphism over a finite field is always surjective.

EXAMPLES:

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism()
sage: Frob.is_surjective()
True
```

order()

Return the order of this endomorphism.

EXAMPLES:

```
sage: k.<t> = GF(5^12)
sage: Frob = k.frobenius_endomorphism()
sage: Frob.order()
12
sage: (Frob^2).order()
6
sage: (Frob^9).order()
4
```

power()

Return an integer n such that this endormorphism is the n-th power of the absolute (arithmetic) Frobenius.

EXAMPLES:

```
sage: k.<t> = GF(5^12)
sage: Frob = k.frobenius_endomorphism()
sage: Frob.power()
1
sage: (Frob^9).power()
9
sage: (Frob^13).power()
1
```

class `sage.rings.finite_rings.hom_finite_field.`**`SectionFiniteFieldHomomorphism_generic`**

Bases: `sage.categories.map.Section`

A class implementing sections of embeddings between finite fields.

CHAPTER

TWELVE

FINITE FIELD MORPHISMS USING GIVARO

Special implementation for givaro finite fields of:

- embeddings between finite fields
- frobenius endomorphisms

SEEALSO:

```
:mod:`sage.rings.finite_rings.hom_finite_field`
```

AUTHOR:

- Xavier Caruso (2012-06-29)

class sage.rings.finite_rings.hom_finite_field_givaro.**FiniteFieldHomomorphism_givaro**

Bases: sage.rings.finite_rings.hom_finite_field.FiniteFieldHomomorphism_generic

TESTS:

```
sage: from sage.rings.finite_rings.hom_finite_field_givaro import FiniteFieldHomomorphism_givaro
sage: k.<t> = GF(3^2)
sage: K.<T> = GF(3^4)
sage: f = FiniteFieldHomomorphism_givaro(Hom(k, K)); f
Ring morphism:
  From: Finite Field in t of size 3^2
  To:   Finite Field in T of size 3^4
  Defn: t |--> 2*T^3 + 2*T^2 + 1

sage: k.<t> = GF(3^10)
sage: K.<T> = GF(3^20)
sage: f = FiniteFieldHomomorphism_givaro(Hom(k, K)); f
Traceback (most recent call last):
...
TypeError: The codomain is not an instance of FiniteField_givaro
```

class sage.rings.finite_rings.hom_finite_field_givaro.**FrobeniusEndomorphism_givaro**

Bases: sage.rings.finite_rings.hom_finite_field.FrobeniusEndomorphism_finite_field

TESTS:

```
sage: k.<t> = GF(5^3)
sage: Frob = k.frobenius_endomorphism(); Frob
Frobenius endomorphism t |--> t^5 on Finite Field in t of size 5^3
sage: type(Frob)
<type 'sage.rings.finite_rings.hom_finite_field_givaro.FrobeniusEndomorphism_givaro'>

sage: k.<t> = GF(5^20)
sage: Frob = k.frobenius_endomorphism(); Frob
```

```
Frobenius endomorphism t |--> t^5 on Finite Field in t of size 5^20
sage: type(Frob)
<type 'sage.rings.finite_rings.hom_finite_field.FrobeniusEndomorphism_finite_field'>
```

fixed_field()

Return the fixed field of `self`.

OUTPUT:

- a tuple (K, e), where K is the subfield of the domain consisting of elements fixed by `self` and e is an embedding of K into the domain.

Note: The name of the variable used for the subfield (if it is not a prime subfield) is suffixed by `_fixed`.

EXAMPLES:

```
sage: k.<t> = GF(5^6)
sage: f = k.frobenius_endomorphism(2)
sage: kfixed, embed = f.fixed_field()
sage: kfixed
Finite Field in t_fixed of size 5^2
sage: embed
Ring morphism:
  From: Finite Field in t_fixed of size 5^2
  To:   Finite Field in t of size 5^6
  Defn: t_fixed |--> 4*t^5 + 2*t^4 + 4*t^2 + t

sage: tfixed = kfixed.gen()
sage: embed(tfixed)
4*t^5 + 2*t^4 + 4*t^2 + t
```

class sage.rings.finite_rings.hom_finite_field_givaro.**SectionFiniteFieldHomomorphism_givaro**

Bases: sage.rings.finite_rings.hom_finite_field.SectionFiniteFieldHomomorphism_generic

TESTS:

```
sage: from sage.rings.finite_rings.hom_finite_field_givaro import FiniteFieldHomomorphism_givaro
sage: k.<t> = GF(3^2)
sage: K.<T> = GF(3^4)
sage: f = FiniteFieldHomomorphism_givaro(Hom(k, K))
sage: g = f.section(); g
Section of Ring morphism:
  From: Finite Field in t of size 3^2
  To:   Finite Field in T of size 3^4
  Defn: t |--> 2*T^3 + 2*T^2 + 1
```

CHAPTER

THIRTEEN

FINITE FIELD MORPHISMS FOR PRIME FIELDS

Special implementation for prime finite field of:

- embeddings of such field into general finite fields
- Frobenius endomorphisms (= identity with our assumptions)

See also:

`sage.rings.finite_rings.hom_finite_field`

AUTHOR:

- Xavier Caruso (2012-06-29)

class `sage.rings.finite_rings.hom_prime_finite_field.`**`FiniteFieldHomomorphism_prime`**

Bases: `sage.rings.finite_rings.hom_finite_field.FiniteFieldHomomorphism_generic`

A class implementing embeddings of prime finite fields into general finite fields.

class `sage.rings.finite_rings.hom_prime_finite_field.`**`FrobeniusEndomorphism_prime`**

Bases: `sage.rings.finite_rings.hom_finite_field.FrobeniusEndomorphism_finite_field`

A class implementing Frobenius endomorphism on prime finite fields (i.e. identity map :-).

`fixed_field``()`

Return the fixed field of `self`.

OUTPUT:

- a tuple (K, e), where K is the subfield of the domain consisting of elements fixed by `self` and e is an embedding of K into the domain.

Note: Since here the domain is a prime field, the subfield is the same prime field and the embedding is necessarily the identity map.

EXAMPLES:

```
sage: k.<t> = GF(5)
sage: f = k.frobenius_endomorphism(2); f
Identity endomorphism of Finite Field of size 5
sage: kfixed, embed = f.fixed_field()

sage: kfixed == k
True
sage: [ embed(x) == x for x in kfixed ]
[True, True, True, True, True]
```

class `sage.rings.finite_rings.hom_prime_finite_field.`**`SectionFiniteFieldHomomorphism_prime`**

Bases: `sage.rings.finite_rings.hom_finite_field.SectionFiniteFieldHomomorphism_generic`

CHAPTER

FOURTEEN

HOMSET FOR FINITE FIELDS

This is the set of all field homomorphisms between two finite fields.

EXAMPLES:

```
sage: R.<t> = ZZ[]
sage: E.<a> = GF(25, modulus = t^2 - 2)
sage: F.<b> = GF(625)
sage: H = Hom(E, F)
sage: f = H([4*b^3 + 4*b^2 + 4*b]); f
Ring morphism:
  From: Finite Field in a of size 5^2
  To:   Finite Field in b of size 5^4
  Defn: a |--> 4*b^3 + 4*b^2 + 4*b
sage: f(2)
2
sage: f(a)
4*b^3 + 4*b^2 + 4*b
sage: len(H)
2
sage: [phi(2*a)^2 for phi in Hom(E, F)]
[3, 3]
```

We can also create endomorphisms:

```
sage: End(E)
Automorphism group of Finite Field in a of size 5^2
sage: End(GF(7))[0]
Ring endomorphism of Finite Field of size 7
  Defn: 1 |--> 1
sage: H = Hom(GF(7), GF(49, 'c'))
sage: H[0](2)
2
```

class `sage.rings.finite_rings.homset.`**`FiniteFieldHomset`**(*R*, *S*, *category=None*)

Bases: `sage.rings.homset.RingHomset_generic`

Set of homomorphisms with domain a given finite field.

`index`(*item*)

Return the index of `self`.

EXAMPLES:

```
sage: K.<z> = GF(1024)
sage: g = End(K)[3]
sage: End(K).index(g) == 3
True
```

is_aut()

Check if `self` is an automorphism

EXAMPLES:

```
sage: Hom(GF(4, 'a'), GF(16, 'b')).is_aut()
False
sage: Hom(GF(4, 'a'), GF(4, 'c')).is_aut()
False
sage: Hom(GF(4, 'a'), GF(4, 'a')).is_aut()
True
```

list()

Return a list of all the elements in this set of field homomorphisms.

EXAMPLES:

```
sage: K.<a> = GF(25)
sage: End(K)
Automorphism group of Finite Field in a of size 5^2
sage: list(End(K))
[Ring endomorphism of Finite Field in a of size 5^2
  Defn: a |--> 4*a + 1,
 Ring endomorphism of Finite Field in a of size 5^2
  Defn: a |--> a]
sage: L.<z> = GF(7^6)
sage: [g for g in End(L) if (g^3)(z) == z]
[Ring endomorphism of Finite Field in z of size 7^6
  Defn: z |--> z,
 Ring endomorphism of Finite Field in z of size 7^6
  Defn: z |--> 5*z^4 + 5*z^3 + 4*z^2 + 3*z + 1,
 Ring endomorphism of Finite Field in z of size 7^6
  Defn: z |--> 3*z^5 + 5*z^4 + 5*z^2 + 2*z + 3]
```

Between isomorphic fields with different moduli:

```
sage: k1 = GF(1009)
sage: k2 = GF(1009, modulus="primitive")
sage: Hom(k1, k2).list()
[
Ring morphism:
  From: Finite Field of size 1009
  To:   Finite Field of size 1009
  Defn: 1 |--> 1
]
sage: Hom(k2, k1).list()
[
Ring morphism:
  From: Finite Field of size 1009
  To:   Finite Field of size 1009
  Defn: 11 |--> 11
]

sage: k1.<a> = GF(1009^2, modulus="first_lexicographic")
sage: k2.<b> = GF(1009^2, modulus="conway")
sage: Hom(k1, k2).list()
[
Ring morphism:
  From: Finite Field in a of size 1009^2
```

```
  To:   Finite Field in b of size 1009^2
  Defn: a |--> 290*b + 864,
Ring morphism:
  From: Finite Field in a of size 1009^2
  To:   Finite Field in b of size 1009^2
  Defn: a |--> 719*b + 145
]
```

TESTS:

Check that trac ticket #11390 is fixed:

```
sage: K = GF(1<<16,'a'); L = GF(1<<32,'b')
sage: K.Hom(L)[0]
Ring morphism:
  From: Finite Field in a of size 2^16
  To:   Finite Field in b of size 2^32
  Defn: a |--> b^29 + b^27 + b^26 + b^23 + b^21 + b^19 + b^18 + b^16 + b^14 + b^13 + b^11 +
```

order()

Return the order of this set of field homomorphisms.

EXAMPLES:

```
sage: K.<a> = GF(125)
sage: End(K)
Automorphism group of Finite Field in a of size 5^3
sage: End(K).order()
3
sage: L.<b> = GF(25)
sage: Hom(L, K).order() == Hom(K, L).order() == 0
True
```

CHAPTER FIFTEEN

RING $\mathbb{Z}/N\mathbb{Z}$ OF INTEGERS MODULO N

EXAMPLES:

```
sage: R = Integers(97)
sage: a = R(5)
sage: a**100000000000000000000000000000000000000000000000000000000000000
61
```

This example illustrates the relation between $\mathbf{Z}/p\mathbf{Z}$ and $\mathbf{F}_p$. In particular, there is a canonical map to $\mathbf{F}_p$, but not in the other direction.

```
sage: r = Integers(7)
sage: s = GF(7)
sage: r.has_coerce_map_from(s)
False
sage: s.has_coerce_map_from(r)
True
sage: s(1) + r(1)
2
sage: parent(s(1) + r(1))
Finite Field of size 7
sage: parent(r(1) + s(1))
Finite Field of size 7
```

We list the elements of $\mathbf{Z}/3\mathbf{Z}$:

```
sage: R = Integers(3)
sage: list(R)
[0, 1, 2]
```

AUTHORS:

- William Stein (initial code)
- David Joyner (2005-12-22): most examples
- Robert Bradshaw (2006-08-24): convert to SageX (Cython)
- William Stein (2007-04-29): square_roots_of_one
- Simon King (2011-04-21): allow to prescribe a category
- Simon King (2013-09): Only allow to prescribe the category of fields

class `sage.rings.finite_rings.integer_mod_ring.`**`IntegerModFactory`**

Bases: `sage.structure.factory.UniqueFactory`

Return the quotient ring $\mathbf{Z}/n\mathbf{Z}$.

INPUT:

- `order` – integer (default: 0); positive or negative
- `is_field` – bool (default: `False`); assert that the order is prime and hence the quotient ring belongs to the category of fields

Note: The optional argument `is_field` is not part of the cache key. Hence, this factory will create precisely one instance of $\mathbf{Z}/n\mathbf{Z}$. However, if `is_field` is true, then a previously created instance of the quotient ring will be updated to be in the category of fields.

Use with care! Erroneously putting $\mathbf{Z}/n\mathbf{Z}$ into the category of fields may have consequences that can compromise a whole Sage session, so that a restart will be needed.

EXAMPLES:

```
sage: IntegerModRing(15)
Ring of integers modulo 15
sage: IntegerModRing(7)
Ring of integers modulo 7
sage: IntegerModRing(-100)
Ring of integers modulo 100
```

Note that you can also use `Integers`, which is a synonym for `IntegerModRing`.

```
sage: Integers(18)
Ring of integers modulo 18
sage: Integers() is Integers(0) is ZZ
True
```

Note: Testing whether a quotient ring $\mathbf{Z}/n\mathbf{Z}$ is a field can of course be very costly. By default, it is not tested whether n is prime or not, in contrast to `GF()`. If the user is sure that the modulus is prime and wants to avoid a primality test, (s)he can provide `category=Fields()` when constructing the quotient ring, and then the result will behave like a field. If the category is not provided during initialisation, and it is found out later that the ring is in fact a field, then the category will be changed at runtime, having the same effect as providing `Fields()` during initialisation.

EXAMPLES:

```
sage: R = IntegerModRing(5)
sage: R.category()
Join of Category of finite commutative rings
    and Category of subquotients of monoids
    and Category of quotients of semigroups
    and Category of finite enumerated sets
sage: R in Fields()
True
sage: R.category()
Join of Category of finite fields
    and Category of subquotients of monoids
    and Category of quotients of semigroups
sage: S = IntegerModRing(5, is_field=True)
sage: S is R
True
```

Warning: If the optional argument `is_field` was used by mistake, there is currently no way to revert its impact, even though `IntegerModRing_generic.is_field()` with the optional argument `proof=True` would return the correct answer. So, prescribe `is_field=True` only if you know what your are doing!

EXAMPLES:

```
sage: R = IntegerModRing(15, is_field=True)
sage: R in Fields()
True
sage: R.is_field()
True
```

If the optional argument $proof = True$ is provided, primality is tested and the mistaken category assignment is reported:

```
sage: R.is_field(proof=True)
Traceback (most recent call last):
...
ValueError: THIS SAGE SESSION MIGHT BE SERIOUSLY COMPROMISED!
The order 15 is not prime, but this ring has been put
into the category of fields. This may already have consequences
in other parts of Sage. Either it was a mistake of the user,
or a probabilitstic primality test has failed.
In the latter case, please inform the developers.
```

However, the mistaken assignment is not automatically corrected:

```
sage: R in Fields()
True
```

create_key_and_extra_args (*order=0, is_field=False*)

An integer mod ring is specified uniquely by its order.

EXAMPLES:

```
sage: Zmod.create_key_and_extra_args(7)
(7, {})
sage: Zmod.create_key_and_extra_args(7, True)
(7, {'category': Category of fields})
```

create_object (*version, order, **kwds*)

EXAMPLES:

```
sage: R = Integers(10)
sage: TestSuite(R).run() # indirect doctest
```

get_object (*version, key, extra_args*)

class sage.rings.finite_rings.integer_mod_ring.**IntegerModRing_generic** (*order, cache=None, category=None*)

Bases: `sage.rings.quotient_ring.QuotientRing_generic`

The ring of integers modulo N, with N composite.

INPUT:

- `order` – an integer

•`category` – a subcategory of `CommutativeRings()` (the default)

OUTPUT:

The ring of integers modulo N.

EXAMPLES:

First we compute with integers modulo 29.

```
sage: FF = IntegerModRing(29)
sage: FF
Ring of integers modulo 29
sage: FF.category()
Join of Category of finite commutative rings
    and Category of subquotients of monoids
    and Category of quotients of semigroups
    and Category of finite enumerated sets
sage: FF.is_field()
True
sage: FF.characteristic()
29
sage: FF.order()
29
sage: gens = FF.unit_gens()
sage: a = gens[0]
sage: a
2
sage: a.is_square()
False
sage: def pow(i): return a**i
sage: [pow(i) for i in range(16)]
[1, 2, 4, 8, 16, 3, 6, 12, 24, 19, 9, 18, 7, 14, 28, 27]
sage: TestSuite(FF).run()
```

We have seen above that an integer mod ring is, by default, not initialised as an object in the category of fields. However, one can force it to be. Moreover, testing containment in the category of fields my re-initialise the category of the integer mod ring:

```
sage: F19 = IntegerModRing(19, is_field=True)
sage: F19.category().is_subcategory(Fields())
True
sage: F23 = IntegerModRing(23)
sage: F23.category().is_subcategory(Fields())
False
sage: F23 in Fields()
True
sage: F23.category().is_subcategory(Fields())
True
sage: TestSuite(F19).run()
sage: TestSuite(F23).run()
```

By trac ticket #15229, there is a unique instance of the integral quotient ring of a given order. Using the `IntegerModRing()` factory twice, and using `is_field=True` the second time, will update the category of the unique instance:

```
sage: F31a = IntegerModRing(31)
sage: F31a.category().is_subcategory(Fields())
False
sage: F31b = IntegerModRing(31, is_field=True)
sage: F31a is F31b
```

```
True
sage: F31a.category().is_subcategory(Fields())
True
```

Next we compute with the integers modulo 16.

```
sage: Z16 = IntegerModRing(16)
sage: Z16.category()
Join of Category of finite commutative rings
    and Category of subquotients of monoids
    and Category of quotients of semigroups
    and Category of finite enumerated sets
sage: Z16.is_field()
False
sage: Z16.order()
16
sage: Z16.characteristic()
16
sage: gens = Z16.unit_gens()
sage: gens
(15, 5)
sage: a = gens[0]
sage: b = gens[1]
sage: def powa(i): return a**i
sage: def powb(i): return b**i
sage: gp_exp = FF.unit_group_exponent()
sage: gp_exp
28
sage: [powa(i) for i in range(15)]
[1, 15, 1, 15, 1, 15, 1, 15, 1, 15, 1, 15, 1, 15, 1]
sage: [powb(i) for i in range(15)]
[1, 5, 9, 13, 1, 5, 9, 13, 1, 5, 9, 13, 1, 5, 9]
sage: a.multiplicative_order()
2
sage: b.multiplicative_order()
4
sage: TestSuite(Z16).run()
```

Saving and loading:

```
sage: R = Integers(100000)
sage: TestSuite(R).run()  # long time (17s on sage.math, 2011)
```

Testing ideals and quotients:

```
sage: Z10 = Integers(10)
sage: I = Z10.principal_ideal(0)
sage: Z10.quotient(I) == Z10
True
sage: I = Z10.principal_ideal(2)
sage: Z10.quotient(I) == Z10
False
sage: I.is_prime()
True
```

```
sage: R = IntegerModRing(97)
sage: a = R(5)
sage: a**(10^62)
61
```

cardinality()

Return the cardinality of this ring.

EXAMPLES:

```
sage: Zmod(87).cardinality()
87
```

characteristic()

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: FF = IntegerModRing(17)
sage: FF.characteristic()
17
sage: R.characteristic()
18
```

degree()

Return 1.

EXAMPLE:

```
sage: R = Integers(12345678900)
sage: R.degree()
1
```

extension(*poly*, *name=None*, *names=None*, *embedding=None*)

Return an algebraic extension of `self`. See `sage.rings.ring.CommutativeRing.extension()` for more information.

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: Integers(8).extension(t^2 - 3)
Univariate Quotient Polynomial Ring in t over Ring of integers modulo 8 with modulus t^2 + 5
```

factored_order()

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: FF = IntegerModRing(17)
sage: R.factored_order()
2 * 3^2
sage: FF.factored_order()
17
```

factored_unit_order()

Return a list of `Factorization` objects, each the factorization of the order of the units in a $\mathbf{Z}/p^n\mathbf{Z}$ component of this group (using the Chinese Remainder Theorem).

EXAMPLES:

```
sage: R = Integers(8*9*25*17*29)
sage: R.factored_unit_order()
[2^2, 2 * 3, 2^2 * 5, 2^4, 2^2 * 7]
```

field()

If this ring is a field, return the corresponding field as a finite field, which may have extra functionality and structure. Otherwise, raise a `ValueError`.

EXAMPLES:

```
sage: R = Integers(7); R
Ring of integers modulo 7
sage: R.field()
Finite Field of size 7
sage: R = Integers(9)
sage: R.field()
Traceback (most recent call last):
...
ValueError: self must be a field
```

is_field(*proof=None*)

Return True precisely if the order is prime.

INPUT:

- `proof` (optional bool or None, default None): If `False`, then test whether the category of the quotient is a subcategory of `Fields()`, or do a probabilistic primality test. If `None`, then test the category and then do a primality test according to the global arithmetic proof settings. If True, do a deterministic primality test.

If it is found (perhaps probabilistically) that the ring is a field, then the category of the ring is refined to include the category of fields. This may change the Python class of the ring!

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: R.is_field()
False
sage: FF = IntegerModRing(17)
sage: FF.is_field()
True
```

By trac ticket #15229, the category of the ring is refined, if it is found that the ring is in fact a field:

```
sage: R = IntegerModRing(127)
sage: R.category()
Join of Category of finite commutative rings
    and Category of subquotients of monoids
    and Category of quotients of semigroups
    and Category of finite enumerated sets
sage: R.is_field()
True
sage: R.category()
Join of Category of finite fields
    and Category of subquotients of monoids
    and Category of quotients of semigroups
```

It is possible to mistakenly put $\mathbf{Z}/n\mathbf{Z}$ into the category of fields. In this case, `is_field()` will return True without performing a primality check. However, if the optional argument $proof = True$ is provided, primality is tested and the mistake is uncovered in a warning message:

```
sage: R = IntegerModRing(21, is_field=True)
sage: R.is_field()
True
sage: R.is_field(proof=True)
Traceback (most recent call last):
...
ValueError: THIS SAGE SESSION MIGHT BE SERIOUSLY COMPROMISED!
The order 21 is not prime, but this ring has been put
into the category of fields. This may already have consequences
```

```
in other parts of Sage. Either it was a mistake of the user,
or a probabilitstic primality test has failed.
In the latter case, please inform the developers.
```

is_finite()

Return `True` since $\mathbf{Z}/N\mathbf{Z}$ is finite for all positive N.

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: R.is_finite()
True
```

is_integral_domain(*proof=None*)

Return `True` if and only if the order of `self` is prime.

EXAMPLES:

```
sage: Integers(389).is_integral_domain()
True
sage: Integers(389^2).is_integral_domain()
False
```

TESTS:

Check that trac ticket #17453 is fixed:

```
sage: R = Zmod(5)
sage: R in IntegralDomains()
True
```

is_noetherian()

Check if `self` is a Noetherian ring.

EXAMPLES:

```
sage: Integers(8).is_noetherian()
True
```

is_prime_field()

Return `True` if the order is prime.

EXAMPLES:

```
sage: Zmod(7).is_prime_field()
True
sage: Zmod(8).is_prime_field()
False
```

is_unique_factorization_domain(*proof=None*)

Return `True` if and only if the order of `self` is prime.

EXAMPLES:

```
sage: Integers(389).is_unique_factorization_domain()
True
sage: Integers(389^2).is_unique_factorization_domain()
False
```

krull_dimension()

Return the Krull dimension of `self`.

EXAMPLES:

```
sage: Integers(18).krull_dimension()
0
```

list_of_elements_of_multiplicative_group()

Return a list of all invertible elements, as python ints.

EXAMPLES:

```
sage: R = Zmod(12)
sage: L = R.list_of_elements_of_multiplicative_group(); L
[1, 5, 7, 11]
sage: type(L[0])
<type 'int'>
```

modulus()

Return the polynomial $x - 1$ over this ring.

Note: This function exists for consistency with the finite-field modulus function.

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: R.modulus()
x + 17
sage: R = IntegerModRing(17)
sage: R.modulus()
x + 16
```

multiplicative_generator()

Return a generator for the multiplicative group of this ring, assuming the multiplicative group is cyclic.

Use the unit_gens function to obtain generators even in the non-cyclic case.

EXAMPLES:

```
sage: R = Integers(7); R
Ring of integers modulo 7
sage: R.multiplicative_generator()
3
sage: R = Integers(9)
sage: R.multiplicative_generator()
2
sage: Integers(8).multiplicative_generator()
Traceback (most recent call last):
...
ValueError: multiplicative group of this ring is not cyclic
sage: Integers(4).multiplicative_generator()
3
sage: Integers(25*3).multiplicative_generator()
Traceback (most recent call last):
...
ValueError: multiplicative group of this ring is not cyclic
sage: Integers(25*3).unit_gens()
(26, 52)
sage: Integers(162).unit_gens()
(83,)
```

multiplicative_group_is_cyclic()

Return `True` if the multiplicative group of this field is cyclic. This is the case exactly when the order is less than 8, a power of an odd prime, or twice a power of an odd prime.

EXAMPLES:

```
sage: R = Integers(7); R
Ring of integers modulo 7
sage: R.multiplicative_group_is_cyclic()
True
sage: R = Integers(9)
sage: R.multiplicative_group_is_cyclic()
True
sage: Integers(8).multiplicative_group_is_cyclic()
False
sage: Integers(4).multiplicative_group_is_cyclic()
True
sage: Integers(25*3).multiplicative_group_is_cyclic()
False
```

We test that trac ticket #5250 is fixed:

```
sage: Integers(162).multiplicative_group_is_cyclic()
True
```

multiplicative_subgroups()

Return generators for each subgroup of $(\mathbf{Z}/N\mathbf{Z})^*$.

EXAMPLES:

```
sage: Integers(5).multiplicative_subgroups()
((2,), (4,), ())
sage: Integers(15).multiplicative_subgroups()
((11, 7), (4, 11), (8,), (11,), (14,), (7,), (4,), ())
sage: Integers(2).multiplicative_subgroups()
((),)
sage: len(Integers(341).multiplicative_subgroups())
80
```

TESTS:

```
sage: IntegerModRing(1).multiplicative_subgroups()
((),)
sage: IntegerModRing(2).multiplicative_subgroups()
((),)
sage: IntegerModRing(3).multiplicative_subgroups()
((2,), ())
```

order()

Return the order of this ring.

EXAMPLES:

```
sage: Zmod(87).order()
87
```

quadratic_nonresidue()

Return a quadratic non-residue in `self`.

EXAMPLES:

```
sage: R = Integers(17)
sage: R.quadratic_nonresidue()
3
```

```
sage: R(3).is_square()
False
```

random_element (*bound=None*)

Return a random element of this ring.

INPUT:

- bound, a positive integer or None (the default). Is given, return the coercion of an integer in the interval [-bound, bound] into this ring.

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: R.random_element()
2
```

We test bound-option:

```
sage: R.random_element(2) in [R(16), R(17), R(0), R(1), R(2)]
True
```

square_roots_of_one ()

Return all square roots of 1 in self, i.e., all solutions to $x^2 - 1 = 0$.

OUTPUT:

The square roots of 1 in self as a tuple.

EXAMPLES:

```
sage: R = Integers(2^10)
sage: [x for x in R if x^2 == 1]
[1, 511, 513, 1023]
sage: R.square_roots_of_one()
(1, 511, 513, 1023)

sage: v = Integers(9*5).square_roots_of_one(); v
(1, 19, 26, 44)
sage: [x^2 for x in v]
[1, 1, 1, 1]
sage: v = Integers(9*5*8).square_roots_of_one(); v
(1, 19, 71, 89, 91, 109, 161, 179, 181, 199, 251, 269, 271, 289, 341, 359)
sage: [x^2 for x in v]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

unit_gens (***kwds*)

Returns generators for the unit group $(\mathbf{Z}/N\mathbf{Z})^*$.

We compute the list of generators using a deterministic algorithm, so the generators list will always be the same. For each odd prime divisor of N there will be exactly one corresponding generator; if N is even there will be 0, 1 or 2 generators according to whether 2 divides N to order 1, 2 or ≥ 3.

OUTPUT:

A tuple containing the units of self.

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: R.unit_gens()
(11,)
sage: R = IntegerModRing(17)
```

```
sage: R.unit_gens()
(3,)
sage: IntegerModRing(next_prime(10^30)).unit_gens()
(5,)
```

The choice of generators is affected by the optional keyword `algorithm`; this can be `'sage'` (default) or `'pari'`. See `unit_group()` for details.

sage: A = Zmod(55) sage: A.unit_gens(algorithm='sage') (12, 46) sage: A.unit_gens(algorithm='pari') (2, 21)

TESTS:

```
sage: IntegerModRing(2).unit_gens()
()
sage: IntegerModRing(4).unit_gens()
(3,)
sage: IntegerModRing(8).unit_gens()
(7, 5)
```

unit_group(*algorithm='sage'*)

Return the unit group of `self`.

INPUT:

- `self` – the ring $\mathbf{Z}/n\mathbf{Z}$ for a positive integer n
- `algorithm` – either `'sage'` (default) or `'pari'`

OUTPUT:

The unit group of `self`. This is a finite Abelian group equipped with a distinguished set of generators, which is computed using a deterministic algorithm depending on the `algorithm` parameter.

- If `algorithm == 'sage'`, the generators correspond to the prime factors $p \mid n$ (one generator for each odd p; the number of generators for $p = 2$ is 0, 1 or 2 depending on the order to which 2 divides n).
- If `algorithm == 'pari'`, the generators are chosen such that their orders form a decreasing sequence with respect to divisibility.

EXAMPLES:

The output of the algorithms `'sage'` and `'pari'` can differ in various ways. In the following example, the same cyclic factors are computed, but in a different order:

```
sage: A = Zmod(15)
sage: G = A.unit_group(); G
Multiplicative Abelian group isomorphic to C2 x C4
sage: G.gens_values()
(11, 7)
sage: H = A.unit_group(algorithm='pari'); H
Multiplicative Abelian group isomorphic to C4 x C2
sage: H.gens_values()
(7, 11)
```

Here are two examples where the cyclic factors are isomorphic, but are ordered differently and have different generators:

```
sage: A = Zmod(40)
sage: G = A.unit_group(); G
Multiplicative Abelian group isomorphic to C2 x C2 x C4
```

```
sage: G.gens_values()
(31, 21, 17)
sage: H = A.unit_group(algorithm='pari'); H
Multiplicative Abelian group isomorphic to C4 x C2 x C2
sage: H.gens_values()
(17, 21, 11)

sage: A = Zmod(192)
sage: G = A.unit_group(); G
Multiplicative Abelian group isomorphic to C2 x C16 x C2
sage: G.gens_values()
(127, 133, 65)
sage: H = A.unit_group(algorithm='pari'); H
Multiplicative Abelian group isomorphic to C16 x C2 x C2
sage: H.gens_values()
(133, 31, 65)
```

In the following examples, the cyclic factors are not even isomorphic:

```
sage: A = Zmod(319)
sage: A.unit_group()
Multiplicative Abelian group isomorphic to C10 x C28
sage: A.unit_group(algorithm='pari')
Multiplicative Abelian group isomorphic to C140 x C2

sage: A = Zmod(30.factorial())
sage: A.unit_group()
Multiplicative Abelian group isomorphic to C2 x C16777216 x C3188646 x C62500 x C2058 x C110
sage: A.unit_group(algorithm='pari')
Multiplicative Abelian group isomorphic to C20499647385305088000000 x C55440 x C12 x C12 x C
```

TESTS:

We test the cases where the unit group is trivial:

```
sage: A = Zmod(1)
sage: A.unit_group()
Trivial Abelian group
sage: A.unit_group(algorithm='pari')
Trivial Abelian group
sage: A = Zmod(2)
sage: A.unit_group()
Trivial Abelian group
sage: A.unit_group(algorithm='pari')
Trivial Abelian group

sage: Zmod(3).unit_group(algorithm='bogus')
Traceback (most recent call last):
...
ValueError: unknown algorithm 'bogus' for computing the unit group
```

unit_group_exponent()

EXAMPLES:

```
sage: R = IntegerModRing(17)
sage: R.unit_group_exponent()
16
sage: R = IntegerModRing(18)
sage: R.unit_group_exponent()
```

```
6
```

unit_group_order()

Return the order of the unit group of this residue class ring.

EXAMPLES:

```
sage: R = Integers(500)
sage: R.unit_group_order()
200
```

sage.rings.finite_rings.integer_mod_ring.**crt**(*v*)

INPUT:

- v – (list) a lift of elements of rings.IntegerMod(n), for various coprime moduli n

EXAMPLES:

```
sage: from sage.rings.finite_rings.integer_mod_ring import crt
sage: crt([mod(3, 8),mod(1,19),mod(7, 15)])
1027
```

sage.rings.finite_rings.integer_mod_ring.**is_IntegerModRing**(*x*)

Return True if x is an integer modulo ring.

EXAMPLES:

```
sage: from sage.rings.finite_rings.integer_mod_ring import is_IntegerModRing
sage: R = IntegerModRing(17)
sage: is_IntegerModRing(R)
True
sage: is_IntegerModRing(GF(13))
True
sage: is_IntegerModRing(GF(4, 'a'))
False
sage: is_IntegerModRing(10)
False
sage: is_IntegerModRing(ZZ)
False
```

CHAPTER

SIXTEEN

FINITE RESIDUE FIELDS.

We can take the residue field of maximal ideals in the ring of integers of number fields. We can also take the residue field of irreducible polynomials over $GF(p)$.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3-7)
sage: P = K.ideal(29).factor()[0][0]
sage: k = K.residue_field(P)
sage: k
Residue field in abar of Fractional ideal (2*a^2 + 3*a - 10)
sage: k.order()
841
```

We reduce mod a prime for which the ring of integers is not monogenic (i.e., 2 is an essential discriminant divisor):

```
sage: K.<a> = NumberField(x^3 + x^2 - 2*x + 8)
sage: F = K.factor(2); F
(Fractional ideal (1/2*a^2 - 1/2*a + 1)) * (Fractional ideal (-a^2 + 2*a - 3)) * (Fractional ideal (-
sage: F[0][0].residue_field()
Residue field of Fractional ideal (1/2*a^2 - 1/2*a + 1)
sage: F[1][0].residue_field()
Residue field of Fractional ideal (-a^2 + 2*a - 3)
sage: F[2][0].residue_field()
Residue field of Fractional ideal (-3/2*a^2 + 5/2*a - 4)
```

We can also form residue fields from **Z**:

```
sage: ZZ.residue_field(17)
Residue field of Integers modulo 17
```

And for polynomial rings over finite fields:

```
sage: R.<t> = GF(5)[]
sage: I = R.ideal(t^2 + 2)
sage: k = ResidueField(I); k
Residue field in tbar of Principal ideal (t^2 + 2) of Univariate Polynomial Ring in t over Finite Fie
```

AUTHORS:

- David Roe (2007-10-3): initial version
- William Stein (2007-12): bug fixes
- John Cremona (2008-9): extend reduction maps to the whole valuation ring add support for residue fields of ZZ
- David Roe (2009-12): added support for $GF(p)(t)$ and moved to new coercion framework.

TESTS:

```
sage: K.<z> = CyclotomicField(7)
sage: P = K.factor(17)[0][0]
sage: ff = K.residue_field(P)
sage: loads(dumps(ff)) is ff
True
sage: a = ff(z)
sage: parent(a*a)
Residue field in zbar of Fractional ideal (17)
sage: TestSuite(ff).run()
```

Verify that trac ticket #15192 has been resolved:

```
sage: a.is_unit()
True

sage: R.<t> = GF(11)[]; P = R.ideal(t^3 + t + 4)
sage: ff.<a> = ResidueField(P)
sage: a == ff(t)
True
sage: parent(a*a)
Residue field in a of Principal ideal (t^3 + t + 4) of Univariate Polynomial Ring in t over Finite Fi
```

Verify that trac ticket #7475 is fixed:

```
sage: K = ZZ.residue_field(2)
sage: loads(dumps(K)) is K
True
```

Reducing a curve modulo a prime:

```
sage: K.<s> = NumberField(x^2+23)
sage: OK = K.ring_of_integers()
sage: E = EllipticCurve([0,0,0,K(1),K(5)])
sage: pp = K.factor(13)[0][0]
sage: Fpp = OK.residue_field(pp)
sage: E.base_extend(Fpp)
Elliptic Curve defined by y^2  = x^3 + x + 5 over Residue field of Fractional ideal (13, 1/2*s + 9/2)

sage: R.<t> = GF(11)[]
sage: P = R.ideal(t^3 + t + 4)
sage: ff.<a> = R.residue_field(P)
sage: E = EllipticCurve([0,0,0,R(1),R(t)])
sage: E.base_extend(ff)
Elliptic Curve defined by y^2 = x^3 + x + a over Residue field in a of Principal ideal (t^3 + t + 4)
```

Calculating Groebner bases over various residue fields. First over a small non-prime field:

```
sage: F1.<u> = NumberField(x^6 + 6*x^5 + 124*x^4 + 452*x^3 + 4336*x^2 + 8200*x + 42316)
sage: reduct_id = F1.factor(47)[0][0]
sage: Rf = F1.residue_field(reduct_id)
sage: type(Rf)
<class 'sage.rings.finite_rings.residue_field.ResidueFiniteField_pari_ffelt_with_category'>
sage: Rf.cardinality().factor()
47^3
sage: R.<X, Y> = PolynomialRing(Rf)
sage: ubar = Rf(u)
sage: I = ideal([ubar*X + Y]); I
```

```
Ideal ((ubar)*X + Y) of Multivariate Polynomial Ring in X, Y over Residue field in ubar of Fractional
sage: I.groebner_basis()
[X + (-19*ubar^2 - 5*ubar - 17)*Y]
```

And now over a large prime field:

```
sage: x = ZZ['x'].0
sage: F1.<u> = NumberField(x^2 + 6*x + 324)
sage: reduct_id = F1.prime_above(next_prime(2^42))
sage: Rf = F1.residue_field(reduct_id)
sage: type(Rf)
<class 'sage.rings.finite_rings.residue_field.ResidueFiniteField_prime_modn_with_category'>
sage: Rf.cardinality().factor()
4398046511119
sage: S.<X, Y, Z> = PolynomialRing(Rf, order='lex')
sage: I = ideal([2*X - Y^2, Y + Z])
sage: I.groebner_basis()
verbose 0 (...: multi_polynomial_ideal.py, groebner_basis) Warning: falling back to very slow toy imp
[X + 2199023255559*Z^2, Y + Z]
sage: S.<X, Y, Z> = PolynomialRing(Rf, order='deglex')
sage: I = ideal([2*X - Y^2, Y + Z])
sage: I.groebner_basis()
verbose 0 (...: multi_polynomial_ideal.py, groebner_basis) Warning: falling back to very slow toy imp
[Z^2 + 4398046511117*X, Y + Z]
```

class `sage.rings.finite_rings.residue_field.`**`LiftingMap`**

Bases: `sage.categories.map.Section`

Lifting map from residue class field to number field.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3 + 2)
sage: F = K.factor(5)[0][0].residue_field()
sage: F.degree()
2
sage: L = F.lift_map(); L
Lifting map:
  From: Residue field in abar of Fractional ideal (a^2 + 2*a - 1)
  To:   Maximal Order in Number Field in a with defining polynomial x^3 + 2
sage: L(F.0^2)
3*a + 1
sage: L(3*a + 1) == F.0^2
True

sage: R.<t> = GF(13)[]
sage: P = R.ideal(8*t^12 + 9*t^11 + 11*t^10 + 2*t^9 + 11*t^8 + 3*t^7 + 12*t^6 + t^4 + 7*t^3 + 5*
sage: k.<a> = P.residue_field()
sage: k.lift_map()
Lifting map:
  From: Residue field in a of Principal ideal (t^12 + 6*t^11 + 3*t^10 + 10*t^9 + 3*t^8 + 2*t^7 +
  To:   Univariate Polynomial Ring in t over Finite Field of size 13
```

class `sage.rings.finite_rings.residue_field.`**`ReductionMap`**

Bases: `sage.categories.map.Map`

A reduction map from a (subset) of a number field or function field to this residue class field.

It will be defined on those elements of the field with non-negative valuation at the specified prime.

EXAMPLES:

```
sage: I = QQ[sqrt(17)].factor(5)[0][0]; I
Fractional ideal (5)
sage: k = I.residue_field(); k
Residue field in sqrt17bar of Fractional ideal (5)
sage: R = k.reduction_map(); R
Partially defined reduction map:
  From: Number Field in sqrt17 with defining polynomial x^2 - 17
  To:   Residue field in sqrt17bar of Fractional ideal (5)

sage: R.<t> = GF(next_prime(2^20))[]; P = R.ideal(t^2 + t + 1)
sage: k = P.residue_field()
sage: k.reduction_map()
Partially defined reduction map:
  From: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 1048583
  To:   Residue field in tbar of Principal ideal (t^2 + t + 1) of Univariate Polynomial Ring in
```

section()

Computes a section of the map, namely a map that lifts elements of the residue field to elements of the field.

EXAMPLES:

```
sage: K.<a> = NumberField(x^5 - 5*x + 2)
sage: P = K.ideal(47).factor()[0][0]
sage: k = K.residue_field(P)
sage: f = k.convert_map_from(K)
sage: s = f.section(); s
Lifting map:
  From: Residue field in abar of Fractional ideal (14*a^4 - 24*a^3 - 26*a^2 + 58*a - 15)
  To:   Number Field in a with defining polynomial x^5 - 5*x + 2
sage: s(k.gen())
a
sage: L.<b> = NumberField(x^5 + 17*x + 1)
sage: P = L.factor(53)[0][0]
sage: l = L.residue_field(P)
sage: g = l.convert_map_from(L)
sage: s = g.section(); s
Lifting map:
  From: Residue field in bbar of Fractional ideal (53, b^2 + 23*b + 8)
  To:   Number Field in b with defining polynomial x^5 + 17*x + 1
sage: s(l.gen()).parent()
Number Field in b with defining polynomial x^5 + 17*x + 1

sage: R.<t> = GF(2)[]; h = t^5 + t^2 + 1
sage: k.<a> = R.residue_field(h)
sage: K = R.fraction_field()
sage: f = k.convert_map_from(K)
sage: f.section()
Lifting map:
  From: Residue field in a of Principal ideal (t^5 + t^2 + 1) of Univariate Polynomial Ring
  To:   Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 2 (using
```

class sage.rings.finite_rings.residue_field.**ResidueFieldFactory**

Bases: sage.structure.factory.UniqueFactory

A factory that returns the residue class field of a prime ideal p of the ring of integers of a number field, or of a polynomial ring over a finite field.

INPUT:

- `p` – a prime ideal of an order in a number field.
- `names` – the variable name for the finite field created. Defaults to the name of the number field variable but with bar placed after it.
- `check` – whether or not to check if p is prime.

OUTPUT:

- The residue field at the prime p.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3-7)
sage: P = K.ideal(29).factor()[0][0]
sage: ResidueField(P)
Residue field in abar of Fractional ideal (2*a^2 + 3*a - 10)
```

The result is cached:

```
sage: ResidueField(P) is ResidueField(P)
True
sage: k = K.residue_field(P); k
Residue field in abar of Fractional ideal (2*a^2 + 3*a - 10)
sage: k.order()
841
```

It also works for polynomial rings:

```
sage: R.<t> = GF(31)[]
sage: P = R.ideal(t^5 + 2*t + 11)
sage: ResidueField(P)
Residue field in tbar of Principal ideal (t^5 + 2*t + 11) of Univariate Polynomial Ring in t ove

sage: ResidueField(P) is ResidueField(P)
True
sage: k = ResidueField(P); k.order()
28629151
```

An example where the generator of the number field doesn't generate the residue class field:

```
sage: K.<a> = NumberField(x^3-875)
sage: P = K.ideal(5).factor()[0][0]; k = K.residue_field(P); k
Residue field in abar of Fractional ideal (5, 1/25*a^2 - 2/5*a - 1)
sage: k.polynomial()
abar^2 + 3*abar + 4
sage: k.0^3 - 875
2
```

An example where the residue class field is large but of degree 1:

```
sage: K.<a> = NumberField(x^3-875); P = K.ideal(2007).factor()[2][0]; k = K.residue_field(P); k
Residue field of Fractional ideal (223, 1/5*a + 11)
sage: k(a)
168
sage: k(a)^3 - 875
0
```

And for polynomial rings:

```
sage: R.<t> = GF(next_prime(2^18))[]
sage: P = R.ideal(t - 5)
sage: k = ResidueField(P); k
Residue field of Principal ideal (t + 262142) of Univariate Polynomial Ring in t over Finite Fie
sage: k(t)
5
```

In this example, 2 is an inessential discriminant divisor, so divides the index of `ZZ[a]` in the maximal order for all `a`:

```
sage: K.<a> = NumberField(x^3 + x^2 - 2*x + 8); P = K.ideal(2).factor()[0][0]; P
Fractional ideal (1/2*a^2 - 1/2*a + 1)
sage: F = K.residue_field(P); F
Residue field of Fractional ideal (1/2*a^2 - 1/2*a + 1)
sage: F(a)
0
sage: B = K.maximal_order().basis(); B
[1, 1/2*a^2 + 1/2*a, a^2]
sage: F(B[1])
1
sage: F(B[2])
0
sage: F
Residue field of Fractional ideal (1/2*a^2 - 1/2*a + 1)
sage: F.degree()
1
```

TESTS:

```
sage: K.<a> = NumberField(polygen(QQ))
sage: K.residue_field(K.ideal(3))
Residue field of Fractional ideal (3)
```

create_key_and_extra_args (*p*, *names=None*, *check=True*, *impl=None*, ***kwds*)

Return a tuple containing the key (uniquely defining data) and any extra arguments.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3-7)
sage: ResidueField(K.ideal(29).factor()[0][0]) # indirect doctest
Residue field in abar of Fractional ideal (2*a^2 + 3*a - 10)
```

create_object (*version*, *key*, ***kwds*)

Create the object from the key and extra arguments. This is only called if the object was not found in the cache.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3-7)
sage: P = K.ideal(29).factor()[0][0]
sage: ResidueField(P) is ResidueField(P) # indirect doctest
True
```

class `sage.rings.finite_rings.residue_field.`**ResidueFieldHomomorphism_global**

Bases: `sage.rings.morphism.RingHomomorphism`

The class representing a homomorphism from the order of a number field or function field to the residue field at a given prime.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3-7)
sage: P  = K.ideal(29).factor()[0][0]
sage: k  = K.residue_field(P)
sage: OK = K.maximal_order()
sage: abar = k(OK.1); abar
abar
sage: (1+abar)^179
24*abar + 12

sage: phi = k.coerce_map_from(OK); phi
Ring morphism:
  From: Maximal Order in Number Field in a with defining polynomial x^3 - 7
  To:   Residue field in abar of Fractional ideal (2*a^2 + 3*a - 10)
sage: phi in Hom(OK,k)
True
sage: phi(OK.1)
abar

sage: R.<t> = GF(19)[]; P = R.ideal(t^2 + 5)
sage: k.<a> = R.residue_field(P)
sage: f = k.coerce_map_from(R); f
Ring morphism:
  From: Univariate Polynomial Ring in t over Finite Field of size 19
  To:   Residue field in a of Principal ideal (t^2 + 5) of Univariate Polynomial Ring in t over
```

lift (x)

Returns a lift of x to the Order, returning a "polynomial" in the generator with coefficients between 0 and $p-1$.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3-7)
sage: P = K.ideal(29).factor()[0][0]
sage: k = K.residue_field(P)
sage: OK = K.maximal_order()
sage: f = k.coerce_map_from(OK)
sage: c = OK(a)
sage: b = k(a)
sage: f.lift(13*b + 5)
13*a + 5
sage: f.lift(12821*b+918)
3*a + 19

sage: R.<t> = GF(17)[]; P = R.ideal(t^3 + t^2 + 7)
sage: k.<a> = P.residue_field(); f = k.coerce_map_from(R)
sage: f.lift(a^2 + 5*a + 1)
t^2 + 5*t + 1
sage: f(f.lift(a^2 + 5*a + 1)) == a^2 + 5*a + 1
True
```

section ()

Computes a section of the map, namely a map that lifts elements of the residue field to elements of the ring of integers.

EXAMPLES:

```
sage: K.<a> = NumberField(x^5 - 5*x + 2)
sage: P = K.ideal(47).factor()[0][0]
sage: k = K.residue_field(P)
```

```
sage: f = k.coerce_map_from(K.ring_of_integers())
sage: s = f.section(); s
Lifting map:
  From: Residue field in abar of Fractional ideal (14*a^4 - 24*a^3 - 26*a^2 + 58*a - 15)
  To:   Maximal Order in Number Field in a with defining polynomial x^5 - 5*x + 2
sage: s(k.gen())
a
sage: L.<b> = NumberField(x^5 + 17*x + 1)
sage: P = L.factor(53)[0][0]
sage: l = L.residue_field(P)
sage: g = l.coerce_map_from(L.ring_of_integers())
sage: s = g.section(); s
Lifting map:
  From: Residue field in bbar of Fractional ideal (53, b^2 + 23*b + 8)
  To:   Maximal Order in Number Field in b with defining polynomial x^5 + 17*x + 1
sage: s(l.gen()).parent()
Maximal Order in Number Field in b with defining polynomial x^5 + 17*x + 1

sage: R.<t> = GF(17)[]; P = R.ideal(t^3 + t^2 + 7)
sage: k.<a> = P.residue_field()
sage: f = k.coerce_map_from(R)
sage: f.section()
(map internal to coercion system -- copy before use)
Lifting map:
  From: Residue field in a of Principal ideal (t^3 + t^2 + 7) of Univariate Polynomial Ring
  To:   Univariate Polynomial Ring in t over Finite Field of size 17
```

class `sage.rings.finite_rings.residue_field.`**`ResidueField_generic`**(*p*)

Bases: `sage.rings.ring.Field`

The class representing a generic residue field.

EXAMPLES:

```
sage: I = QQ[i].factor(2)[0][0]; I
Fractional ideal (I + 1)
sage: k = I.residue_field(); k
Residue field of Fractional ideal (I + 1)
sage: type(k)
<class 'sage.rings.finite_rings.residue_field.ResidueFiniteField_prime_modn_with_category'>

sage: R.<t> = GF(29)[]; P = R.ideal(t^2 + 2); k.<a> = ResidueField(P); k
Residue field in a of Principal ideal (t^2 + 2) of Univariate Polynomial Ring in t over Finite F
sage: type(k)
<class 'sage.rings.finite_rings.residue_field.ResidueFiniteField_givaro_with_category'>
```

`ideal`()

Return the maximal ideal that this residue field is the quotient by.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3 + x + 1)
sage: P = K.ideal(29).factor()[0][0]
sage: k = K.residue_field(P) # indirect doctest
sage: k.ideal() is P
True
sage: p = next_prime(2^40); p
1099511627791
sage: k = K.residue_field(K.prime_above(p))
sage: k.ideal().norm() == p
```

```
True

sage: R.<t> = GF(17)[]; P = R.ideal(t^3 + t^2 + 7)
sage: k.<a> = R.residue_field(P)
sage: k.ideal()
Principal ideal (t^3 + t^2 + 7) of Univariate Polynomial Ring in t over Finite Field of size
```

lift (x)

Returns a lift of x to the Order, returning a "polynomial" in the generator with coefficients between 0 and $p-1$.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3-7)
sage: P = K.ideal(29).factor()[0][0]
sage: k =K.residue_field(P)
sage: OK = K.maximal_order()
sage: c = OK(a)
sage: b = k(a)
sage: k.lift(13*b + 5)
13*a + 5
sage: k.lift(12821*b+918)
3*a + 19

sage: R.<t> = GF(17)[]; P = R.ideal(t^3 + t^2 + 7)
sage: k.<a> = P.residue_field()
sage: k.lift(a^2 + 5)
t^2 + 5
```

lift_map ()

Returns the standard map from this residue field up to the ring of integers lifting the canonical projection.

EXAMPLES:

```
sage: I = QQ[3^(1/3)].factor(5)[1][0]; I
Fractional ideal (-a + 2)
sage: k = I.residue_field(); k
Residue field of Fractional ideal (-a + 2)
sage: f = k.lift_map(); f
Lifting map:
  From: Residue field of Fractional ideal (-a + 2)
  To:   Maximal Order in Number Field in a with defining polynomial x^3 - 3
sage: f.domain()
Residue field of Fractional ideal (-a + 2)
sage: f.codomain()
Maximal Order in Number Field in a with defining polynomial x^3 - 3
sage: f(k.0)
1

sage: R.<t> = GF(17)[]; P = R.ideal(t^3 + t^2 + 7)
sage: k.<a> = P.residue_field()
sage: f = k.lift_map(); f
(map internal to coercion system -- copy before use)
Lifting map:
  From: Residue field in a of Principal ideal (t^3 + t^2 + 7) of Univariate Polynomial Ring
  To:   Univariate Polynomial Ring in t over Finite Field of size 17
sage: f(a^2 + 5)
t^2 + 5
```

reduction_map()

Return the partially defined reduction map from the number field to this residue class field.

EXAMPLES:

```
sage: I = QQ[2^(1/3)].factor(2)[0][0]; I
Fractional ideal (a)
sage: k = I.residue_field(); k
Residue field of Fractional ideal (a)
sage: pi = k.reduction_map(); pi
Partially defined reduction map:
  From: Number Field in a with defining polynomial x^3 - 2
  To:   Residue field of Fractional ideal (a)
sage: pi.domain()
Number Field in a with defining polynomial x^3 - 2
sage: pi.codomain()
Residue field of Fractional ideal (a)

sage: K.<a> = NumberField(x^3 + x^2 - 2*x + 32)
sage: F = K.factor(2)[0][0].residue_field()
sage: F.reduction_map().domain()
Number Field in a with defining polynomial x^3 + x^2 - 2*x + 32
sage: K.<a> = NumberField(x^3 + 128)
sage: F = K.factor(2)[0][0].residue_field()
sage: F.reduction_map().codomain()
Residue field of Fractional ideal (1/4*a)

sage: R.<t> = GF(17)[]; P = R.ideal(t^3 + t^2 + 7)
sage: k.<a> = P.residue_field(); f = k.reduction_map(); f
Partially defined reduction map:
  From: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 17
  To:   Residue field in a of Principal ideal (t^3 + t^2 + 7) of Univariate Polynomial Ring
sage: f(1/t)
12*a^2 + 12*a
```

class sage.rings.finite_rings.residue_field.**ResidueFiniteField_givaro**(*p*, *q*, *name*, *modulus*, *to_vs*, *to_order*, *PB*)

Bases: sage.rings.finite_rings.residue_field.ResidueField_generic, sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro

The class representing residue fields of number fields that have non-prime order strictly less than 2^16.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(x^3-7)
sage: P = K.ideal(29).factor()[0][0]
sage: k =K.residue_field(P)
sage: k.degree()
2
sage: OK = K.maximal_order()
sage: c = OK(a)
sage: b = k(c)
sage: b*c^2
7
sage: b*c
13*abar + 5
```

```
sage: R.<t> = GF(7)[]; P = R.ideal(t^2 + 4)
sage: k.<a> = R.residue_field(P); type(k)
<class 'sage.rings.finite_rings.residue_field.ResidueFiniteField_givaro_with_category'>
sage: k(1/t)
5*a
```

class `sage.rings.finite_rings.residue_field.`**`ResidueFiniteField_ntl_gf2e`**(*q*, *name*, *modulus*, *repr*, *p*, *to_vs*, *to_order*, *PB*)

Bases: `sage.rings.finite_rings.residue_field.ResidueField_generic`, `sage.rings.finite_rings.finite_field_ntl_gf2e.FiniteField_ntl_gf2e`

The class representing residue fields with order a power of 2.

When the order is less than 2^16, givaro is used by default instead.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(x^3-7)
sage: P = K.ideal(29).factor()[0][0]
sage: k =K.residue_field(P)
sage: k.degree()
2
sage: OK = K.maximal_order()
sage: c = OK(a)
sage: b = k(c)
sage: b*c^2
7
sage: b*c
13*abar + 5
```

```
sage: R.<t> = GF(2)[]; P = R.ideal(t^19 + t^5 + t^2 + t + 1)
sage: k.<a> = R.residue_field(P); type(k)
<class 'sage.rings.finite_rings.residue_field.ResidueFiniteField_ntl_gf2e_with_category'>
sage: k(1/t)
a^18 + a^4 + a + 1
sage: k(1/t)*t
1
```

class sage.rings.finite_rings.residue_field.**ResidueFiniteField_pari_ffelt**(*p*, *characteristic*, *name*, *modulus*, *to_vs*, *to_order*, *PB*)

Bases: sage.rings.finite_rings.residue_field.ResidueField_generic, sage.rings.finite_rings.finite_field_pari_ffelt.FiniteField_pari_ffelt

The class representing residue fields of number fields that have non-prime order at least 2^16.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3-7)
sage: P = K.ideal(923478923).factor()[0][0]
sage: k = K.residue_field(P)
sage: k.degree()
2
sage: OK = K.maximal_order()
sage: c = OK(a)
sage: b = k(c)
sage: b+c
2*abar
sage: b*c
664346875*abar + 535606347
sage: k.base_ring()
Finite Field of size 923478923

sage: R.<t> = GF(5)[]; P = R.ideal(4*t^12 + 3*t^11 + 4*t^10 + t^9 + t^8 + 3*t^7 + 2*t^6 + 3*t^4
sage: k.<a> = P.residue_field()
sage: type(k)
<class 'sage.rings.finite_rings.residue_field.ResidueFiniteField_pari_ffelt_with_category'>
sage: k(1/t)
3*a^11 + a^10 + 3*a^9 + 2*a^8 + 2*a^7 + a^6 + 4*a^5 + a^3 + 2*a^2 + a
```

class sage.rings.finite_rings.residue_field.**ResidueFiniteField_prime_modn**(*p*, *name*, *intp*, *to_vs*, *to_order*, *PB*)

Bases: sage.rings.finite_rings.residue_field.ResidueField_generic, sage.rings.finite_rings.finite_field_prime_modn.FiniteField_prime_modn

The class representing residue fields of number fields that have prime order.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(x^3-7)
sage: P = K.ideal(29).factor()[1][0]
sage: k = ResidueField(P)
sage: k
```

```
Residue field of Fractional ideal (a^2 + 2*a + 2)
sage: k.order()
29
sage: OK = K.maximal_order()
sage: c = OK(a)
sage: b = k(a)
sage: k.coerce_map_from(OK)(c)
16
sage: k(4)
4
sage: k(c + 5)
21
sage: b + c
3

sage: R.<t> = GF(7)[]; P = R.ideal(2*t + 3)
sage: k = P.residue_field(); k
Residue field of Principal ideal (t + 5) of Univariate Polynomial Ring in t over Finite Field of
sage: k(t^2)
4
sage: k.order()
7
```

CHAPTER

SEVENTEEN

ALGEBRAIC CLOSURES OF FINITE FIELDS

Let $\mathbf{F}$ be a finite field, and let $\overline{\mathbf{F}}$ be an algebraic closure of $\mathbf{F}$; this is unique up to (non-canonical) isomorphism. For every $n \geq 1$, there is a unique subfield $\mathbf{F}_n$ of $\overline{\mathbf{F}}$ such that $\mathbf{F} \subset \mathbf{F}_n$ and $[\mathbf{F}_n : \mathbf{F}] = n$.

In Sage, algebraic closures of finite fields are implemented using compatible systems of finite fields. The resulting Sage object keeps track of a finite lattice of the subfields $\mathbf{F}_n$ and the embeddings between them. This lattice is extended as necessary.

The Sage class corresponding to $\overline{\mathbf{F}}$ can be constructed from the finite field $\mathbf{F}$ by using the `algebraic_closure()` method.

The Sage class for elements of $\overline{\mathbf{F}}$ is `AlgebraicClosureFiniteFieldElement`. Such an element is represented as an element of one of the $\mathbf{F}_n$. This means that each element $x \in \mathbf{F}$ has infinitely many different representations, one for each n such that x is in $\mathbf{F}_n$.

Note: Only prime finite fields are currently accepted as base fields for algebraic closures. To obtain an algebraic closure of a non-prime finite field $\mathbf{F}$, take an algebraic closure of the prime field of $\mathbf{F}$ and embed $\mathbf{F}$ into this.

Algebraic closures of finite fields are currently implemented using (pseudo-)Conway polynomials; see `AlgebraicClosureFiniteField_pseudo_conway` and the module `conway_polynomials`. Other implementations may be added by creating appropriate subclasses of `AlgebraicClosureFiniteField_generic`.

In the current implementation, algebraic closures do not satisfy the unique parent condition. Moreover, there is no coercion map between different algebraic closures of the same finite field. There is a conceptual reason for this, namely that the definition of pseudo-Conway polynomials only determines an algebraic closure up to *non-unique* isomorphism. This means in particular that different algebraic closures, and their respective elements, never compare equal.

AUTHORS:

- Peter Bruin (August 2013): initial version
- Vincent Delecroix (November 2013): additional methods

`sage.rings.algebraic_closure_finite_field.`**`AlgebraicClosureFiniteField`**(*base_ring*, *name*, *category=None*, *implementation=None*, ***kwds*)

Construct an algebraic closure of a finite field.

The recommended way to use this functionality is by calling the `algebraic_closure()` method of the finite field.

Note: Algebraic closures of finite fields in Sage do not have the unique representation property, because they are not determined up to unique isomorphism by their defining data.

EXAMPLES:

```
sage: from sage.rings.algebraic_closure_finite_field import AlgebraicClosureFiniteField
sage: F = GF(2).algebraic_closure()
sage: F1 = AlgebraicClosureFiniteField(GF(2), 'z')
sage: F1 is F
False
```

In the pseudo-Conway implementation, non-identical instances never compare equal:

```
sage: F1 == F
False
sage: loads(dumps(F)) == F
False
```

This is to ensure that the result of comparing two instances cannot change with time.

class `sage.rings.algebraic_closure_finite_field.`**`AlgebraicClosureFiniteFieldElement`**(*parent*, *value*)

Bases: `sage.structure.element.FieldElement`

Element of an algebraic closure of a finite field.

EXAMPLES:

```
sage: F = GF(3).algebraic_closure()
sage: F.gen(2)
z2
sage: type(F.gen(2))
<class 'sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField_pseudo_conway_with
```

`as_finite_field_element`(*minimal=False*)

Return `self` as a finite field element.

INPUT:

- `minimal` – boolean (default: `False`). If `True`, always return the smallest subfield containing `self`.

OUTPUT:

- a triple (`field`, `element`, `morphism`) where `field` is a finite field, `element` an element of `field` and `morphism` a morphism from `field` to `self.parent()`.

EXAMPLES:

```
sage: F = GF(3).algebraic_closure('t')
sage: t = F.gen(5)
sage: t.as_finite_field_element()
(Finite Field in t5 of size 3^5,
 t5,
 Ring morphism:
  From: Finite Field in t5 of size 3^5
  To:   Algebraic closure of Finite Field of size 3
  Defn: t5 |--> t5)
```

By default, `field` is not necessarily minimal. We can force it to be minimal using the `minimal` option:

```
sage: s = t + 1 - t
sage: s.as_finite_field_element()[0]
Finite Field in t5 of size 3^5
sage: s.as_finite_field_element(minimal=True)[0]
Finite Field of size 3
```

This also works when the element has to be converted between two non-trivial finite subfields (see trac ticket #16509):

```
sage: K = GF(5).algebraic_closure()
sage: z = K.gen(5) - K.gen(5) + K.gen(2)
sage: z.as_finite_field_element(minimal=True)
(Finite Field in z2 of size 5^2, z2, Ring morphism:
   From: Finite Field in z2 of size 5^2
   To:   Algebraic closure of Finite Field of size 5
   Defn: z2 |--> z2)
```

There is currently no automatic conversion between the various subfields:

```
sage: a = K.gen(2) + 1
sage: _,b,_ = a.as_finite_field_element()
sage: K4 = K.subfield(4)[0]
sage: K4(b)
Traceback (most recent call last):
...
TypeError: unable to coerce from a finite field other than the prime
subfield
```

Nevertheless it is possible to use the inclusions that are implemented at the level of the algebraic closure:

```
sage: f = K.inclusion(2,4); f
Ring morphism:
  From: Finite Field in z2 of size 5^2
  To:   Finite Field in z4 of size 5^4
  Defn: z2 |--> z4^3 + z4^2 + z4 + 3
sage: f(b)
z4^3 + z4^2 + z4 + 4
```

change_level(n)

Return a representation of `self` as an element of the subfield of degree n of the parent, if possible.

EXAMPLES:

```
sage: F = GF(3).algebraic_closure()
sage: z = F.gen(4)
sage: (z^10).change_level(6)
2*z6^5 + 2*z6^3 + z6^2 + 2*z6 + 2
sage: z.change_level(6)
Traceback (most recent call last):
...
ValueError: z4 is not in the image of Ring morphism:
  From: Finite Field in z2 of size 3^2
  To:   Finite Field in z4 of size 3^4
  Defn: z2 |--> 2*z4^3 + 2*z4^2 + 1

sage: a = F(1).change_level(3); a
1
sage: a.change_level(2)
1
```

```
sage: F.gen(3).change_level(1)
Traceback (most recent call last):
...
ValueError: z3 is not in the image of Ring morphism:
  From: Finite Field of size 3
  To:   Finite Field in z3 of size 3^3
  Defn: 1 |--> 1
```

is_square()

Return `True` if `self` is a square.

This always returns `True`.

EXAMPLES:

```
sage: F = GF(3).algebraic_closure()
sage: F.gen(2).is_square()
True
```

minimal_polynomial()

Return the minimal polynomial of `self` over the prime field.

EXAMPLES:

```
sage: F = GF(11).algebraic_closure()
sage: F.gen(3).minpoly()
x^3 + 2*x + 9
```

minpoly()

Return the minimal polynomial of `self` over the prime field.

EXAMPLES:

```
sage: F = GF(11).algebraic_closure()
sage: F.gen(3).minpoly()
x^3 + 2*x + 9
```

multiplicative_order()

Return the multiplicative order of `self`.

EXAMPLES:

```
sage: K = GF(7).algebraic_closure()
sage: K.gen(5).multiplicative_order()
16806
sage: (K.gen(1) + K.gen(2) + K.gen(3)).multiplicative_order()
7353
```

nth_root(n)

Return an n-th root of `self`.

EXAMPLES:

```
sage: F = GF(5).algebraic_closure()
sage: t = F.gen(2) + 1
sage: s = t.nth_root(15); s
4*z6^5 + 3*z6^4 + 2*z6^3 + 2*z6^2 + 4
sage: s**15 == t
True
```

Todo

This function could probably be made faster.

pth_power(*k=1*)

Return the p^k-th power of `self`, where p is the characteristic of `self.parent()`.

EXAMPLES:

```
sage: K = GF(13).algebraic_closure('t')
sage: t3 = K.gen(3)
sage: s = 1 + t3 + t3**2
sage: s.pth_power()
10*t3^2 + 6*t3
sage: s.pth_power(2)
2*t3^2 + 6*t3 + 11
sage: s.pth_power(3)
t3^2 + t3 + 1
sage: s.pth_power(3).parent() is K
True
```

pth_root(*k=1*)

Return the unique p^k-th root of `self`, where p is the characteristic of `self.parent()`.

EXAMPLES:

```
sage: K = GF(13).algebraic_closure('t')
sage: t3 = K.gen(3)
sage: s = 1 + t3 + t3**2
sage: s.pth_root()
2*t3^2 + 6*t3 + 11
sage: s.pth_root(2)
10*t3^2 + 6*t3
sage: s.pth_root(3)
t3^2 + t3 + 1
sage: s.pth_root(2).parent() is K
True
```

sqrt()

Return a square root of `self`.

EXAMPLES:

```
sage: F = GF(3).algebraic_closure()
sage: F.gen(2).sqrt()
z4^3 + z4 + 1
```

class `sage.rings.algebraic_closure_finite_field.`**AlgebraicClosureFiniteField_generic**(*base_ring*, *name*, *category=None*)

Bases: `sage.rings.ring.Field`

Algebraic closure of a finite field.

Element

alias of `AlgebraicClosureFiniteFieldElement`

algebraic_closure()

Return an algebraic closure of `self`.

This always returns `self`.

EXAMPLES:

```
sage: from sage.rings.algebraic_closure_finite_field import AlgebraicClosureFiniteField
sage: F = AlgebraicClosureFiniteField(GF(5), 'z')
sage: F.algebraic_closure() is F
True
```

cardinality()

Return the cardinality of `self`.

This always returns `+Infinity`.

Todo

When trac ticket #10963 is merged we should remove that method and set the category to infinite fields (i.e. `Fields().Infinite()`).

EXAMPLES:

```
sage: F = GF(3).algebraic_closure()
sage: F.cardinality()
+Infinity
```

characteristic()

Return the characteristic of `self`.

EXAMPLES:

```
sage: from sage.rings.algebraic_closure_finite_field import AlgebraicClosureFiniteField
sage: p = next_prime(1000)
sage: F = AlgebraicClosureFiniteField(GF(p), 'z')
sage: F.characteristic() == p
True
```

gen(*n*)

Return the n-th generator of `self`.

EXAMPLES:

```
sage: from sage.rings.algebraic_closure_finite_field import AlgebraicClosureFiniteField
sage: F = AlgebraicClosureFiniteField(GF(5), 'z')
sage: F.gen(2)
z2
```

gens()

Return a family of generators of `self`.

OUTPUT:

- a `Family`, indexed by the positive integers, whose n-th element is `self.gen(n)`.

EXAMPLES:

```
sage: from sage.rings.algebraic_closure_finite_field import AlgebraicClosureFiniteField
sage: F = AlgebraicClosureFiniteField(GF(5), 'z')
sage: g = F.gens()
sage: g
Lazy family (<lambda>(i))_{i in Positive integers}
sage: g[3]
z3
```

inclusion (m, n)

Return the canonical inclusion map from the subfield of degree m to the subfield of degree n.

EXAMPLES:

```
sage: F = GF(3).algebraic_closure()
sage: F.inclusion(1, 2)
Ring morphism:
  From: Finite Field of size 3
  To:   Finite Field in z2 of size 3^2
  Defn: 1 |--> 1
sage: F.inclusion(2, 4)
Ring morphism:
  From: Finite Field in z2 of size 3^2
  To:   Finite Field in z4 of size 3^4
  Defn: z2 |--> 2*z4^3 + 2*z4^2 + 1
```

is_finite ()

Returns `False` as an algebraically closed field is always infinite.

Todo

When trac ticket #10963 is merged we should remove that method and set the category to infinite fields (i.e. `Fields().Infinite()`).

EXAMPLES:

```
sage: GF(3).algebraic_closure().is_finite()
False
```

ngens ()

Return the number of generators of `self`, which is infinity.

EXAMPLES:

```
sage: from sage.rings.algebraic_closure_finite_field import AlgebraicClosureFiniteField
sage: AlgebraicClosureFiniteField(GF(5), 'z').ngens()
+Infinity
```

some_elements ()

Return some elements of this field.

EXAMPLES:

```
sage: F = GF(7).algebraic_closure()
sage: F.some_elements()
(1, z2, z3 + 1)
```

subfield (n)

Return the unique subfield of degree n of `self` together with its canonical embedding into `self`.

EXAMPLES:

```
sage: F = GF(3).algebraic_closure()
sage: F.subfield(1)
(Finite Field of size 3,
 Ring morphism:
   From: Finite Field of size 3
   To:   Algebraic closure of Finite Field of size 3
   Defn: 1 |--> 1)
sage: F.subfield(4)
```

```
(Finite Field in z4 of size 3^4,
 Ring morphism:
   From: Finite Field in z4 of size 3^4
   To:   Algebraic closure of Finite Field of size 3
   Defn: z4 |--> z4)
```

class sage.rings.algebraic_closure_finite_field.**AlgebraicClosureFiniteField_pseudo_conway**(*base, name, category, lattice, use_*

Bases: sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField_generic, sage.misc.fast_methods.WithEqualityById

Algebraic closure of a finite field, constructed using pseudo-Conway polynomials.

EXAMPLES:

```
sage: F = GF(5).algebraic_closure(implementation='pseudo_conway')
sage: F.cardinality()
+Infinity
sage: F.algebraic_closure() is F
True
sage: x = F(3).nth_root(12); x
z4^3 + z4^2 + 4*z4
sage: x**12
3
```

TESTS:

```
sage: F3 = GF(3).algebraic_closure()
sage: F3 == F3
True
sage: F5 = GF(5).algebraic_closure()
sage: F3 == F5
False
```

CHAPTER EIGHTEEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

[HL99] L. Heath and N. Loehr (1999). New algorithms for generating Conway polynomials over finite fields. Proceedings of the tenth annual ACM-SIAM symposium on discrete algorithms, pp. 429-437.

[CP] Wikipedia entry on Conway polynomials, Wikipedia article Conway_polynomial_(finite_fields)

r

INDEX

A

C

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z