Finite Rings

Release 10.3

The Sage Development Team

Mar 20, 2024
1.1 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 $\mathbb{Z}/p\mathbb{Z}$ and $\mathbb{F}_p$. In particular, there is a canonical map to $\mathbb{F}_p$, 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 $\mathbb{Z}/3\mathbb{Z}$:

```
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)
- 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: UniqueFactory

Return the quotient ring \( \mathbb{Z}/n\mathbb{Z} \).

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
- category (optional) - the category that the quotient ring belongs to.

**Note:** The optional argument *is_field* is not part of the cache key. Hence, this factory will create precisely one instance of \( \mathbb{Z}/n\mathbb{Z} \). 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 \( \mathbb{Z}/n\mathbb{Z} \) into the category of fields may have consequences that can compromise a whole Sage session, so that a restart will be needed.

**EXAMPLES:**

```python
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`.

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

**Note:** Testing whether a quotient ring \( \mathbb{Z}/n\mathbb{Z} \) 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:**

```python
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 enumerated fields
```

(continues on next page)
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:**

```python
sage: R = IntegerModRing(33, 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:

```python
sage: R.is_field(proof=True)
Traceback (most recent call last):
  ... ValueError: THIS SAGE SESSION MIGHT BE SERIOUSLY COMPROMISED!
The order 33 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 probabilistic primality test has failed.
In the latter case, please inform the developers.
```

However, the mistaken assignment is not automatically corrected:

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

To avoid side-effects of this test on other tests, we clear the cache of the ring factory:

```python
sage: IntegerModRing._cache.clear()
```

**create_key_and_extra_args** (`order=0, is_field=False, category=None`)

An integer mod ring is specified uniquely by its order.

**EXAMPLES:**

```python
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:**

1.1. *Ring* $\mathbb{Z}/n\mathbb{Z}$ of integers modulo $n$
```
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: QuotientRing_generic, IntegerModRing

The ring of integers modulo \( N \).

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: # needs sage.groups
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:
By [github issue #15229](https://github.com/sagemath/sage/issues/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:

```python
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.

```python
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: # needs sage.groups
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()
(continues on next page)
```
2
\begin{verbatim}
sage: b.multiplicative_order()
sage: TestSuite(Z16).run()
\end{verbatim}

Saving and loading:
\begin{verbatim}
sage: R = Integers(100000)
sage: TestSuite(R).run()  # long time (17s on sage.math, 2011)
\end{verbatim}

Testing ideals and quotients:
\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
sage: R = IntegerModRing(97)
sage: a = R(5)
sage: a**(10^62)
61
\end{verbatim}

\textbf{cardinality()}

Return the cardinality of this ring.

\textbf{EXAMPLES:}
\begin{verbatim}
sage: 2mod(87).cardinality()
87
\end{verbatim}

\textbf{characteristic()}

\textbf{EXAMPLES:}
\begin{verbatim}
sage: R = IntegerModRing(18)
sage: FF = IntegerModRing(17)
sage: FF.characteristic()
17
sage: R.characteristic()
18
\end{verbatim}

\textbf{degree()}

Return 1.

\textbf{EXAMPLES:}
\begin{verbatim}
sage: R = Integers(12345678900)
sage: R.degree()
1
\end{verbatim}
extension (poly, name=None, names=None, **kwds)

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 \( \mathbb{Z}/p^n\mathbb{Z} \) 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 [github issue #15229](https://github.com/sagemath/sage/issues/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 enumerated fields and Category of subquotients of monoids and Category of quotients of semigroups
```

It is possible to mistakenly put \( \mathbb{Z}/n\mathbb{Z} \) 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 probabilistic primality test has failed. In the latter case, please inform the developers.
```

To avoid side-effects of this test on other tests, we clear the cache of the ring factory:

```
sage: IntegerModRing._cache.clear()
```

`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() #-- needs sage.libs.pari
False
```

`is_noetherian()`
Check if `self` is a Noetherian ring.
EXAMPLES:

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

**is_prime_field()**

Return `True` if the order is prime.

EXAMPLES:

```python
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:

```python
sage: Integers(389).is_unique_factorization_domain()
True
sage: Integers(389^2).is_unique_factorization_domain()  # needs sage.libs.pari
False
```

**krull_dimension()**

Return the Krull dimension of `self`.

EXAMPLES:

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

**list_of_elements_of_multiplicative_group()**

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

EXAMPLES:

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

```python
sage: # needs sage.groups sage.libs.pari
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:**

```python
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 github issue #5250 is fixed:

```
sage: Integers(162).multiplicative_group_is_cyclic()  # needs sage.libs.pari
True
```

**multiplicative_subgroups()**

Return generators for each subgroup of \((\mathbb{Z}/n\mathbb{Z})^*\).

**EXAMPLES:**

```
sage: # optional - gap_package_polycyclic, needs sage.groups
sage: Integers(5).multiplicative_subgroups()
((2,), (4,), ())
sage: Integers(15).multiplicative_subgroups()
((11, 7), (11, 4), (2,), (11,), (14,), (7,), (4,), ())
sage: Integers(2).multiplicative_subgroups()
(())
sage: len(Integers(341).multiplicative_subgroups())
80
```

**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()  # needs sage.libs.pari
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 \([-\text{bound}, \text{bound}]\) into this ring.

**EXAMPLES:**

```
sage: R = IntegerModRing(18)
sage: R.random_element().parent() is R
True
sage: found = [False]*18
sage: while not all(found):
....:     found[R.random_element()] = True
We test bound-option:
```

1.1. Ring \(\mathbb{Z}/n\mathbb{Z}\) of integers modulo \(n\)
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: # needs sage.libs.pari
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 \((\mathbb{Z}/N\mathbb{Z})^*\).

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 \(\geq 3\).

OUTPUT:
A tuple containing the units of self.

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: R.unit_gens()  # needs sage.groups
(11,)
sage: R = IntegerModRing(17)
sage: R.unit_gens()  # needs sage.groups
(3,)
sage: IntegerModRing(next_prime(10^30)).unit_gens()  # needs sage.groups
(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')
(continues on next page)
needs sage.groups

(12, 46)
sage: A.unit_gens(algorithm='pari')
# needs sage.groups sage.libs.pari
(2, 21)

\[ \textbf{unit\_group}(\texttt{algorithm}=\textit{\textquotesingle}sage\textquotesingle}) \]

Return the unit group of \texttt{self}.

INPUT:

\begin{itemize}
\item \texttt{self} – the ring \(\mathbb{Z}/n\mathbb{Z}\) for a positive integer \(n\)
\item \texttt{algorithm} – either 'sage' (default) or 'pari'
\end{itemize}

OUTPUT:

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

\begin{itemize}
\item If \texttt{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 \(n\)).
\item If \texttt{algorithm} == 'pari', the generators are chosen such that their orders form a decreasing sequence with respect to divisibility.
\end{itemize}

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:

\begin{verbatim}
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)
\end{verbatim}

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

\begin{verbatim}
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()
(11, 13)
\end{verbatim}
In the following examples, the cyclic factors are not even isomorphic:

```python
sage: A = Zmod(319)
sage: A.unit_group()  # needs sage.groups
Multiplicative Abelian group isomorphic to C10 x C28
sage: A.unit_group(algorithm='pari')  # needs sage.groups sage.libs.pari
Multiplicative Abelian group isomorphic to C2 x C16777216 x C3188646 x C156 x C62500 x C2058 x C110 x C156 x C2 x C16 x C2 x C18 x C22 x C2
```

**unit_group_exponent()**

`unit_group_exponent()` returns the exponent of the unit group of a residue class ring. Here are some examples:

```python
sage: R = IntegerModRing(17)
sage: R.unit_group_exponent()  # needs sage.groups
16
```

**unit_group_order()**

`unit_group_order()` returns the order of the unit group of a residue class ring. Here are some examples:

```python
sage: R = IntegerModRing(18)
sage: R.unit_group_order()  # needs sage.groups
6
```
Finite Rings, Release 10.3

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:

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

1.2 Elements of $\mathbb{Z}/n\mathbb{Z}$

An element of the integers modulo $n$.

There are three types of integer_mod classes, depending on the size of the modulus.

- `IntegerMod_int` stores its value in a `int_fast32_t` (typically an int); this is used if the modulus is less than $\sqrt{2^{31}} - 1$.
- `IntegerMod_int64` stores its value in a `int_fast64_t` (typically a `long long`); this is used if the modulus is less than $2^{31} - 1$. In many places, we assume that the values and the modulus actually fit inside an `unsigned long`.
- `IntegerMod_gmp` stores its value in a `mpz_t`; this can be used for an arbitrarily large modulus.

All extend `IntegerMod_abstract`.

For efficiency reasons, it stores the modulus (in all three forms, if possible) in a common (cdef) class `NativeIntStruct` rather than in the parent.

AUTHORS:

- Robert Bradshaw: most of the work
- Didier Deshommes: bit shifting
- William Stein: editing and polishing; new arith architecture
- Robert Bradshaw: implement native is_square and square_root
- William Stein: sqrt
- Maarten Derickx: moved the valuation code from the global valuation function to here

```python
class sage.rings.finite_rings.integer_mod.Int_to_IntegerMod
    Bases: IntegerMod_hom

EXAMPLES:

We make sure it works for every type.
```

(continues on next page)
sage.rings.finite_rings.integer_mod.IntegerMod\( (\text{parent}, \text{value}) \)

Create an integer modulo \( n \) with the given parent.

EXAMPLES:

```python
sage: from sage.rings.finite_rings.integer_mod import IntegerMod
sage: R = IntegerModRing(100)
sage: type(R._pyx_order.table)
<class 'list'>
sage: IntegerMod(R, 42)
42
sage: IntegerMod(R, 142)
42
sage: IntegerMod(R, 10**100 + 42)
42
sage: IntegerMod(R, -9158)
42
```

class sage.rings.finite_rings.integer_mod.IntegerMod_abstract

Bases: \texttt{FiniteRingElement}

EXAMPLES:

```python
sage: a = Mod(10, 30**10); a
10
sage: type(a)
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_gmp'>
sage: loads(a.dumps()) == a
True
```

additive_order()

Returns the additive order of self.

This is the same as \texttt{self.order()}.

EXAMPLES:

```python
sage: Integers(20)(2).additive_order()
10
sage: Integers(20)(7).additive_order()
20
sage: Integers(90308402384902)(2).additive_order()
45154201192451
```

charpoly(\(var=x\))

Returns the characteristic polynomial of this element.

EXAMPLES:
AUTHORS:

- Craig Citro

\texttt{crt (other)}

Use the Chinese Remainder Theorem to find an element of the integers modulo the product of the moduli that reduces to \texttt{self} and to \texttt{other}. The modulus of \texttt{other} must be coprime to the modulus of \texttt{self}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: a = mod(3, 5)
sage: b = mod(2, 7)
sage: a.crt(b)
23
sage: a = mod(37, 10^8)
sage: b = mod(9, 3^8)
sage: a.crt(b)
125900000037

sage: b = mod(0, 1)
sage: a.crt(b) == a
True
sage: a.crt(b).modulus()
100000000
\end{verbatim}

AUTHORS:

- Robert Bradshaw

\texttt{divides (other)}

Test whether \texttt{self} divides \texttt{other}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: R = Zmod(6)
sage: R(2).divides(R(4))
True
sage: R(4).divides(R(2))
True
sage: R(2).divides(R(3))
False
\end{verbatim}

\texttt{generalised_log ()}

Return integers $[n_1, \ldots, n_d]$ such that

$$\prod_{i=1}^{d} x_i^{n_i} = \text{self},$$

where $x_1, \ldots, x_d$ are the generators of the unit group returned by \texttt{self.parent().unit_gens()}.

\textbf{EXAMPLES:}
sage: m = Mod(3, 1568)
sage: v = m.generalised_log(); v
[1, 3, 1]
sage: prod([Zmod(1568).unit_gens()[i] ** v[i] for i in [0..2]])
3

Warning: The output is given relative to the set of generators obtained by passing
algorithm='sage' to the method \texttt{unit\_gens()} of the parent (which is the default). Specifying
algorithm='pari' usually yields a different set of generators that is incompatible with this method.

\textbf{is\_nilpotent()}\newline
Returns True if self is nilpotent, i.e., some power of self is zero.

\textbf{is\_one()}\newline
\textbf{is\_primitive\_root()}\newline
Determines whether this element generates the group of units modulo n.

EXAMPLES:
sage: mod(3, 98).is_primitive_root()  # needs sage.libs.pari
True
sage: mod(11, 1009^2).is_primitive_root()  # needs sage.libs.pari
True

is_square()

EXAMPLES:

sage: Mod(3, 17).is_square()
False
sage: # needs sage.libs.pari
sage: Mod(9, 17).is_square()
True
sage: Mod(9, 17*19^2).is_square()
True
sage: Mod(-1, 17^30).is_square()
True
sage: Mod(1/9, next_prime(2^40)).is_square()
True
sage: Mod(1/25, next_prime(2^90)).is_square()
True

ALGORITHM: Calculate the Jacobi symbol \((\text{self}/p)\) at each prime \(p\) dividing \(n\). It must be 1 or 0 for each prime, and if it is 0 mod \(p\), where \(p^k||n\), then \(ord_p(\text{self})\) must be even or greater than \(k\).

The case \(p = 2\) is handled separately.

AUTHORS:
  • Robert Bradshaw

is_unit()

lift_centered()

Lift \text{self} to a centered congruent integer.

OUTPUT:

The unique integer \(i\) such that \(-n/2 < i \leq n/2\) and \(i \equiv \text{self} \mod n\) (where \(n\) denotes the modulus).

EXAMPLES:

sage: Mod(0,5).lift_centered()
0
sage: Mod(1,5).lift_centered()
1
sage: Mod(2,5).lift_centered()
2
sage: Mod(3,5).lift_centered()
-2
sage: Mod(4,5).lift_centered()
-1
sage: Mod(50,100).lift_centered()
50
sage: Mod(51,100).lift_centered()
(continues on next page)
Compute the discrete logarithm of this element to base \( b \), that is, return an integer \( x \) such that \( b^x = a \), where \( a \) is self.

**INPUT:**

- self - unit modulo \( n \)
- \( b \) - a unit modulo \( n \). If \( b \) is not given, \( R.multiplicative_generator() \) is used, where \( R \) is the parent of self.

**OUTPUT:**

Integer \( x \) such that \( b^x = a \), if this exists; a `ValueError` otherwise.

**Note:** The algorithm first factors the modulus, then invokes Pari’s `pari:znlog` function for each odd prime power in the factorization of the modulus. This method can be quite slow for large moduli.

**EXAMPLES:**

```python
sage: # needs sage.libs.pari sage.modules
sage: r = Integers(125)
sage: b = r.multiplicative_generator()^3
sage: a = b^17
sage: a.log(b)
17
sage: a.log()
51
```

A bigger example:

```python
sage: # needs sage.rings.finite_rings
sage: FF = FiniteField(2^32 + 61)
sage: c = FF(4294967356)
sage: x = FF(2)
sage: a = c.log(x)
sage: a
2147483678
sage: a^x
4294967356
```

An example with a highly composite modulus:

```python
sage: m = 2^99 * 77^7 * 123456789 * 1371292357615486607^2
sage: (Mod(5,m)^5735816763073854953388147237921).log(5)  # needs sage.libs.pari
5735816763073854953388147237921
```

Errors are generated if the logarithm doesn’t exist or the inputs are not units:

```python
sage: Mod(3, 7).log(Mod(2, 7))  # needs sage.libs.pari
```

(continues on next page)
AUTHORS:

- David Joyner and William Stein (2005-11)
- Simon King (2010-07-07): fix a side effect on PARI
- Lorenz Panny (2021): speedups for composite moduli

minimal_polynomial (var='x')

Returns the minimal polynomial of this element.

EXAMPLES:

sage: GF(241, 'a')(1).minimal_polynomial(var = 'z')
z + 240

minpoly (var='x')

Returns the minimal polynomial of this element.

EXAMPLES:

sage: GF(241, 'a')(1).minpoly()
x + 240

modulus ()

EXAMPLES:

sage: Mod(3,17).modulus()
17

multiplicative_order ()

Returns the multiplicative order of self.

EXAMPLES:

sage: Mod(-1, 5).multiplicative_order()  #...

needs sage.libs.pari
2
sage: Mod(1, 5).multiplicative_order()  #...

needs sage.libs.pari
1
sage: Mod(0, 5).multiplicative_order()  #...

needs sage.libs.pari
Traceback (most recent call last):
...
ArithmeticError: multiplicative order of 0 not defined since it is not a unit modulo 5

1.2. Elements of \( \mathbb{Z}/n\mathbb{Z} \)
norm()

Returns the norm of this element, which is itself. (This is here for compatibility with higher order finite fields.)

EXAMPLES:

```plaintext
sage: k = GF(691)
sage: a = k(389)
sage: a.norm()
389
```

AUTHORS:

• Craig Citro

nth_root(n, extend=False, all=False, algorithm=None, cunningham=False)

Returns an nth root of self.

INPUT:

• n - integer ≥ 1
• extend - bool (default: True); if True, return an nth 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 nth roots of self, instead of just one.
• algorithm - string (default: None); The algorithm for the prime modulus case. CRT and p-adic log techniques are used to reduce to this case. ‘Johnston’ is the only currently supported option.
• cunningham - bool (default: False); In some cases, factorization of n is computed. If cunningham is set to True, the factorization of n is computed using trial division for all primes in the so called Cunningham table. Refer to sage.rings.factorint.factor_cunningham for more information. You need to install an optional package to use this method, this can be done with the following command line sage -i cunningham_tables

OUTPUT:

If self has an nth root, returns one (if all is False) or a list of all of them (if all is True). Otherwise, raises a ValueError (if extend is False) or a NotImplementedError (if extend is True).

**Warning:** The ‘extend’ option is not implemented (yet).

NOTE:

• If n = 0:
  – if all=True:
    * if self=1: all nonzero elements of the parent are returned in a list. Note that this could be very expensive for large parents.
    * otherwise: an empty list is returned
  – if all=False:
    * if self=1: self is returned
    * otherwise; a ValueError is raised
  – If n < 0:
    – if self is invertible, the (−n)th root of the inverse of self is returned
- otherwise a `ValueError` is raised or empty list returned.

EXAMPLES:

```python
sage: K = GF(31)
sage: a = K(22)
sage: K(22).nth_root(7)
13
sage: K(25).nth_root(5)
5
sage: K(23).nth_root(3)
29

sage: # needs sage.rings.padics
sage: mod(225, 2^5*3^2).nth_root(4, all=True)
[225, 129, 33, 63, 255, 159, 9, 201, 105, 279, 183, 87, 81, 273, 177, 207, 111, 15, 153, 57, 249, 135, 39, 231]
sage: mod(275, 2^5*7^4).nth_root(7, all=True)
[58235, 25307, 69211, 36283, 3355, 47259, 14331]
sage: mod(1,8).nth_root(2, all=True)
[1, 7, 5, 3]
sage: mod(4,8).nth_root(2, all=True)
[2, 6]
sage: mod(1,16).nth_root(4, all=True)
[1, 15, 13, 3, 9, 7, 5, 11]
sage: (mod(22,31)^200).nth_root(200)
5
sage: mod(3,6).nth_root(0, all=True)
[]
sage: mod(3,6).nth_root(0)
Traceback (most recent call last):
  ... ValueError
sage: mod(1,6).nth_root(0, all=True)
[1, 2, 3, 4, 5]
```

ALGORITHM:
The default for prime modulus is currently an algorithm described in [Joh1999].

AUTHORS:

- David Roe (2010-02-13)

`polynomial(var='x')`

Returns a constant polynomial representing this value.

EXAMPLES:

```python
sage: k = GF(7)
sage: a = k.gen(); a
1
sage: a.polynomial()
1
sage: type(a.polynomial())
<...>
# needs sage.libs.flint
<class 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>
```

`rational_reconstruction()`
Use rational reconstruction to try to find a lift of this element to the rational numbers.

EXAMPLES:

```
sage: R = IntegerModRing(97)
sage: a = R(2) / R(3)
sage: a
33
sage: a.rational_reconstruction()
2/3
```

This method is also inherited by prime finite fields elements:

```
sage: k = GF(97)
```

```
sage: a = k(RationalField()('2/3'))
sage: a
33
sage: a.rational_reconstruction()
2/3
```

```
sqrt (extend=True, all=False)
```

Return square root or square roots of self modulo n.

INPUT:

- extend - bool (default: True); if True, return a square root in an extension ring, if necessary. Otherwise, raise a ValueError if the square root is not in the base ring.
- all - bool (default: False); if True, return [all] square roots of self, instead of just one.

ALGORITHM: Calculates the square roots mod p for each of the primes p dividing the order of the ring, then lifts them p-adically and uses the CRT to find a square root mod n.

See also square_root_mod_prime_power() and square_root_mod_prime() for more algorithmic details.

EXAMPLES:

```
sage: mod(-1, 17).sqrt() 4
sage: mod(5, 389).sqrt() 86
sage: mod(7, 18).sqrt() 5
```

```
sage: # needs sage.libs.pari
sage: a = mod(14, 5^60).sqrt()
sage: a*a
14
sage: mod(15, 389).sqrt(extend=False)
Traceback (most recent call last):
  ...
ValueError: self must be a square
```

```
sage: Mod(1/9, next_prime(2^40)).sqrt()^(-2) 9
```

```
sage: Mod(1/25, next_prime(2^90)).sqrt()^(-2) 25
```
sage: a = Mod(3, 5); a
3
sage: x = Mod(-1, 360)
sage: x.sqrt(extend=False)
Traceback (most recent call last):
  ...  
ValueError: self must be a square
sage: y = x.sqrt(); y
sqrt359
sage: y.parent()
Univariate Quotient Polynomial Ring in sqrt359 over
   Ring of integers modulo 360 with modulus x^2 + 1
sage: y^2
359

We compute all square roots in several cases:

sage: R = Integers(5*2^3*3^2); R
Ring of integers modulo 360
sage: R(40).sqrt(all=True)
[20, 160, 200, 340]
sage: [x for x in R if x^2 == 40]  # Brute force verification
[20, 160, 200, 340]
sage: R(1).sqrt(all=True)
[1, 19, 71, 89, 109, 161, 179, 181, 199, 251, 269, 271, 289, 341, 359]
sage: R(0).sqrt(all=True)
[0, 60, 120, 180, 240, 300]

sage: # needs sage.libs.pari
sage: R = Integers(5*13^3*37); R
Ring of integers modulo 406445
sage: v = R(-1).sqrt(all=True); v
[78853, 111808, 160142, 193097, 213348, 246303, 294637, 327592]
sage: [x^2 for x in v]
[406444, 406444, 406444, 406444, 406444, 406444, 406444, 406444]
sage: v = R(169).sqrt(all=True); min(v), -max(v), len(v)
(13, 13, 104)
sage: all(x^2 == 169 for x in v)
True

sage: # needs sage.rings.finite_rings
sage: t = FiniteField(next_prime(2^100))(4)
sage: t.sqrt(extend=False, all=True)
[2, 126765060028229401496703205651]
sage: t = FiniteField(next_prime(2^100))(2)
sage: t.sqrt(extend=False, all=True)
[]

Modulo a power of 2:

sage: R = Integers(2^7); R
Ring of integers modulo 128
sage: a = R(17)
sage: a.sqrt()
23
sage: a.sqrt(all=True)
[23, 41, 87, 105]

(continues on next page)
sage: [x for x in R if x^2==17]
[23, 41, 87, 105]

\textbf{sage:} square_root ($extend=True$, $all=False$)

Return square root or square roots of self modulo $n$.

INPUT:

- extend - bool (default: True); if True, return a square root in an extension ring, if necessary. Otherwise, raise a \texttt{ValueError} if the square root is not in the base ring.

- all - bool (default: False); if True, return \{all\} square roots of self, instead of just one.

ALGORITHM: Calculates the square roots mod $p$ for each of the primes $p$ dividing the order of the ring, then lifts them $p$-adically and uses the CRT to find a square root mod $n$.

See also \texttt{square_root_mod_prime_power()} and \texttt{square_root_mod_prime()} for more algorithmic details.

EXAMPLES:

```sage
sage: mod(-1, 17).sqrt() 4
sage: mod(5, 389).sqrt() 86
sage: mod(7, 18).sqrt() 5

sage: # needs sage.libs.pari
sage: a = mod(14, 5^60).sqrt()
sage: a*a
14

sage: mod(15, 389).sqrt(extend=False) Traceback (most recent call last):
... ValueError: self must be a square
sage: Mod(1/9, next_prime(2^40)).sqrt()^(-2) 9
sage: Mod(1/25, next_prime(2^90)).sqrt()^(-2) 25

sage: a = Mod(3, 5); a
3
sage: x = Mod(-1, 360)
sage: x.sqrt(extend=False) Traceback (most recent call last):
... ValueError: self must be a square
sage: y = x.sqrt(); y
sqrt359
sage: y.parent() Univariate Quotient Polynomial Ring in sqrt359 over
  Ring of integers modulo 360 with modulus x^2 + 1
sage: y^2
359
```

We compute all square roots in several cases:
sage: R = Integers(5*2^3*3^2); R
Ring of integers modulo 360
sage: R(40).sqrt(all=True)
[20, 160, 200, 340]
sage: [x for x in R if x^2 == 40]  # Brute force verification
[20, 160, 200, 340]
sage: R(1).sqrt(all=True)
[1, 19, 71, 89, 91, 109, 161, 179, 181, 199, 251, 269, 271, 289, 341, 359]
sage: R(0).sqrt(all=True)
[0, 60, 120, 180, 240, 300]

sage: # needs sage.libs.pari
sage: R = Integers(5*13^3*37); R
Ring of integers modulo 406445
sage: v = R(-1).sqrt(all=True); v
[78853, 111808, 160142, 193097, 213348, 246303, 294637, 327592]
sage: [x^2 for x in v]
[406444, 406444, 406444, 406444, 406444, 406444, 406444, 406444]
sage: v = R(169).sqrt(all=True); min(v), -max(v), len(v)
(13, 13, 104)
sage: all(x^2 == 169 for x in v)
True

sage: # needs sage.rings.finite_rings
sage: t = FiniteField(next_prime(2^100))(4)
sage: t.sqrt(extend=False, all=True)
[2, 1267650600228229401496703205651]
sage: t = FiniteField(next_prime(2^100))(2)
sage: t.sqrt(extend=False, all=True)
[]

Modulo a power of 2:

sage: R = Integers(2^7); R
Ring of integers modulo 128
sage: a = R(17)
sage: a.sqrt()
23
sage: a.sqrt(all=True)
[23, 41, 87, 105]

trace()
Returns the trace of this element, which is itself. (This is here for compatibility with higher order finite fields.)

EXAMPLES:

sage: k = GF(691)
sage: a = k(389)
sage: a.trace()
389

AUTHORS:

• Craig Citro
valuation\( (p) \)
The largest power \( r \) such that \( m \) is in the ideal generated by \( p^r \) or infinity if there is not a largest such power. However it is an error to take the valuation with respect to a unit.

**Note:** This is not a valuation in the mathematical sense. As shown with the examples below.

**EXAMPLES:**
This example shows that \((a*b)\text{.valuation}(n)\) is not always the same as \(a\text{.valuation}(n) + b\text{.valuation}(n)\)

```sage
R = ZZ.quo(9)
sage: a = R(3)
sage: b = R(6)
sage: a.valuation(3)
1
sage: a.valuation(3) + b.valuation(3)
2
sage: (a*b).valuation(3)
+Infinity
```

The valuation with respect to a unit is an error

```sage
a.valuation(4)
```

```
Traceback (most recent call last):
...
ValueError: Valuation with respect to a unit is not defined.
```

class sage.rings.finite_rings.integer_mod.IntegerMod_gmp

**Bases:** `IntegerMod_abstract`

Elements of \( Z/nZ \) for \( n \) not small enough to be operated on in word size.

**AUTHORS:**
- Robert Bradshaw (2006-08-24)

gcd\( (other) \)
Greatest common divisor
Returns the “smallest” generator in \( Z/NZ \) of the ideal generated by `self` and `other`.

**INPUT:**
- `other` – an element of the same ring as this one.

**EXAMPLES:**

```sage
mod(2^3*3^2*5, 3^3*2^2*17^8).gcd(mod(2^4*3*17, 3^3*2^2*17^8))
```

```
12
```

```sage
mod(0,17^8).gcd(mod(0,17^8))
```

```
0
```

**is_one()**
Returns True if this is 1, otherwise False.

**EXAMPLES:**
Finite Rings, Release 10.3

```python
sage: mod(1, 5^23).is_one()
True
sage: mod(0, 5^23).is_one()
False
```

**is_unit()**

Return True iff this element is a unit.

**EXAMPLES:**

```python
sage: mod(13, 5^23).is_unit()
True
sage: mod(25, 5^23).is_unit()
False
```

**lift()**

Lift an integer modulo \( n \) to the integers.

**EXAMPLES:**

```python
sage: a = Mod(8943, 2^70); type(a)
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_gmp'>
sage: lift(a)
8943
sage: a.lift()
8943
```

**class** `sage.rings.finite_rings.integer_mod.IntegerMod_hom`

Bases: `Morphism`

**class** `sage.rings.finite_rings.integer_mod.IntegerMod_int`

Bases: `IntegerMod_abstract`

Elements of \( \mathbb{Z}/n\mathbb{Z} \) for \( n \) small enough to be operated on in 32 bits

**AUTHORS:**

- Robert Bradshaw (2006-08-24)

**EXAMPLES:**

```python
sage: a = Mod(10,30); a
10
sage: loads(a.dumps()) == a
True
```

**gcd(other)**

Greatest common divisor

Returns the “smallest” generator in \( \mathbb{Z}/N\mathbb{Z} \) of the ideal generated by self and other.

**INPUT:**

- other – an element of the same ring as this one.

**EXAMPLES:**

```python
sage: R = Zmod(60); S = Zmod(72)
sage: a = R(40).gcd(S(30)); a
```

(continues on next page)
sage: a.parent()
Ring of integers modulo 12
sage: b = R(17).gcd(60); b
1
sage: b.parent()
Ring of integers modulo 60
sage: mod(72*5, 3^3*2^2*17^2).gcd(mod(48*17, 3^3*2^2*17^2))
12
sage: mod(0,1).gcd(mod(0,1))
0

is_one()
Returns True if this is 1, otherwise False.

EXAMPLES:

sage: mod(6,5).is_one()
True
sage: mod(0,5).is_one()
False
sage: mod(1, 1).is_one()
True
sage: Zmod(1).one().is_one()
True

is_unit()
Return True if this element is a unit.

EXAMPLES:

sage: a=Mod(23,100)
sage: a.is_unit()
True
sage: a=Mod(24,100)
sage: a.is_unit()
False

lift()
Lift an integer modulo n to the integers.

EXAMPLES:

sage: a = Mod(8943, 2^10); type(a)
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
sage: lift(a)
751
sage: a.lift()
751

sqrt(extend=True, all=False)
Return square root or square roots of self modulo n.

INPUT:

- extend - bool (default: True); if True, return a square root in an extension ring, if necessary. Otherwise, raise a ValueError if the square root is not in the base ring.
• all - bool (default: False); if True, return {all} square roots of self, instead of just one.

ALGORITHM: Calculates the square roots mod \( p \) for each of the primes \( p \) dividing the order of the ring, then lifts them \( p \)-adically and uses the CRT to find a square root mod \( n \).

See also square_root_mod_prime_power() and square_root_mod_prime() for more algorithmic details.

EXAMPLES:

```sage
sage: mod(-1, 17).sqrt()
4
sage: mod(5, 389).sqrt()
86
sage: mod(7, 18).sqrt()
5

sage: # needs sage.libs.pari
sage: a = mod(14, 5^60).sqrt()
14
sage: a*a
Traceback (most recent call last):
... ValueError: self must be a square
sage: Mod(1/9, next_prime(2^40)).sqrt()^(-2)
9
sage: Mod(1/25, next_prime(2^90)).sqrt()^(-2)
25

sage: a = Mod(3,5); a
3
sage: x = Mod(-1, 360)
sage: x.sqrt(extend=False)
Traceback (most recent call last):
... ValueError: self must be a square
sage: y = x.sqrt(); y
sqrt359
sage: y.parent()
Univariate Quotient Polynomial Ring in sqrt359
over Ring of integers modulo 360 with modulus x^2 + 1
sage: y^2
359
```

We compute all square roots in several cases:

```sage
sage: R = Integers(5*2^3*3^2); R
Ring of integers modulo 360
sage: R(40).sqrt(all=True)
[20, 160, 200, 340]
sage: [x for x in R if x^2 == 40] # Brute force verification
[20, 160, 200, 340]
sage: R(1).sqrt(all=True)
[1, 19, 71, 89, 91, 109, 161, 179, 181, 199, 251, 269, 271, 289, 341, 359]
sage: R(0).sqrt(all=True)
[0, 60, 120, 180, 240, 300]
sage: GF(107)(0).sqrt(all=True)
[0]
```

1.2. Elements of \( \mathbb{Z}/n\mathbb{Z} \)
sage: # needs sage.libs.pari
sage: R = Integers(5*13^3*37); R
Ring of integers modulo 406445
sage: v = R(-1).sqrt(all=True); v
[78853, 111808, 160142, 193097, 213348, 246303, 294637, 327592]
sage: [x^2 for x in v]
[406444, 406444, 406444, 406444, 406444, 406444, 406444, 406444]
sage: v = R(169).sqrt(all=True); min(v), -max(v), len(v)
(13, 13, 104)
sage: all(x^2 == 169 for x in v)
True

Modulo a power of 2:

sage: R = Integers(2^7); R
Ring of integers modulo 128
sage: a = R(17)
sage: a.sqrt()
23
sage: a.sqrt(all=True)
[23, 41, 87, 105]
sage: [x for x in R if x^2==17]
[23, 41, 87, 105]

class sage.rings.finite_rings.integer_mod.IntegerMod_int64
Bases: IntegerMod_abstract

Elements of \( \mathbb{Z}/n\mathbb{Z} \) for \( n \) small enough to be operated on in 64 bits

EXAMPLES:

sage: a = Mod(10,3^10); a
10
sage: type(a)
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_int64'>
sage: loads(a.dumps()) == a
True
sage: Mod(5, 2^31)
5

AUTHORS:

• Robert Bradshaw (2006-09-14)

gcd(other)

Greatest common divisor

Returns the “smallest” generator in \( \mathbb{Z}/N\mathbb{Z} \) of the ideal generated by self and other.

INPUT:

• other – an element of the same ring as this one.

EXAMPLES:

sage: mod(2^3*3^2*5, 3^3*2^2*17^5).gcd(mod(2^4*3^17, 3^3*2^2*17^5))
12
sage: mod(0,17^5).gcd(mod(0,17^5))
0
is_one()

Returns True if this is 1, otherwise False.

EXAMPLES:

```
sage: (mod(-1,5^10)^2).is_one()
True
sage: mod(0,5^10).is_one()
False
```

is_unit()

Return True iff this element is a unit.

EXAMPLES:

```
sage: mod(13, 5^10).is_unit()
True
sage: mod(25, 5^10).is_unit()
False
```

lift()

Lift an integer modulo $n$ to the integers.

EXAMPLES:

```
sage: a = Mod(8943, 2^25); type(a)
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_int64'>
sage: lift(a)
8943
sage: a.lift()
8943
```

class sage.rings.finite_rings.integer_mod.IntegerMod_to_Integer

Bases: Map

Map to lift elements to Integer.

EXAMPLES:

```
sage: ZZ.convert_map_from(GF(2))
Lifting map:
  From: Finite Field of size 2
  To:   Integer Ring
```

class sage.rings.finite_rings.integer_mod.IntegerMod_to_IntegerMod

Bases: IntegerMod_hom

Very fast IntegerMod to IntegerMod homomorphism.

EXAMPLES:

```
sage: from sage.rings.finite_rings.integer_mod import IntegerMod_to_IntegerMod
sage: Rs = [Integers(3**k) for k in range(1,30,5)]
sage: [type(R()) for R in Rs]
[<class 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>,
 <class 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>,
 <class 'sage.rings.finite_rings.integer_mod.IntegerMod_int64'>,
 <class 'sage.rings.finite_rings.integer_mod.IntegerMod_int64'>,
 <class 'sage.rings.finite_rings.integer_mod.IntegerMod_gmp'>,
```

(continues on next page)
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_gmp'>]
sage: fs = [IntegerMod_to_IntegerMod(S, R)
....:   for R in Rs for S in Rs if S is not R and S.order() > R.order()]
sage: all(f(-1) == f.codomain()(-1) for f in fs)
True
sage: [f(-1) for f in fs]
[2, 2, 2, 2, 2, 728, 728, 728, 728, 177146, 177146, 177146, 43046720, 43046720,...
úmero104603532022]

**is_injective()**

Return whether this morphism is injective.

**EXAMPLES:**

```
sage: Zmod(4).hom(Zmod(2)).is_injective()
False
```

**is_surjective()**

Return whether this morphism is surjective.

**EXAMPLES:**

```
sage: Zmod(4).hom(Zmod(2)).is_surjective()
True
```

**class sage.rings.finite_rings.integer_mod.Integer_to_IntegerMod**

**Bases:** *IntegerMod_hom*

Fast \( \mathbb{Z} \to \mathbb{Z}/n\mathbb{Z} \) morphism.

**EXAMPLES:**

We make sure it works for every type.

```
sage: from sage.rings.finite_rings.integer_mod import Integer_to_IntegerMod
sage: Rs = [Integers(10), Integers(10^5), Integers(10^10)]
sage: [type(R(0)) for R in Rs]
[<class 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>,
 <class 'sage.rings.finite_rings.integer_mod.IntegerMod_int64'>,
 <class 'sage.rings.finite_rings.integer_mod.IntegerMod_gmp'>]
sage: fs = [Integer_to_IntegerMod(R) for R in Rs]
sage: [f(-1) for f in fs]
[9, 99999, 9999999999]
```

**is_injective()**

Return whether this morphism is injective.

**EXAMPLES:**

```
sage: ZZ.hom(Zmod(2)).is_injective()
False
```

**is_surjective()**

Return whether this morphism is surjective.

**EXAMPLES:**
```
sage: ZZ.hom(Zmod(2)).is_surjective()
True
```

**section()**

sage.rings.finite_rings.integer_mod.Mod(n, m, parent=None)

Return the equivalence class of $n$ modulo $m$ as an element of $\mathbb{Z}/m\mathbb{Z}$.

**EXAMPLES:**

```
sage: x = Mod(12345678, 32098203845329048)
sage: x
12345678
sage: x^100
101732209155072
```

You can also use the lowercase version:

```
sage: mod(12,5)
2
```

Illustrates that `github issue #5971` is fixed. Consider $n$ modulo $m$ when $m = 0$. Then $\mathbb{Z}/0\mathbb{Z}$ is isomorphic to $\mathbb{Z}$ so $n$ modulo 0 is equivalent to $n$ for any integer value of $n$:

```
sage: Mod(10, 0)
10
sage: a = randint(-100, 100)
sage: Mod(a, 0) == a
True
```

class sage.rings.finite_rings.integer_mod.NativeIntStruct

Bases: object

We store the various forms of the modulus here rather than in the parent for efficiency reasons.

We may also store a cached table of all elements of a given ring in this class.

**inverses**

**precompute_table**(parent)

Function to compute and cache all elements of this class.

If inverses == True, also computes and caches the inverses of the invertible elements.

**EXAMPLES:**

```
sage: from sage.rings.finite_rings.integer_mod import NativeIntStruct
sage: R = IntegerModRing(10)
sage: M = NativeIntStruct(R.order())
sage: M.precompute_table(R)
sage: M.table
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: M.inverses
[None, 1, None, 7, None, None, None, 3, None, 9]
```

This is used by the `sage.rings.finite_rings.integer_mod_ring. IntegerModRing_generic` constructor:
sage: from sage.rings.finite_rings.integer_mod_ring import IntegerModRing_generic
sage: R = IntegerModRing_generic(39, cache=False)
sage: R(5)^-1
8
sage: R(5)^-1 is R(8)
False
sage: R = IntegerModRing_generic(39, cache=True)  # indirect doctest
sage: R(5)^-1 is R(8)
True

Check that the inverse of 0 modulo 1 works, see github issue #13639:

sage: R = IntegerModRing_generic(1, cache=True)  # indirect doctest
sage: R(0)^-1 is R(0)
True

.. table

sage.rings.finite_rings.integer_mod.is_IntegerMod(x)

Return True if and only if x is an integer modulo n.

EXAMPLES:

sage: from sage.rings.finite_rings.integer_mod import is_IntegerMod
sage: is_IntegerMod(5)
False
sage: is_IntegerMod(Mod(5,10))
True

sage.rings.finite_rings.integer_mod.lucas(k, P=1, Q=1, n=None)

Return \([V_k(P, Q) \mod n, Q^{\lfloor k/2 \rfloor} \mod n]\) where \(V_k\) is the Lucas function defined by the recursive relation

\[
V_k(P, Q) = PV_{k-1}(P, Q) - QV_{k-2}(P, Q)
\]

with \(V_0 = 2, V_1 = P\).

INPUT:

- k – integer; index to compute
- P, Q – integers or modular integers; initial values
- n – integer (optional); modulus to use if P is not a modular integer

REFERENCES:

- [IEEEP1363]

AUTHORS:

- Somindu Chaya Ramanna, Shashank Singh and Srinivas Vivek Venkatesh (2011-09-15, ECC2011 summer school)
- Robert Bradshaw

EXAMPLES:
Finite Rings, Release 10.3

```python
sage: [lucas(k,4,5,11)[0] for k in range(30)]
[2, 4, 6, 4, 8, 1, 8, 5, 2, 5, 10, 4, 10, 9, 8, 9, 7, 5, 7, 3, 10, 3, 6, 9, 6, 1,...
→7, 1, 2, 3]
sage: lucas(20,4,5,11)
[10, 1]
```

`sage.rings.finite_rings.integer_mod.lucas_q1(mm, P)`

Return $V_k(P,1)$ where $V_k$ is the Lucas function defined by the recursive relation

$$V_k(P,Q) = PV_{k-1}(P,Q) - QV_{k-2}(P,Q)$$

with $V_0 = 2, V_1(P_Q) = P$.

REFERENCES:

- [Pos1988]

AUTHORS:

- Robert Bradshaw

`sage.rings.finite_rings.integer_mod.makeNativeIntStruct` alias of `NativeIntStruct`

`sage.rings.finite_rings.integer_mod.mod(n, m, parent=None)`

Return the equivalence class of $n$ modulo $m$ as an element of $\mathbb{Z}/m\mathbb{Z}$.

EXAMPLES:

```python
sage: x = Mod(12345678, 32098203845329048)
sage: x
12345678
sage: x^100
101732209155072
```

You can also use the lowercase version:

```python
sage: mod(12,5)
2
```

Illustrates that github issue #5971 is fixed. Consider $n$ modulo $m$ when $m = 0$. Then $\mathbb{Z}/0\mathbb{Z}$ is isomorphic to $\mathbb{Z}$ so $n$ modulo 0 is equivalent to $n$ for any integer value of $n$:

```python
sage: Mod(10, 0)
10
sage: a = randint(-100, 100)
sage: Mod(a, 0) == a
True
```

`sage.rings.finite_rings.integer_mod.square_root_mod_prime(a, p=None)`

Calculates the square root of $a$, where $a$ is an integer mod $p$; if $a$ is not a perfect square, this returns an (incorrect) answer without checking.

ALGORITHM: Several cases based on residue class of $p$ mod 16.

- $p$ mod 2 = 0: $p = 2$ so $\sqrt{a} = a$.
- $p$ mod 4 = 3: $\sqrt{a} = a^{(p+1)/4}$.
- $p$ mod 8 = 5: $\sqrt{a} = \zeta ia$ where $\zeta = (2a)^{(p-5)/8}, i = \sqrt{-1}$.

1.2. Elements of $\mathbb{Z}/n\mathbb{Z}$
\begin{itemize}
\item $p \mod 16 = 9$: Similar, work in a bi-quadratic extension of $F_p$ for small $p$, Tonelli and Shanks for large $p$.
\item $p \mod 16 = 1$: Tonelli and Shanks.
\end{itemize}

REFERENCES:
\begin{itemize}
\item [Mul2004]
\item [Atk1992]
\item [Pos1988]
\end{itemize}

AUTHORS:
\begin{itemize}
\item Robert Bradshaw
\end{itemize}

`sage.rings.finite_rings.integer_mod.square_root_mod_prime_power(a, p, e)`

Calculates the square root of $a$, where $a$ is an integer mod $p^e$.

ALGORITHM: Compute $p$-adically by stripping off even powers of $p$ to get a unit and lifting $\sqrt{\text{unit}} \mod p$ via Newton's method whenever $p$ is odd and by a variant of Hensel lifting for $p = 2$.

AUTHORS:
\begin{itemize}
\item Robert Bradshaw
\item Lorenz Panny (2022): polynomial-time algorithm for $p = 2$
\end{itemize}

EXAMPLES:
```
sage: from sage.rings.finite_rings.integer_mod import square_root_mod_prime_power
sage: a = Mod(17, 2^20)
sage: b = square_root_mod_prime_power(a, 2, 20)
sage: b^2 == a
True
```
```
sage: a = Mod(72, 97^10)
sage: b = square_root_mod_prime_power(a, 97, 10)  # needs sage.libs.pari
sage: b^2 == a  # needs sage.libs.pari
True
```
```
sage: mod(100, 5^7).sqrt()^2  # needs sage.libs.pari
100
```
2.1 Finite fields

Sage supports arithmetic in finite prime and extension fields. Several implementation for prime fields are implemented natively in Sage for several sizes of primes $p$. These implementations are

- `sage.rings.finite_rings.integer_mod.IntegerMod_int`,
- `sage.rings.finite_rings.integer_mod.IntegerMod_int64`, and
- `sage.rings.finite_rings.integer_mod.IntegerMod_gmp`.

Small extension fields of cardinality $< 2^{16}$ are implemented using tables of Zech logs via the Givaro C++ library (`sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro`). While this representation is very fast it is limited to finite fields of small cardinality. Larger finite extension fields of order $q > 2^{16}$ are internally represented as polynomials over smaller finite prime fields. If the characteristic of such a field is 2 then NTL is used internally to represent the field (`sage.rings.finite_rings.finite_field_ntl_gf2e.FiniteField_ntl_gf2e`). In all other cases the PARI C library is used (`sage.rings.finite_rings.finite_field_pari_ffelt.FiniteField_pari_ffelt`).

However, this distinction is internal only and the user usually does not have to worry about it because consistency across all implementations is aimed for. In all extension field implementations the user may either specify a minimal polynomial or leave the choice to Sage.

For small finite fields the default choice are Conway polynomials.

The Conway polynomial $C_n$ is the lexicographically first monic irreducible, primitive polynomial of degree $n$ over $GF(p)$ with the property that for a root $\alpha$ of $C_n$ we have that $\beta = \alpha(p^n - 1)/(p^m - 1)$ is a root of $C_m$ for all $m$ dividing $n$. Sage contains a database of Conway polynomials which also can be queried independently of finite field construction.

A pseudo-Conway polynomial satisfies all of the conditions required of a Conway polynomial except the condition that it is lexicographically first. They are therefore not unique. If no variable name is specified for an extension field, Sage will fit the finite field into a compatible lattice of field extensions defined by pseudo-Conway polynomials. This lattice is stored in an `AlgebraicClosureFiniteField` object; different algebraic closure objects can be created by using a different prefix keyword to the finite field constructor.

Note that the computation of pseudo-Conway polynomials is expensive when the degree is large and highly composite. If a variable name is specified then a random polynomial is used instead, which will be much faster to find.

While Sage supports basic arithmetic in finite fields some more advanced features for computing with finite fields are still not implemented. For instance, Sage does not calculate embeddings of finite fields yet.

EXAMPLES:
Finite Fields support iteration, starting with 0.

```sage
define k = GF(9, 'a')
define for i, x in enumerate(k): print("{} {}".format(i, x))
0 0
 1 a
 2 a + 1
 3 2*a + 1
 4 2
 5 2*a
 6 2*a + 2
 7 a + 2
 8 1
```

We output the base rings of several finite fields.

```sage
define k = GF(3); type(k)
define k.base_ring()
Finite Field of size 3
```

```sage
# needs sage.libs.linbox
define k = GF(9, 'alpha'); type(k)
define k.base_ring()
Finite Field of size 3
```
sage: k = GF((3, 40), 'b'); type(k)
<class 'sage.rings.finite_rings.finite_field_pari_ffelt.FiniteField_pari_ffelt_with_category'>
sage: k.base_ring()
Finite Field of size 3

Further examples:

sage: GF(2).is_field()
True
sage: GF(next_prime(10^20)).is_field()
True
sage: GF(19^20, 'a').is_field()
True
sage: GF(8, 'a').is_field()
True

AUTHORS:

- William Stein: initial version
- Robert Bradshaw: prime field implementation
- Martin Albrecht: Givaro and ntl.GF2E implementations

class sage.rings.finite_rings.finite_field_constructor.FiniteFieldFactory(*args, **kwds)

Bases: UniqueFactory

Return the globally unique finite field of given order with generator labeled by the given name and possibly with given modulus.

INPUT:

- order – a prime power
- name – string, optional. Note that there can be a substantial speed penalty (in creating extension fields) when omitting the variable name, since doing so triggers the computation of pseudo-Conway polynomials in order to define a coherent lattice of extensions of the prime field. The speed penalty grows with the size of extension degree and with the number of factors of the extension degree.
- modulus – (optional) either a defining polynomial for the field, or a string specifying an algorithm to use to generate such a polynomial. If modulus is a string, it is passed to irreducible_element() as the parameter algorithm; see there for the permissible values of this parameter. In particular, you can specify modulus="primitive" to get a primitive polynomial. You may not specify a modulus if you do not specify a variable name.
- impl – (optional) a string specifying the implementation of the finite field. Possible values are:
  - 'modn' – ring of integers modulo p (only for prime fields).
  - 'givaro' – Givaro, which uses Zech logs (only for fields of at most 65521 elements).
  - 'ntl' – NTL using GF2X (only in characteristic 2).
  - 'pari' or 'pari_ffelt' – PARI's FFELT type (only for extension fields).
- elem_cache – (default: order < 500) cache all elements to avoid creation time; ignored unless impl='givaro'
- repr – (default: 'poly') ignored unless impl='givaro'; controls the way elements are printed to the user:
Finite Rings, Release 10.3

- 'log': repr is log_repr()
- 'int': repr is int_repr()
- 'poly': repr is poly_repr()

- check_irreducible – verify that the polynomial modulus is irreducible
- proof – bool (default: True): if True, use provable primality test; otherwise only use pseudoprimality test.

ALIAS: You can also use GF instead of FiniteField – they are identical.

EXAMPLES:

```
sage: k.<a> = FiniteField(9); k
Finite Field in a of size 3^2
sage: parent(a)
Finite Field in a of size 3^2
sage: charpoly(a, 'y')
y^2 + 2*y + 2
```

We illustrate the proof flag. The following example would hang for a very long time if we didn’t use proof=False.

**Note:** Magma only supports proof=False for making finite fields, so falsely appears to be faster than Sage – see github issue #10975.

```
sage: k = FiniteField(10^1000 + 453, proof=False)
sage: k = FiniteField((10^1000 + 453)^2, a, proof=False)  # long time --...
˓→about 5 seconds
```

```
sage: F.<x> = GF(5)[]
sage: K.<a> = GF(5**5, name='a', modulus=x^5 - x +1 )
sage: f = K.modulus(); f
x^5 + 4*x + 1
sage: type(f)
<class 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>
```

By default, the given generator is not guaranteed to be primitive (a generator of the multiplicative group), use modulus="primitive" if you need this:

```
sage: K.<a> = GF(5^45)
sage: a.multiplicative_order()
7105427357601001858711242675781
sage: a.is_square()
True
sage: K.<b> = GF(5^45, modulus="primitive")
sage: b.multiplicative_order()
28421709430404007434844970703124
```

The modulus must be irreducible:

```
sage: K.<a> = GF(5**5, name='a', modulus=x^5 - x)
Traceback (most recent call last):
...
ValueError: finite field modulus must be irreducible but it is not
```
You can't accidentally fool the constructor into thinking the modulus is irreducible when it is not, since it actually tests irreducibility modulo \( p \). Also, the modulus has to be of the right degree (this is always checked):

```
sage: F.<x> = QQ[]
sage: factor(x^5 + 2)
x^5 + 2
sage: K.<a> = GF(5^5, modulus=x^5 + 2)
Traceback (most recent call last):
  ... ValueError: finite field modulus must be irreducible but it is not
sage: K.<a> = GF(5^5, modulus=x^3 + 3*x + 3, check_irreducible=False)
Traceback (most recent call last):
  ... ValueError: the degree of the modulus does not equal the degree of the field
```

Any type which can be converted to the polynomial ring \( GF(p)[x] \) is accepted as modulus:

```
sage: K.<a> = GF(13^3, modulus=[1,0,0,2])
sage: K.<a> = GF(13^10, modulus=pari("ffinit(13,10)"))
sage: var('x')
x
sage: K.<a> = GF(13^2, modulus=x^2 - 2)
sage: K.<a> = GF(13^2, modulus=sin(x))
Traceback (most recent call last):
  ... TypeError: self must be a numeric expression
```

If you wish to live dangerously, you can tell the constructor not to test irreducibility using `check_irreducible=False`, but this can easily lead to crashes and hangs – so do not do it unless you know that the modulus really is irreducible!

```
sage: K.<a> = GF(5**2, name=a, modulus=x^2 + 2, check_irreducible=False)
```

Even for prime fields, you can specify a modulus. This will not change how Sage computes in this field, but it will change the result of the `modulus()` and `gen()` methods:

```
sage: k.<a> = GF(5, modulus="primitive")
sage: k.modulus()
x + 3
sage: a
2
```

The order of a finite field must be a prime power:

```
sage: GF(1)
Traceback (most recent call last):
  ... ValueError: the order of a finite field must be at least 2
sage: GF(100)
Traceback (most recent call last):
  ... ValueError: the order of a finite field must be a prime power
```

Finite fields with explicit random modulus are not cached:

```
sage: k.<a> = GF(5**10, modulus=random)
sage: n.<a> = GF(5**10, modulus='random')
```

(continues on next page)
sage: while k.modulus() == n.modulus():
....: n.<a> = GF(5**10, modulus='random')
sage: n is k
False
sage: GF(5**10, 'a') is GF(5**10, 'a')
True

We check that various ways of creating the same finite field yield the same object, which is cached:

sage: K = GF(7, 'a')
sage: L = GF(7, 'b')
sage: K is L  # name is ignored for prime fields
True
sage: K is GF(7, modulus=K.modulus())
True
sage: K = GF(4,'a'); K.modulus()
x^2 + x + 1
sage: L = GF(4,'a', K.modulus())
sage: K is L
True
sage: M = GF(4,'a', K.modulus().change_variable_name('y'))
sage: K is M
True

You may print finite field elements as integers. This currently only works if the order of field is \(< 2^{16}\), though:

sage: k.<a> = GF(2^8, repr='int')
sage: a
2

The following demonstrate coercions for finite fields using Conway polynomials:

sage: k = GF(5^2); a = k.gen()
sage: l = GF(5^5); b = l.gen()
sage: a + b
3*z10^5 + z10^4 + z10^2 + 3*z10 + 1

Note that embeddings are compatible in lattices of such finite fields:

sage: m = GF(5^3); c = m.gen()
sage: (a+b)+c == a+(b+c)
True
sage: (a*b)*c == a*(b*c)
True
sage: from sage.categories.pushout import pushout
sage: n = pushout(k, l)
sage: o = pushout(l, m)
sage: q = pushout(n, o)
sage: q(o(b)) == q(n(b))
True

Another check that embeddings are defined properly:

sage: k = GF(3**10)
sage: l = GF(3**20)
sage: l(k.gen())**10 == l(k.gen())**10
True
Using pseudo-Conway polynomials is slow for highly composite extension degrees:

```
sage: k = GF(3^120)  # long time (about 3 seconds)
sage: GF(3^40).gen().minimal_polynomial()(k.gen()^((3^120-1)/(3^40-1)))  # long...
  → time (because of previous line)
0
```

Before github issue #17569, the boolean keyword argument `conway` was required when creating finite fields without a variable name. This keyword argument is now removed (github issue #21433). You can still pass in `prefix` as an argument, which has the effect of changing the variable name of the algebraic closure:

```
sage: K = GF(3^10, prefix='w'); L = GF(3^10); K is L
False
sage: K.variable_name(), L.variable_name()
('w10', 'z10')
sage: list(K.polynomial()) == list(L.polynomial())
True
```

### create_key_and_extra_args

```
create_key_and_extra_args(order, name=None, modulus=None, names=None, impl=None,
                        proof=None, check_prime=True, check_irreducible=True, prefix=None,
                        repr=None, elem_cache=None, **kwds)
```

**EXAMPLES:**

```
sage: GF.create_key_and_extra_args(9, 'a')  # needs sage.libs.linbox
((9, ('a',)), x^2 + 2*x + 2, 'givaro', 3, 2, True, None, 'poly', True, True,...
  →True), (})
```

The order \( q \) can also be given as a pair \((p, n)\):

```
sage: GF.create_key_and_extra_args((3, 2), 'a')  # needs sage.libs.linbox
((9, ('a',)), x^2 + 2*x + 2, 'givaro', 3, 2, True, None, 'poly', True, True,...
  →True), (})
```

We do not take invalid keyword arguments and raise a value error to better ensure uniqueness:

```
sage: GF.create_key_and_extra_args(9, 'a', foo='value')
Traceback (most recent call last):
 ...
TypeError: ...create_key_and_extra_args() got an unexpected keyword argument...
  →'foo'
```

Moreover, `repr` and `elem_cache` are ignored when not using givaro:

```
sage: GF.create_key_and_extra_args(16, 'a', impl='ntl', repr='poly')  # needs sage.libs.nltl
((16, ('a',)), x^4 + x + 1, 'ntl', 2, 4, True, None, None, None, True, True), (})
sage: GF.create_key_and_extra_args(16, 'a', impl='ntl', elem_cache=False)  # needs sage.libs.nltl
((16, ('a',)), x^4 + x + 1, 'ntl', 2, 4, True, None, None, None, True, True), (})
sage: GF(16, impl='ntl') is GF(16, impl='ntl', repr='foo')  # needs sage.libs.nltl
True
```

We handle extra arguments for the givaro finite field and create unique objects for their defaults:
We explicitly take `structure`, `implementation` and `prec` attributes for compatibility with `AlgebraicExtensionFunctor` but we ignore them as they are not used, see github issue #21433:

```
sage: GF.create_key_and_extra_args(9, a, structure=None)  # needs sage.libs.linbox
((9, ('a',)), x^2 + 2*x + 2, 'givaro', 3, 2, True, None, 'poly', True, True, 
   True), {})
```

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

**EXAMPLES:**

```
sage: K = GF(19)  # indirect doctest
sage: TestSuite(K).run()
```

We try to create finite fields with various implementations:

```
sage: k = GF(2, impl='modn')
sage: k = GF(2, impl='givaro')  # needs sage.libs.linbox
sage: k = GF(2, impl='ntl')  # needs sage.libs.ntl
sage: k = GF(2, impl='pari')
Traceback (most recent call last):
...
ValueError: the degree must be at least 2
sage: k = GF(2, impl='supercalifragilisticexpialidocious')
Traceback (most recent call last):
...
ValueError: no such finite field implementation:
   'supercalifragilisticexpialidocious'
sage: k.<a> = GF(2^15, impl='modn')
Traceback (most recent call last):
...
ValueError: the 'modn' implementation requires a prime order
sage: k.<a> = GF(2^15, impl='givaro')  # needs sage.libs.linbox
sage: k.<a> = GF(2^15, impl='ntl')  # needs sage.libs.ntl
sage: k.<a> = GF(2^15, impl='pari')
sage: k.<a> = GF(3^60, impl='modn')
Traceback (most recent call last):
...
ValueError: the 'modn' implementation requires a prime order
sage: k.<a> = GF(3^60, impl='givaro')  # needs sage.libs.linbox
Traceback (most recent call last):
...
```

(continues on next page)
sage: k.<a> = GF(3^60, impl='ntl')  # needs sage.libsntl
Traceback (most recent call last):
  ...  
ValueError: q must be a 2-power
sage: k.<a> = GF(3^60, impl='pari')

sage.rings.finite_rings.finite_field_constructor.is_PrimeFiniteField(x)

Return True if x is a prime finite field.

This function is deprecated.

EXAMPLES:

sage: from sage.rings.finite_rings.finite_field_constructor import is_
  PrimeFiniteField
sage: is_PrimeFiniteField(QQ)
False
sage: is_PrimeFiniteField(GF(7))
True
sage: is_PrimeFiniteField(GF(7^2, 'a'))
False
sage: is_PrimeFiniteField(GF(next_prime(10^90, proof=False)))
True

2.2 Base class for finite fields

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: 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 \texttt{self}. Note that mathematically speaking, this is only unique up to \textit{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 \textit{Conway polynomials}; see for example Wikipedia article \texttt{Conway_polynomial_(finite_fields)}. These have the drawback that computing them takes a long time. Therefore Sage implements a variant called \textit{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.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: F = GF(5).algebraic_closure()
sage: F
Algebraic closure of Finite Field of size 5
sage: F.gen(3)
z3
\end{verbatim}

The default name is 'z' but you can change it through the option \texttt{name}:

\begin{verbatim}
sage: Ft = GF(5).algebraic_closure('t')
sage: Ft.gen(3)
t3
\end{verbatim}

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:

\begin{verbatim}
sage: F = GF(7).algebraic_closure()
sage: x = F.gen(2)
sage: loads(dumps(x)) == x
False
\end{verbatim}

\textbf{Note:} This is currently only implemented for prime fields.

\textbf{cardinality()}\texttt{ }

Return the cardinality of \texttt{self}.

Same as \texttt{order()}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: GF(997).cardinality()
997
\end{verbatim}

\textbf{construction()}\texttt{ }

Return the construction of this finite field, as a \texttt{ConstructionFunctor} and the base field.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: v = GF(3^3).construction(); v
(AlgebraicExtensionFunctor, Finite Field of size 3)
sage: v[0].polys[0]
3
\end{verbatim}
The implementation is taken into account, by github issue #15223:

```python
sage: k = FiniteField(9, 'a', impl='pari_ffelt')
sage: F, R = k.construction()
sage: F(R) is k
True
```

**dual_basis** *(basis=None, check=True)*

Return the dual basis of `basis`, or the dual basis of the power basis if no basis is supplied.

If \( e = \{e_0, e_1, \ldots, e_{n-1}\} \) is a basis of \( F_p^n \) as a vector space over \( F_p \), then the dual basis of \( e, d = \{d_0, d_1, \ldots, d_{n-1}\} \), is the unique basis such that \( \text{Tr}(e_i d_j) = \delta_{i,j}, 0 \leq i, j \leq n - 1 \), where \( \text{Tr} \) is the trace function.

**INPUT:**

- `basis` – (default: `None`): a basis of the finite field `self`, \( F_p^n \), as a vector space over the base field \( F_p \).
  - Uses the power basis \( \{x^i : 0 \leq i \leq n - 1\} \) as input if no basis is supplied, where \( x \) is the generator of `self`.
- `check` – (default: `True`): verifies that `basis` is a valid basis of `self`.

**ALGORITHM:**

The algorithm used to calculate the dual basis comes from pages 110–111 of [McE1987].

Let \( e = \{e_0, e_1, \ldots, e_{n-1}\} \) be a basis of \( F_p^n \) as a vector space over \( F_p \), and \( d = \{d_0, d_1, \ldots, d_{n-1}\} \) be the dual basis of \( e \). Since \( e \) is a basis, we can rewrite any \( d_c, 0 \leq c \leq n - 1 \), as \( d_c = \beta_0 e_0 + \beta_1 e_1 + \cdots + \beta_{n-1} e_{n-1} \), for some \( \beta_0, \beta_1, \ldots, \beta_{n-1} \in F_p \). Using properties of the trace function, we can rewrite the \( n \) equations of the form \( \text{Tr}(e_i d_c) = \delta_{i,c} \) and express the result as the matrix vector product: \( A[\beta_0, \beta_1, \ldots, \beta_{n-1}] = i_c \), where the \( i, j \)-th element of \( A \) is \( \text{Tr}(e_i e_j) \) and \( i_c \) is the \( i \)-th column of the \( n \times n \) identity matrix. Since \( A \) is an invertible matrix, \( [\beta_0, \beta_1, \ldots, \beta_{n-1}] = A^{-1} i_c \), from which we can easily calculate \( d_c \).

**EXAMPLES:**

```python
sage: F.<a> = GF(2^4)
sage: F.dual_basis(basis=None, check=False)  # needs sage.modules
[a^3 + 1, a^2, a, 1]
```

We can test that the dual basis returned satisfies the defining property of a dual basis: \( \text{Tr}(e_i d_j) = \delta_{i,j}, 0 \leq i, j \leq n - 1 \)

```python
sage: # needs sage.modules
sage: F.<a> = GF(7^4)
sage: e = [4*a^3, 2*a^3 + a^2 + 3*a + 5, ...: 3*a^3 + 5*a^2 + 4*a + 2, 2*a^3 + 2*a^2 + 2]
sage: d = F.dual_basis(e, check=True); d
[3*a^3 + 4*a^2 + 2*a + 5, a^3 + 6*a + 5, 3*a^3 + 6*a^2 + 2*a + 5, 5*a^2 + 4*a + 3]

sage: vals = [[(x * y).trace() for x in e] for y in d]
sage: matrix(vals) == matrix.identity(4)
True
```

We can test that if \( d \) is the dual basis of \( e \), then \( e \) is the dual basis of \( d \):
sage: # needs sage.modules
sage: F.<a> = GF(7^8)
sage: e = [a^0, a^1, a^2, a^3, a^4, a^5, a^6, a^7]
sage: d = F.dual_basis(e, check=False); d
[6*a^6 + 4*a^5 + 4*a^4 + a^3 + 6*a^2 + 3,
  6*a^7 + 4*a^6 + 4*a^5 + 2*a^4 + a^2,
  4*a^6 + 5*a^5 + 5*a^4 + 4*a^3 + 5*a^2 + a + 6,
  5*a^7 + a^6 + a^4 + 4*a^3 + 4*a^2 + 1,
  2*a^7 + 5*a^6 + a^5 + a^3 + 2*a^2 + 4,
  a^7 + 2*a^6 + 5*a^5 + a^4 + 5*a^2 + 4*a + 4,
  a^7 + a^6 + 2*a^5 + 5*a^4 + a^3 + 4*a^2 + 4*a + 6,
  5*a^7 + a^6 + a^5 + 2*a^4 + 5*a^3 + 6*a]
sage: F.dual_basis(d)
[1, a, a^2, a^3, a^4, a^5, a^6, a^7]

We cannot calculate the dual basis if basis is not a valid basis.

sage: F.<a> = GF(2^3)
sage: F.dual_basis([a], check=True)              # needs sage.modules
Traceback (most recent call last):
  ...
ValueError: basis length should be 3, not 1
sage: F.dual_basis([a^0, a, a^0 + a], check=True)  # needs sage.modules
Traceback (most recent call last):
  ...
ValueError: value of 'basis' keyword is not a basis

AUTHOR:

• Thomas Gagne (2015-06-16)

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

Return an extension of this finite field.

INPUT:

• modulus – a polynomial with coefficients in self, or an integer.
• name or names – string: the name of the generator in the new extension
• latex_name or latex_names – string: latex 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:
An example using the map argument:

```
FINITE: k = GF(2)
FINITE: R.<x> = k[
FINITE: k.extension(x^1000 + x^5 + x^4 + x^3 + 1, 'a')
FINITE Field in a of size 2^1000
FINITE: k = GF(3^4)
FINITE: R.<x> = k[
FINITE: k.extension(3)
FINITE Field in z12 of size 3^12
FINITE: K = k.extension(2, 'a')
FINITE: k.is_subring(K)
FINITE True
```

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

```
FINITE: 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^5 + x^2 + x + 2
```

**factored_order()**

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

EXAMPLES:

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

**factored_unit_order()**

Returns the factorization of `self.order()-1`, as a 1-tuple.

The format is for compatibility with `integer_mod_ring`.

EXAMPLES:

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

**fetch_int(*args, **kwds)**

Deprecated: Use `from_integer()` instead. See github issue #33941 for details.

**free_module(base=None, basis=None, map=True)**

Return the vector space over the subfield isomorphic to this finite field as a vector space, along with the isomorphisms.
INPUT:

- **base** — a subfield of or a morphism into this finite field. If not given, the prime subfield is assumed. A subfield means a finite field with coercion to this finite field.
- **basis** — a basis of the finite field as a vector space over the subfield. If not given, one is chosen automatically.
- **map** — boolean (default: True); if True, isomorphisms from and to the vector space are also returned.

The basis maps to the standard basis of the vector space by the isomorphisms.

OUTPUT: if map is False,

- vector space over the subfield or the domain of the morphism, isomorphic to this finite field.

and if map is True, then also

- an isomorphism from the vector space to the finite field.

- the inverse isomorphism to the vector space from the finite field.

EXAMPLES:

```python
sage: GF(27, 'a').vector_space(map=False)  #˓→ needs sage.modules
Vector space of dimension 3 over Finite Field of size 3

sage: # needs sage.modules
sage: F = GF(8)
sage: E = GF(64)
sage: V, from_V, to_V = E.vector_space(F, map=True)
sage: V
Vector space of dimension 2 over Finite Field in z3 of size 2^3
sage: to_V(E.gen())
(0, 1)
sage: all(from_V(to_V(e)) == e for e in E)
True
sage: all(to_V(e1 + e2) == to_V(e1) + to_V(e2) for e1 in E for e2 in E)
True
sage: all(to_V(c * e) == c * to_V(e) for e in E for c in F)
True

sage: # needs sage.modules
sage: basis = [E.gen(), E.gen() + 1]
sage: W, from_W, to_W = E.vector_space(F, basis, map=True)
sage: all(from_W(to_W(e)) == e for e in E)
True
sage: all(to_W(c * e) == c * to_W(e) for e in E for c in F)
True
sage: all(to_W(e1 + e2) == to_W(e1) + to_W(e2) for e1 in E for e2 in E)  #˓→ long time
True
sage: to_W(basis[0]); to_W(basis[1])
(1, 0)
(0, 1)
```

```python
sage: # needs sage.modules
sage: x = polygen(ZZ)
sage: F = GF(9, t, modulus=x^2 + x - 1)
sage: E = GF(81)
```

(continues on next page)
sage: h = Hom(F,E).an_element()

sage: V, from_V, to_V = E.vector_space(h, map=True)

sage: V
Vector space of dimension 2 over Finite Field in t of size 3^2

sage: V.base_ring() is F
True

sage: all(from_V(to_V(e)) == e for e in E)
True

sage: all(to_V(e1 + e2) == to_V(e1) + to_V(e2) for e1 in E for e2 in E)
True

sage: all(to_V(h(c) * e) == c * to_V(e) for e in E for c in F)
True

\textbf{frobenius\_endomorphism}(n=1)

INPUT:

\begin{itemize}
\item n – an integer (default: 1)
\end{itemize}

OUTPUT:

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

EXAMPLES:

\begin{Verbatim}
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()
# needs sage.modules
sage: Frob(a) == a^3
# needs sage.modules
True
\end{Verbatim}

We can specify a power:

\begin{Verbatim}
sage: k.frobenius_endomorphism(2)
Frobenius endomorphism t |--> t^{3^2} on Finite Field in t of size 3^5
\end{Verbatim}

The result is simplified if possible:

\begin{Verbatim}
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
\end{Verbatim}

Comparisons work:

\begin{Verbatim}
sage: k.frobenius_endomorphism(6) == Frob
True

sage: from sage.categories.morphism import IdentityMorphism
sage: k.frobenius_endomorphism(5) == IdentityMorphism(k)
True
\end{Verbatim}

AUTHOR:

\begin{itemize}
\item Xavier Caruso (2012-06-29)
\end{itemize}
**from_bytes** (*input_bytes, byteorder='big')

Return the integer represented by the given array of bytes.

Internally relies on the python `int.from_bytes()` method.

**INPUT:**

- `input_bytes` – a bytes-like object or iterable producing bytes
- `byteorder` – `str` (default: "big"); determines the byte order of `input_bytes`; can only be "big" or "little"

**EXAMPLES:**

```python
sage: input_bytes = b"some_bytes"
sage: F = GF(2**127 - 1)
sage: F.from_bytes(input_bytes)
545127616933790290830707
sage: a = F.from_bytes(input_bytes, byteorder="little"); a
544943659528996309004147
sage: type(a)
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_gmp'>
```

**from_integer** (*n*, `reverse=False`)

Return the finite field element obtained by reinterpreting the base-\(p\) expansion of \(n\) as a polynomial and evaluating it at the generator of this finite field.

If `reverse` is set to `True` (default: `False`), the list of digits is reversed prior to evaluation.

**Inverse of** `sage.rings.finite_rings.element_base.FinitePolyExtElement.to_integer()`.

**INPUT:**

- `n` – integer between 0 and the cardinality of this field minus 1.

**EXAMPLES:**

```python
sage: p = 4091
sage: F = GF(p^4, 'a')
sage: n = 100*p^3 + 37*p^2 + 12*p + 6
sage: F.from_integer(n)
100*a^3 + 37*a^2 + 12*a + 6
sage: F.from_integer(n) in F
True
sage: F.from_integer(n, reverse=True)
6*a^3 + 12*a^2 + 37*a + 100
```

**galois_group()**

Return the Galois group of this finite field, a cyclic group generated by Frobenius.

**EXAMPLES:**

---

**54 Chapter 2. Finite Fields**
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
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
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
sage: k.<a> = FiniteField(3^4)
sage: k.is_field()  
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
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, \( \mathbb{F}_{p^n} \), the base field is \( \mathbb{F}_p \).

**OUTPUT:**

- a monic polynomial over \( \mathbb{F}_p \) in the variable \( x \).

**EXAMPLES:**

```plaintext
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.)

```plaintext
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:

```plaintext
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:

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

```python
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
```

**ngens()**

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

**EXAMPLES:**

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

**order()**

Return the order of this finite field.

**EXAMPLES:**

```python
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 \( \mathbb{F}_p \) 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: # needs sage.libsntl
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(9^7)
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
random_element(*args, **kwds)
A random element of the finite field. Passes arguments to random_element() function of underlying vector space.

EXAMPLES:

```python
sage: k = GF(19^4, 'a')
sage: k.random_element().parent() is k
True
```

Passes extra positional or keyword arguments through:

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

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

EXAMPLES:

```python
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]
```

subfield(degree, name=None, map=False)
Return the subfield of the field of degree.

The inclusion maps between these subfields will always commute, but they are only added as coercion maps if the following condition holds for the generator \( g \) of the field, where \( d \) is the degree of this field over the prime field:

The element \( g(p^d-1)/(p^n-1) \) generates the subfield of degree \( n \) for all divisors \( n \) of \( d \).

INPUT:

- degree – integer; degree of the subfield
- name – string; name of the generator of the subfield
- map – boolean (default False); whether to also return the inclusion map

EXAMPLES:

```python
sage: k = GF(2^21)
sage: k.subfield(3)
Finite Field in z3 of size 2^3
sage: k.subfield(7, 'a')
Finite Field in a of size 2^7
sage: k.coerce_map_from(_)
Ring morphism:
  From: Finite Field in a of size 2^7
  To:   Finite Field in z21 of size 2^21
  Defn: a |--> z21^20 + z21^19 + z21^17 + z21^15 + z21^14 + z21^6 + z21^4 +...
  ~z21^3 + z21
sage: k.subfield(8)
Traceback (most recent call last):
```
...\nValueError: no subfield of order 2^8

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

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')
k.order()
1024
k.unit_group_exponent()
1023
```

**zeta** *(n=None)*

Return an element of multiplicative order *n* in this finite field. If there is no such element, raise `ValueError`.

**Warning:** In general, this returns an arbitrary element of the correct order. There are no compatibility guarantees: `F.zeta(9) ^ 3` may not be equal to `F.zeta(3)`.

**EXAMPLES:**

```sage
k = GF(7)
k.zeta()
3
k.zeta().multiplicative_order()
6
k.zeta(3)
2
k.zeta(3).multiplicative_order()
3
k = GF(49, 'a')
k.zeta().multiplicative_order()
48
k.zeta(6)
3
k.zeta(5)
Traceback (most recent call last):
...
ValueError: no 5th root of unity in Finite Field in a of size 7^2
```

Even more examples:

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

This works even in very large finite fields, provided that *n* can be factored (see github issue #25203):

```sage
k.<a> = GF(2^2000)
p =...
z = k.zeta(p)
z
a^1999 + a^1996 + a^1995 + a^1994 + ... + a^7 + a^5 + a^4 + 1
z ^ p
1
```

2.2. Base class for finite fields

61
**zeta_order()**

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

**EXAMPLES:**

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

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

Return whether the implementation of `R` has the interface provided by the standard finite field implementation.

This function is deprecated.

**EXAMPLES:**

```python
sage: from sage.rings.finite_rings.finite_field_base import is_FiniteField
dsage: is_FiniteField(GF(9, 'a'))
doctest:...: DeprecationWarning: the function is_FiniteField is deprecated; use...
˓→isinstance(x, sage.rings.finite_rings.finite_field_base.FiniteField) instead
See https://github.com/sagemath/sage/issues/32664 for details.
True
dsage: is_FiniteField(GF(next_prime(10^10)))
True
```

Note that the integers modulo `n` are not backed by the finite field type:

```python
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.

**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.

## 2.3 Base class for finite field elements

**AUTHORS:**

- David Roe (2010-01-14): factored out of sage.structure.element
- Sebastian Oehms (2018-07-19): added `conjugate()` (see github issue #26761)

**class** `sage.rings.finite_rings.element_base.Cache_base`

**Bases:** `SageObject`
**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.

**EXAMPLES:**

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

**class** *sage.rings.finite_rings.element_base.FinitePolyExtElement*

Bases: *FiniteRingElement*

Elements represented as polynomials modulo a given ideal.

**additive_order()**

Return the additive order of this finite field element.

**EXAMPLES:**

```python
sage: k.<a> = FiniteField(2^12, 'a')
sage: b = a^3 + a + 1
sage: b.additive_order()
2
sage: k(0).additive_order()
1
```

**charpoly** *(var='x', algorithm='pari')*

Return the characteristic polynomial of *self* as a polynomial with given variable.

**INPUT:**

- **var** – string (default: 'x')
- **algorithm** – string (default: 'pari')
  - 'pari' – use pari’s charpoly
  - 'matrix' – return the charpoly computed from the matrix of left multiplication by *self*

The result is not cached.

**EXAMPLES:**

```python
sage: from sage.rings.finite_rings.element_base import FinitePolyExtElement
sage: k.<a> = FiniteField(2^12, 'a')
sage: b = a^3 + a + 1
sage: b.charpoly('x', algorithm='pari')  # needs sage.modules
x^3 + x + 1
```

(continues on next page)
conjugate()

This method returns the result of the Frobenius morphism in the case where the field is a quadratic extension, say GF(q^2), where q = p^k is a prime power and p the characteristic of the field.

OUTPUT:

Instance of this class representing the image under the Frobenius morphisms.

EXAMPLES:

```sage
sage: F.<a> = GF(16)
sage: b = a.conjugate(); b
a + 1
sage: a == b.conjugate()
True

sage: F.<a> = GF(27)
sage: a.conjugate()
Traceback (most recent call last):
...
TypeError: cardinality of the field must be a square number
```

frobenius(k=1)

Return the (p^k)^th power of self, where p is the characteristic of the field.

INPUT:

- `k` – integer (default: 1, must fit in C int type)

Note that if k is negative, then this computes the appropriate root.

EXAMPLES:

```sage
sage: F.<a> = GF(29^2)
sage: z = a^2 + 5*a + 1
sage: z.pth_power()
19*a + 20
sage: z.pth_power(10)
19*a + 20
sage: z.pth_power(-10) == z
True

sage: F.<b> = GF(2^12)
sage: y = b^3 + b + 1
sage: y == (y.pth_power(-3))^(2^3)
True
```

integer_representation(*args, **kwds)

Deprecated: Use to_integer() instead. See github issue #33941 for details.

is_square()

Returns True if and only if this element is a perfect square.

EXAMPLES:
Finite Rings, Release 10.3

sage: k.<a> = FiniteField(9, impl='givaro', modulus='primitive')  # needs sage.libs.linbox
sage: a.is_square()  # needs sage.libs.linbox
False
sage: (a**2).is_square()  # needs sage.libs.linbox
True
sage: k.<a> = FiniteField(4, impl='ntl', modulus='primitive')  # needs sage.libs.ntl
sage: (a**2).is_square()  # needs sage.libs.ntl
True
sage: k.<a> = FiniteField(17^5, impl='pari_ffelt', modulus='primitive')  # needs sage.libs.pari
sage: a.is_square()  # needs sage.libs.pari
False
sage: (a**2).is_square()  # needs sage.libs.pari
True
sage: k(0).is_square()  # needs sage.libs.linbox
True

list()

Return the list of coefficients (in little-endian) of this finite field element when written as a polynomial in the generator.

Equivalent to calling list() on this element.

EXAMPLES:

sage: x = polygen(GF(71))
sage: F.<u> = GF(71^7, modulus=x^7 + x + 1)
sage: a = 3 + u + 3*u^2 + 3*u^3 + 7*u^4
sage: a.list()
[3, 1, 3, 3, 7, 0, 0]
sage: a.list() == list(a) == [a[i] for i in range(F.degree())]
True

The coefficients returned are those of a fully reduced representative of the finite field element:

sage: b = u^777
sage: b.list()
[9, 69, 4, 27, 40, 10, 56]
sage: (u.polynomial()^777).list()
[0, 0, 0, 0, ..., 0, 1]

matrix(reverse=False)

Return the matrix of left multiplication by the element on the power basis $1, x, x^2, \ldots, x^{d-1}$ for the field extension.

Thus the columns of this matrix give the images of each of the $x^i$.

INPUT:

* reverse – if True, act on vectors in reversed order

2.3. Base class for finite field elements
EXAMPLES:

```python
sage: # needs sage.modules
sage: k.<a> = GF(2^4)
sage: b = k.random_element()
```
```python
sage: vector(a*b) == a.matrix() * vector(b)
True
sage: (a*b)._vector_(reverse=True) == a.matrix(reverse=True) * b._vector_(reverse=True)
True
```

**minimal_polynomial**(var='x')

Returns the minimal polynomial of this element (over the corresponding prime subfield).

EXAMPLES:

```python
sage: k.<a> = FiniteField(3^4)
```
```python
sage: parent(a)
Finite Field in a of size 3^4
sage: b=a**20;p=charpoly(b,"y");p
```
```python
y^4 + 2*y^2 + 1
```
```python
sage: factor(p)
```
```python
(y^2 + 1)^2
```
```python
sage: b.minimal_polynomial('y')
```
```python
y^2 + 1
```

**minpoly**(var='x', algorithm='pari')

Returns the minimal polynomial of this element (over the corresponding prime subfield).

INPUT:

- `var` - string (default: 'x')
- `algorithm` - string (default: 'pari')
  - 'pari' – use pari’s minpoly
  - 'matrix' – return the minpoly computed from the matrix of left multiplication by self

EXAMPLES:

```python
sage: from sage.rings.finite_rings.element_base import FinitePolyExtElement
sage: k.<a> = FiniteField(19^2)
```
```python
sage: b=a**20
```
```python
sage: p=FinitePolyExtElement.minpoly(b,"x", algorithm="pari")
```
```python
sage: q=FinitePolyExtElement.minpoly(b,"x", algorithm="matrix")
```
```python
sage: p == q
True
```
```python
sage: p
```
```python
x + 17
```

**multiplicative_order**()

Return the multiplicative order of this field element.

EXAMPLES:

```python
sage: S.<a> = GF(5^3); S
```
```python
Finite Field in a of size 5^3
```

(continues on next page)
sage: a.multiplicative_order()
124
sage: (a^8).multiplicative_order()
31
sage: S(0).multiplicative_order()
Traceback (most recent call last):
  ...
ArithmeticError: Multiplicative order of 0 not defined.

norm()

Return the norm of self down to the prime subfield.

This is the product of the Galois conjugates of self.

EXAMPLES:

sage: S.<b> = GF(5^2); S
Finite Field in b of size 5^2
sage: b.norm()
2
sage: b.charpoly('t')
t^2 + 4*t + 2

Next we consider a cubic extension:

sage: S.<a> = GF(5^3); S
Finite Field in a of size 5^3
sage: a.norm()
2
sage: a.charpoly('t')
t^3 + 3*t + 3
sage: a * a^5 * (a^25)
2

nth_root (n, extend=False, all=False, algorithm=None, cunningham=False)

Returns an n-th root of self.

INPUT:

• n – integer ≥ 1

• extend – bool (default: False): if True, return an n-th 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 n-th roots of self, instead of just one.

• algorithm – string (default: None): ‘Johnston’ is the only currently supported option. For IntegerMod elements, the problem is reduced to the prime modulus case using CRT and \( p \)-adic logs, and then this algorithm used.

OUTPUT:

If self has an n-th root, returns one (if all is False) or a list of all of them (if all is True). Otherwise, raises a ValueError (if extend is False) or a Not Implemented Error (if extend is True).

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

EXAMPLES:
\begin{verbatim}
sage: K = GF(31)
sage: a = K(22)
sage: K(22).nth_root(7)
sage: K(25).nth_root(5)
sage: K(23).nth_root(3)
sage: K.<a> = GF(625)
sage: (3*a^2+a+1).nth_root(13)**13
13*a^2 + a + 1
sage: k.<a> = GF(29^2)
sage: b = a^2 + 5*a + 1
sage: b.nth_root(11)
3*a + 20
sage: b.nth_root(5)
Traceback (most recent call last):
...  
ValueError: no nth root
sage: b.nth_root(5, all = True)
[]
sage: b.nth_root(3, all = True)
[14*a + 18, 10*a + 13, 5*a + 27]
sage: k.<a> = GF(29^5)
sage: b = a^2 + 5*a + 1
sage: b.nth_root(5)
19*a^4 + 2*a^3 + 2*a^2 + 16*a + 3
sage: b.nth_root(7)
Traceback (most recent call last):
...  
ValueError: no nth root
sage: b.nth_root(4, all=True)
[]
\end{verbatim}

ALGORITHM:

The default is currently an algorithm described in [Joh1999].

AUTHOR:

- David Roe (2010-02-13)

\texttt{pth\_power (k=1)}

Return the \((p^k)^{th}\) power of self, where \(p\) is the characteristic of the field.

INPUT:

- \(k\) – integer (default: \(1\), must fit in \texttt{C} int type)

Note that if \(k\) is negative, then this computes the appropriate root.

EXAMPLES:

\begin{verbatim}
sage: F.<a> = GF(29^2)
sage: z = a^2 + 5*a + 1
sage: z.pth\_power()
19*a + 20
\end{verbatim}
```
sage: z.pth_power(10)
10*a + 28
sage: z.pth_power(-10) == z
True
sage: F.<b> = GF(2^12)
sage: y = b^3 + b + 1
sage: y == (y.pth_power(-3))^(2^3)
True
sage: y.pth_power(2)
b^7 + b^6 + b^5 + b^4 + b^3 + b
```

**pth_root** \((k=1)\)

Return the \((p^k)^{th}\) root of self, where \(p\) is the characteristic of the field.

**INPUT:**

- \(k\) – integer (default: 1, must fit in C int type)

Note that if \(k\) is negative, then this computes the appropriate power.

**EXAMPLES:**

```
sage: F.<b> = GF(2^12)
sage: y = b^3 + b + 1
sage: y == (y.pth_root(3))^(2^3)
True
sage: y.pth_root(2)
b^11 + b^10 + b^9 + b^7 + b^5 + b^4 + b^2 + b
```

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

See `square_root()`.

**EXAMPLES:**

```
sage: k.<a> = GF(3^17)
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 + 2
```

**square_root** \((extend=False, all=False)\)

The square root function.

**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).

**EXAMPLES:**
Finite Fields

\texttt{sage: F = FiniteField(7^2, 'a')}
\texttt{sage: F(2).square_root()}
\texttt{4}
\texttt{sage: F(3).square_root()}
\texttt{2*a + 6}
\texttt{sage: F(3).square_root()**2}
\texttt{3}
\texttt{sage: F(4).square_root()}
\texttt{2}
\texttt{sage: K = FiniteField(7^3, alpha, impl='pari_ffelt')}
\texttt{sage: K(3).square_root()}
\texttt{Traceback (most recent call last):
...}
\texttt{ValueError: must be a perfect square.}

\textbf{to\_bytes (byteorder='big')}

Return an array of bytes representing an integer.

Internally relies on the python \texttt{int.to\_bytes()} method. Length of byte array is determined from the field's order.

INPUT:

- \texttt{byteorder} – str (default: "big"); determines the byte order of the output; can only be "big" or "little"

EXAMPLES:

\texttt{sage: F.<z5> = GF(3^5)}
\texttt{sage: a = z5^4 + 2*z5^3 + 1}
\texttt{sage: a.to\_bytes()}
\texttt{b'\x88'}

\texttt{sage: F.<z3> = GF(163^3)}
\texttt{sage: a = 136*z3^2 + 10*z3 + 125}
\texttt{sage: a.to\_bytes()}
\texttt{b'7\xa3'}

\textbf{to\_integer (reverse=False)}

Return an integer representation of this finite field element obtained by lifting its representative polynomial to \(\mathbb{Z}\) and evaluating it at the characteristic \(p\).

If \texttt{reverse} is set to \texttt{True} (default: \texttt{False}), the list of coefficients is reversed prior to evaluation.

Inverse of \texttt{sage.rings.finite\_rings.finite\_field\_base.FiniteField.\from\_integer()}. 

EXAMPLES:

\texttt{sage: F.<t> = GF(7^5)}
\texttt{sage: F(5).to\_integer()}
\texttt{5}
\texttt{sage: t.to\_integer()}
\texttt{7}
\texttt{sage: (t^2).to\_integer()}
\texttt{49}
\texttt{sage: (t^2+1).to\_integer()}
\texttt{50}

(continues on next page)
sage: (t^2+t+1).to_integer()
57

sage: F.<t> = GF(2^8)
sage: u = F.from_integer(0xd1)
sage: bin(u.to_integer(False))
'0b11010001'
sage: bin(u.to_integer(True))
'0b10001011'

trace()
Return the trace of this element, which is the sum of the Galois conjugates.

EXAMPLES:

sage: S.<a> = GF(5^3); S
Finite Field in a of size 5^3
sage: a.trace()
0
sage: a.charpoly(t)
t^3 + 3*t + 3
sage: a + a^5 + a^25
0
sage: z = a^2 + a + 1
sage: z.trace()
2
sage: z.charpoly(t)
t^3 + 3*t^2 + 2*t + 2
sage: z + z^5 + z^25
2

class sage.rings.finite_rings.element_base.FiniteRingElement
Bases: CommutativeRingElement
to_bytes (byteorder='big')
Return an array of bytes representing an integer.

Internally relies on the python int.to_bytes() method. Length of byte array is determined from the field's order.

INPUT:

- byteorder – str (default: "big"); determines the byte order of input_bytes; can only be "big" or "little"

EXAMPLES:

sage: F = GF(65537)
sage: a = F(8726)
sage: a.to_bytes()
b'

sage.rings.finite_rings.element_base.is_FiniteFieldElement(x)
Return True if x is a finite field element.

This function is deprecated.

EXAMPLES:

2.3. Base class for finite field elements
2.4 Homset for finite fields

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

EXAMPLES:

```sage
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
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: 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]

(continues on next page)
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
]

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

2.5 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 # random
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) # random
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 $f$:

```python
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 + ...
This is not partially defined because
sage: a = k.random_element()
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: ...
```

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

```python
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...
```

Construction of Frobenius endomorphisms:

```python
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:

```python
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:
And that comparisons work:

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

AUTHOR:

- Xavier Caruso (2012-06-29)

```python
class sage.rings.finite_rings.hom_finite_field.FiniteFieldHomomorphism_generic:
    Bases: RingHomomorphism_im_gens

    A class implementing embeddings between finite fields.

    is_injective():
        Return True since a embedding between finite fields is always injective.

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

    is_surjective():
        Return True if this embedding is surjective (and hence an isomorphism).

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

    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:
        ```python
        ```
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 # random
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: a = k.random_element()
sage: b = k.random_element()

sage: g(f(a) + f(b)) == a + b
True

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: ...

class sage.rings.finite_rings.hom_finite_field.FrobeniusEndomorphism_finite_field

Bases: 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 # random
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) # random
4*t^5 + 2*t^4 + 4*t^2 + t

sage: embed(tfixed) == embed.im_gens()[0]
True
**inverse()**

Return the inverse of this Frobenius endomorphism.

**EXAMPLES:**

```python
sage: k.<a> = GF(7^11)
sage: f = k.frobenius_endomorphism(5)
sage: (f.inverse() * f).is_identity()
```

**is_identity()**

Return True if this morphism is the identity morphism.

**EXAMPLES:**

```python
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:**

```python
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:**

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

**order()**

Return the order of this endomorphism.

**EXAMPLES:**

```python
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 endomorphism 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: Section

    A class implementing sections of embeddings between finite fields.
CHAPTER THREE

PRIME FIELDS

3.1 Finite prime fields

AUTHORS:

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

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

Bases: FiniteField, IntegerModRing_generic

Finite field of order $p$ where $p$ is prime.

EXAMPLES:

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

sage: FiniteField(next_prime(1000))  # needs sage.rings.finite_rings
Finite Field of size 1009
```

characteristic()

Return the characteristic of code{self}.

EXAMPLES:

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

construction()

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

EXAMPLES:

```python
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: # needs sage.rings.finite_rings
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: # needs sage.rings.finite_rings
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()
sage: 5
```

polynomial(name=None)

Returns the polynomial name.

EXAMPLES:

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

### 3.2 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: `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: `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:
finite rings, release 10.3

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: SectionFiniteFieldHomomorphism_generic
4.1 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.

```python
class sage.rings.finite_rings.finite_field_pari_ffelt.FiniteField_pari_ffelt(p, modulus, name=None):
    Bases: 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()  # (continues on next page)
```
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 :class:`FiniteFieldElement_pari_ffelt`

**characteristic()**

Return the characteristic of :obj:`self`.

```
EXAMPLES:

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

**degree()**

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

```
EXAMPLES:

sage: F = FiniteField(3^20, 'a', impl='pari_ffelt')
sage: F.degree()
20
```
Gen \( n=0 \)

Return a generator of \( \text{self} \) over its prime field, which is a root of \( \text{self.modulus()} \).

INPUT:

- \( n \) – must be 0

OUTPUT:

An element \( a \) of \( \text{self} \) such that \( \text{self.modulus()}(a) == 0 \).

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

**Examples:**

```python
sage: R.<x> = PolynomialRing(GF(2))
sage: FiniteField(2^4, 'b', impl='pari_ffelt').gen()
```

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

### 4.2 Finite field elements implemented via PARI’s FFELT type

**AUTHORS:**

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

**Class:** \( \text{sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt} \)

Bases: \( \text{FinitePolyExtElement} \)

An element of a finite field implemented using PARI.

**Examples:**

```python
class sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt

Bases: FinitePolyExtElement

An element of a finite field implemented using PARI.

**EXAMPLES:**

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

**charpoly** \( \text{(var='x')} \)

Return the characteristic polynomial of \( \text{self} \).

**INPUT:**

- \( \text{var} \) – string (default: ‘x’): variable name to use.

**EXAMPLES:**
Finite Rings, Release 10.3

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

**frobenius**\((k=1)\)

Return the \(p^k\)th power of self, where \(p\) is the characteristic of the field.

**INPUT:**

- \(k\) – integer (default: 1); must fit in a C `int`

Note that if \(k\) is negative, then this computes the appropriate root.

**is_one()**

Return `True` if `self` equals 1.

**EXAMPLES:**

```python
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:**

```python
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.

**EXAMPLES:**

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

Return True if self equals 0.

**EXAMPLES:**

```
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.

**EXAMPLES:**

```
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, order=None, check=False)

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

**INPUT:**

- base – non-zero field element
- order – integer (optional), the order of the base
- check – boolean (optional, default False): If set, test whether the given order is correct.

**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
sage: p = 2^127-1
sage: F.<t> = GF((p, 3))
sage: elt = F.random_element()^(p^2+p+1)
sage: (elt^2).log(elt, p-1)
2
```

4.2. Finite field elements implemented via PARI’s FFELT type
Passing the `order` argument can lead to huge speedups when factoring the order of the entire unit group is expensive but the order of the base element is much smaller:

```
sage: %timeit (elt^2).log(elt) # not tested
6.18 s ± 85 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
sage: %timeit (elt^2).log(elt, p-1) # not tested
147 ms ± 1.39 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

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
```

**minpoly** *(var='x')*

Return the minimal polynomial of `self`.

**INPUT:**

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

**EXAMPLES:**

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

**multiplicative_order** ()

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

**EXAMPLES:**

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

**polynomial** *(name=None)*

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.

**INPUT:**

- `name` – (optional) variable name

**EXAMPLES:**
pte_power \(k=1\)

Return the \(p^k\)th power of self, where \(p\) is the characteristic of the field.

**INPUT:**

- \(k\) – integer (default: 1); must fit in a C int

Note that if \(k\) is negative, then this computes the appropriate root.

\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 \texttt{ValueError} is raised if the element is not a square in the base field.

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

**EXAMPLES:**

\[
sage: F = FiniteField(7^2, 'a', impl='pari_ffelt')
sage: F(2).sqrt()  
4
sage: F(3).sqrt() in (2*F.gen() + 6, 5*F.gen() + 1)  
True
\]
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 + 2

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

EXAMPLES:

sage: # needs sage.modules
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

FINITE FIELDS USING GIVARO

5.1 Givaro finite fields

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.

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

Bases: `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:

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

(continues on next page)
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 $a \cdot b - c$.

INPUT:

- $a, b, c$ – $\text{FiniteField}_\text{givaroElement}$

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 $a \cdot b + c$. This is faster than multiplying $a$ and $b$ first and adding $c$ to the result.

INPUT:

- $a, b, c$ – $\text{FiniteField}_\text{givaroElement}$

EXAMPLES:
```python
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 \( c - a \times b \).

**INPUT:**

- \( a, b, c \) — \texttt{FiniteField\_givaroElement}

**EXAMPLES:**

```python
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:**

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

**degree()**

If the cardinality of \texttt{self} is \( p^n \), then this returns \( n \).

**OUTPUT:**

Integer – the degree

**EXAMPLES:**

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

**fetch_int** (*args*, **kwargs)

Deprecated: Use \texttt{from\_integer()} instead. See github issue #33941 for details.

**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:**

```python
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)

`from_integer(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)
k.from_integer(8)
a^3
```
```sage
e = k.from_integer(151); e
a^7 + a^4 + a^2 + a + 1
```
```sage
2^7 + 2^4 + 2^2 + 2 + 1
```
```sage
151
```

`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)
```
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 \( \text{self} \).

INPUT:
- \( n \) – integer representation of a finite field element

OUTPUT:
log representation of \( n \)

EXAMPLES:

```python
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 \( \text{self} \); 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:

```python
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 \( \text{self} \).

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

**prime_subfield()**

Return the prime subfield \( F_p \) of self if \( self \) is \( F_{p^n} \).

**EXAMPLES:**

```python
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:**

```python
sage: k = GF(23**3, 'a')
sage: e = k.random_element()
sage: e.parent() is k
True
sage: type(e)
<class 'sage.rings.finite_rings.element_givaro.FiniteField_givaroElement'>
sage: P.<x> = PowerSeriesRing(GF(3^3, 'a'))
sage: P.random_element(5).parent() is P
True
```

### 5.2 Givaro finite 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:**

```python
sage: k = GF(5); type(k)
<class 'sage.rings.finite_rings.finite_field_prime_modn_with_category'>
sage: k = GF(5^2,'c'); type(k)
<class 'sage.rings.finite_rings.finite_field_givaro_with_category'>
sage: k = GF(2^16,'c'); type(k)
```
AUTHORS:

• Martin Albrecht <malb@informatik.uni-bremen.de> (2006-06-05)
• William Stein (2006-12-07): editing, lots of docs, etc.

class sage.rings.finite_rings.element_givaro.Cache_givaro

Bases: Cache_base

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:

```python
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()
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:

```
```python
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:**

```python
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:**

```python
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(map=False).gen(1); e
(0, 1, 0, 0, 0, 0, 0, 0)
sage: k(e) #indirect doctest
a
```

**exponent()**

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

**EXAMPLES:**

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

**fetch_int(number)**

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

**EXAMPLES:**

```python
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()**

Return a generator of the field.

**EXAMPLES:**

```python
```
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 a 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\( () \)

Return the order of this field.

**EXAMPLES:**

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

order_c\( () \)

Return the order of this field.

**EXAMPLES:**
random_element(*args, **kwds)

Return a random element of self.

EXAMPLES:

```sage
sage: k = GF(23**3, 'a')
sage: e = k._cache.random_element()
sage: e.parent() is k
True
sage: type(e)
<class 'sage.rings.finite_rings.element_givaro.FiniteField_givaroElement'>
sage: P.<x> = PowerSeriesRing(GF(3^3, 'a'))
sage: P.random_element(5).parent() is P
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
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]
```
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 \( b^n = \text{self} \).

**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
```

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
```

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)
<class '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:**

```python
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)
Traceback (most recent call last):
...
ValueError: must be a perfect square.
```

```python
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:**

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

```python
sage.rings.finite_rings.element_givaro.unpickle_Cache_givaro

    `(parent, p, k, modulus, rep, cache)`

    **EXAMPLES:**

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

```python
sage.rings.finite_rings.element_givaro.unpickle_FiniteField_givaroElement

    `(parent, x)`
```
5.3 Givaro finite field morphisms

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)

```python
class sage.rings.finite_rings.hom_finite_field_givaro.FiniteFieldHomomorphism_givaro
    Bases: FiniteFieldHomomorphism_generic
class sage.rings.finite_rings.hom_finite_field_givaro.FrobeniusEndomorphism_givaro
    Bases: 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
# random
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)
# random
4*t^5 + 2*t^4 + 4*t^2 + t
```
```
6.1 Finite fields of characteristic 2

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

Bases: `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:

```python
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(6397)
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')
```

(continues on next page)
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(*args, **kwds)

Deprecated: Use from_integer() instead. See github issue #33941 for details.

from_integer(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.from_integer(2^43 + 2^15 + 1)
a^43 + a^15 + 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:

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

order()

Return the cardinality of this field.

EXAMPLES:

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

prime_subfield()

Return the prime subfield \( F_p \) of self if self is \( F_{pn} \).

EXAMPLES:

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

6.2 Elements of 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 \( n \geq 16 \).

AUTHORS:

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

class sage.rings.finite_rings.element_ntl_gf2e.Cache_ntl_gf2e

    Bases: Cache_base

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 github issue #12062).
degree()

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

```sage
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

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

import_data(e)

```sage
sage: k.<a> = GF(2^17)
sage: V = k.vector_space(map=False)
sage: v = [1,0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0]
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
```

order()

Return the cardinality of the field.

```sage
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.

```sage
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, 1]
```

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
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
```

**is_one()**

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

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

**EXAMPLES:**

```sage
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 *F*₂ⁿ is a square.

**EXAMPLES:**

```sage
sage: k.<a> = GF(2^18)
sage: e = k.random_element()
sage: e.parent() is k
True
sage: e.is_square()
True
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
sage: k.<a> = GF(2^17)
sage: a.is_unit()
True
```

(continues on next page)
sage: k(0).is_unit()
False

log(base)

Compute an integer $x$ such that $b^x = a$, where $a$ is self and $b$ is base.

INPUT:

- base – finite field element

OUTPUT:

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

ALGORITHM: pari:fflog

EXAMPLES:

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

Big instances used to take a very long time before github issue #32842:

sage: g = GF(2^61).gen()
sage: g.log(g^7)
1976436865040309101

AUTHORS:

- David Joyner and William Stein (2005-11)
- Lorenz Panny (2021-11): use PARI’s pari:fflog instead of sage.groups.generic.discrete_log()

minpoly(var='x')

Return the minimal polynomial of self, which is the smallest degree polynomial $f \in \mathbb{F}_2[x]$ 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^20 + x^19 + x^16 + x^15 + x^11 + x^9 + x^8 + x^6 + x^5 + x^3 + 1
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:**

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

```python
sage: from sage.rings.polynomial.polynomial_element import Polynomial
sage: isinstance(e.polynomial(), Polynomial)
True
sage: e.polynomial('x')
x^15 + x^13 + x^11 + x^10 + x^9 + x^8 + x^7 + x^4 + x + 1
```

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

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

**EXAMPLES:**

```python
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
```

This failed before github issue #4899:

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

**trace** *

Return the trace of self.

**EXAMPLES:**

```python
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
```

(continues on next page)
weight()

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

EXAMPLES:

```python
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 + a^6 + a^4 + a^3 + a^2
sage: b.weight()
18
```

```python
sage.rings.finite_rings.element_ntl_gf2e.unpickleFiniteField_ntl_gf2eElement
```

EXAMPLES:

```python
sage: k.<a> = GF(2^20)
sage: e = k.random_element()
sage: f = loads(dumps(e))  # indirect doctest
sage: e == f
True
```
7.1 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 $\mathbb{F}_p$.

EXAMPLES:

```sage
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 7)
sage: P = K.ideal(29).factor()[0][0]
sage: k = K.residue_field(P); 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
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 (3/2*a^2 - 5/2*a + 4))
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 $\mathbb{Z}$:

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

And for polynomial rings over finite fields:

```sage
sage: # needs sage.rings.finite_rings
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 Field of size 5
```
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.

```python
class sage.rings.finite_rings.residue_field.LiftingMap
    Bases: Section

Lifting map from residue class field to number field.

EXAMPLES:

sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
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 generated by a 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
```

```python
class sage.rings.finite_rings.residue_field.ReductionMap
    Bases: 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: # needs sage.rings.number_field sage.symbolic
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:
```
(continues on next page)
From: Number Field in sqrt17 with defining polynomial \(x^2 - 17\) with \(\sqrt{17} = 4.123105625617660?\)
To: Residue field in \(\sqrt{17}\)bar of Fractional ideal \(5\)

```
sage: # needs sage.rings.finite_rings
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 t over Finite Field of size 1048583
```

**section()**

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

**EXAMPLES:**

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
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: # needs sage.rings.finite_rings
sage: R.<t> = GF(2[^21]); 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()  
```

7.1. Finite residue fields
class sage.rings.finite_rings.residue_field.ResidueFieldFactory

Bases: 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.
- \( \text{name} \) – the variable name for the finite field created. Defaults to the name of the number field variable but with bar placed after it.
- \( \text{check} \) – whether or not to check if \( p \) is prime.

OUTPUT:

The residue field at the prime \( p \).

EXAMPLES:

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

The result is cached:

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

It also works for polynomial rings:

```
sage: R.<t> = GF(31)[]
sage: P = R.ideal(t^5 + 2*t + 11)
sage: ResidueField(P)  # needs sage.rings.finite_rings
Residue field in tbar of Principal ideal (t^5 + 2*t + 11) of Univariate Polynomial Ring in t over Finite Field of size 31
sage: ResidueField(P) is ResidueField(P)  # needs sage.rings.finite_rings
True
sage: k = ResidueField(P); k.order()  # needs sage.rings.finite_rings
28629151
```

An example where the generator of the number field doesn’t generate the residue class field:
An example where the residue class field is large but of degree 1:

```
sage: # needs sage.rings.number_field
sage: K.<a> = NumberField(x^3 - 875)
sage: 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: # needs sage.rings.finite_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 Field of size 262147
sage: k(t)
5
```

In this example, 2 is an inessential discriminant divisor, so divides the index of $\mathbb{Z}[a]$ in the maximal order for all $a$:

```
sage: # needs sage.rings.number_field
sage: K.<a> = NumberField(x^3 + x^2 - 2*x + 8)
sage: 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
```

`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:**
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:**

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

```plaintext
class \texttt{sage.rings.finite_rings.residue_field.ResidueFieldHomomorphism_global}

**Bases:** \texttt{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:**

```plaintext
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
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: # needs sage.rings.number_field
sage: phi = k.coerce_map_from(OK); phi
Ring morphism:
  From: Maximal Order generated by a 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: # needs sage.rings.finite_rings
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
```

lift \((x)\)

Returns a lift of \(x\) to the Order, returning a “polynomial” in the generator with coefficients between 0 and \(p - 1\).

**EXAMPLES:**

```plaintext
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 7)
sage: P = K.ideal(29).factor()[0][0]
sage: k = K.residue_field(P)
sage: f = k.coerce_map_from(K)
sage: a = K(3)

sage: f.lift(13*a + 5)
13*a + 5
sage: f.lift(12821*a + 918)
3*a + 19
```

```plaintext
sage: # needs sage.rings.finite_rings
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: a^2 + 5*a + 1
```

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

**EXAMPLES:**

```plaintext
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^5 - 5*x + 2)
sage: P = K.ideal(47).factor()[0][0]
sage: k = K.residue_field(P)
sage: s = k.coerce_map_from(K)
sage: s(k.gen())
```

`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:**

```plaintext
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
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)
sage: s = f.section(); s
```

(continues on next page)
Lifting map:
From: Residue field in bbar of Fractional ideal (53, b^2 + 23*b + 8)
To: Maximal Order generated by b in Number Field in b
    with defining polynomial x^5 + 17*x + 1
sage: s(l.gen()).parent()
Maximal Order generated by b in Number Field in b with defining polynomial x^5 + 17*x + 1
sage: # needs sage.rings.finite_rings
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 in t over Finite Field of size 17
To: Univariate Polynomial Ring in t over Finite Field of size 17

class sage.rings.finite_rings.residue_field.ResidueField_generic(p)
Bases: Field
The class representing a generic residue field.

EXAMPLES:

sage: # needs sage.rings.number_field
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: # needs sage.libs.linbox
<class 'sage.rings.finite_rings.residue_field_givaro.ResidueFiniteField_givaro_with_category'>

construction()
Construction of this residue field.

OUTPUT:
An AlgebraicExtensionFunctor and the number field that this residue field has been obtained from.
The residue field is determined by a prime (fractional) ideal in a number field. If this ideal can be coerced into a
different number field, then the construction functor applied to this number field will return the corresponding
residue field. See github issue #15223.

EXAMPLES:
Residue field in zbar of Fractional ideal (17)

sage: F, R = k.construction()
sage: F
AlgebraicExtensionFunctor
sage: R
Cyclotomic Field of order 7 and degree 6
sage: F(R) is k
True
sage: F(ZZ)
Residue field of Integers modulo 17
sage: F(CyclotomicField(49))
Residue field in zbar of Fractional ideal (17)

ideal()

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

EXAMPLES:

lift(x)

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

EXAMPLES:
13*a + 5
\[\text{sage: } k.lift(12821*b + 918)\]
3*a + 19

\[\text{sage: } \# \text{ needs sage.rings.finite_rings}\]
\[\text{sage: } R.<t> = GF(17)[]; P = R.ideal(t^3 + t^2 + 7)\]
\[\text{sage: } k.<a> = P.residue_field()\]
\[\text{sage: } k.lift(a^2 + 5)\]
t^2 + 5

**lift_map()**

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

**EXAMPLES:**

\[\text{sage: } \# \text{ needs sage.rings.number_field sage.symbolic}\]
\[\text{sage: } I = QQ[3^(1/3)].factor(5)[[1][0]]; I\]
Fractional ideal (a - 2)
\[\text{sage: } k = I.residue_field(); k\]
Residue field of Fractional ideal (a - 2)
\[\text{sage: } f = k.lift_map(); f\]
Lifting map:
  From: Residue field of Fractional ideal (a - 2)
  To: Maximal Order generated by a in Number Field in a
  with defining polynomial x^3 - 3 with a = 1.442249570307409?
\[\text{sage: } f.domain()\]
Residue field of Fractional ideal (a - 2)
\[\text{sage: } f.codomain()\]
Maximal Order generated by a in Number Field in a
  with defining polynomial x^3 - 3 with a = 1.442249570307409?
\[\text{sage: } f(k.0)\]
1

\[\text{sage: } \# \text{ needs sage.rings.finite_rings}\]
\[\text{sage: } R.<t> = GF(17)[]; P = R.ideal(t^3 + t^2 + 7)\]
\[\text{sage: } k.<a> = P.residue_field()\]
\[\text{sage: } f = k.lift_map(); f\]
(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 in t over Finite Field of size 17
  To: Univariate Polynomial Ring in t over Finite Field of size 17
\[\text{sage: } f(a^2 + 5)\]
t^2 + 5

**reduction_map()**

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

**EXAMPLES:**

\[\text{sage: } \# \text{ needs sage.rings.number_field sage.symbolic}\]
\[\text{sage: } I = QQ[2^(1/3)].factor(2)[0][0]; I\]
Fractional ideal (a)
\[\text{sage: } k = I.residue_field(); k\]
Residue field of Fractional ideal (a)
\[\text{sage: } pi = k.reduction_map(); pi\]
Partially defined reduction map:

(continues on next page)
From: Number Field in a with defining polynomial $x^3 - 2$
with $a = 1.2599210498948737$
To: Residue field of Fractional ideal (a)
sage: pi.domain()
Number Field in a with defining polynomial $x^3 - 2$ with $a = 1.2599210498948737$
sage: pi.codomain()
Residue field of Fractional ideal (a)
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
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: # needs sage.rings.finite_rings
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 in t over Finite Field of size 17
sage: f(1/t)
12*a^2 + 12*a

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

Bases: ResidueField_generic, FiniteField_prime_modn

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

EXAMPLES:

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

(continues on next page)
sage: k(4)
4
sage: k(c + 5)
21
sage: b + c
3

sage: # needs sage.rings.finite_rings
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 size 7
sage: k(t^2)
4
sage: k.order()
7

7.2 Algebraic closures of finite fields

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

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 $F_n$ and the embeddings between them. This lattice is extended as necessary.

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

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

**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 $F$, take an algebraic closure of the prime field of $F$ and embed $F$ 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
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:**

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

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

```python
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: `FieldElement`

Element of an algebraic closure of a finite field.

**EXAMPLES:**

```python
sage: F = GF(3).algebraic_closure()
sage: F.gen(2)
z2
```

```python
sage: type(F.gen(2))
<\class 'sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField\_\_\_pseudo_conway_with_category.element_class'>
```

**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:**

---

7.2. Algebraic closures of finite fields
• a triple \((\text{field}, \text{element}, \text{morphism})\) where \text{field} is a finite field, \text{element} an element of \text{field} and \text{morphism} a morphism from \text{field} to \text{self.parent()}.

EXAMPLES:

```python
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, \text{field} is not necessarily minimal. We can force it to be minimal using the \text{minimal} option:

```python
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 github issue #16509):

```python
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 are automatic coercions between the various subfields:

```python
sage: a = K.gen(2) + 1
sage: _,b,_ = a.as_finite_field_element()
sage: K4 = K.subfield(4)[0]
sage: K4(b)
z4^3 + z4^2 + z4 + 4
sage: b.minimal_polynomial() == K4(b).minimal_polynomial()
True
sage: K(K4(b)) == K(b)
True
```

You can also use the inclusions that are implemented at the level of the algebraic closure:

```python
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
```

\text{change_level} \((n)\)

Return a representation of \text{self} as an element of the subfield of degree \(n\) of the parent, if possible.
EXAMPLES:

```python
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
```

```python
is_square()
    Return True if self is a square.
    This always returns True.
```

EXAMPLES:

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

```python
minimal_polynomial()
    Return the minimal polynomial of self over the prime field.
```

EXAMPLES:

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

```python
minpoly()
    Return the minimal polynomial of self over the prime field.
```

EXAMPLES:

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

```python
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 (all=False)

Return a square root of self.
If the optional keyword argument all is set to True, return a list of all square roots of self instead.

EXAMPLES:

```
sage: F = GF(3).algebraic_closure()
sage: F.gen(2).sqrt()
z4^3 + z4 + 1
sage: F.gen(2).sqrt(all=True)
z4^3 + z4 + 1, 2*z4^3 + 2*z4 + 2
sage: (F.gen(2)^2).sqrt()
z2
sage: (F.gen(2)^2).sqrt(all=True)
z2, 2*z2
```

class `sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField_generic`

```
Bases: Field

Algebraic closure of a finite field.

Element

alias of `AlgebraicClosureFiniteFieldElement`

canonical_element()  
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
```

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')
```

(continues on next page)
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(); g
Lazy family (...(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

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:

```python
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

Algebraic closure of a finite field, constructed using pseudo-Conway polynomials.

EXAMPLES:

```python
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
```

### 7.3 Routines for Conway and pseudo-Conway polynomials

AUTHORS:

- David Roe
- Jean-Pierre Flori
- Peter Bruin

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

Bases: WithEqualityById, SageObject
A pseudo-Conway lattice over a given finite prime field.

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

- $f_n$ is irreducible.
- In the quotient field $\mathbb{F}_p[x]/(f_n)$, the element $x \mod f_n$ generates the multiplicative group.
- The minimal polynomial of $(x \mod f_n)^{p^n-1}$ 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:

```python
sage: from sage.rings.finite_rings.conway_polynomials import PseudoConwayLattice
sage: PCL = PseudoConwayLattice(2, use_database=False)
sage: PCL.polynomial(3)  # random
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:

```python
sage: PCL = PseudoConwayLattice(2, use_database=False)
sage: PCL.check_consistency(6)
sage: PCL.check_consistency(60)  # long time
```

`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 [HL1999], 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.

EXAMPLES:
sage: # needs sage.rings.finite_rings
sage: from sage.rings.finite_rings.conway_polynomials import *

sage: PCL = PseudoConwayLattice(2, use_database=False)
sage: PCL.polynomial(3)  # random
x^3 + x + 1
sage: PCL.polynomial(4)  # random
x^4 + x^3 + 1
sage: PCL.polynomial(60)  # random
x^60 + x^59 + x^58 + x^55 + x^53 + x^52 + x^51 + x^48 + x^46 + x^45 + ...
-→ x^42 + x^41 + x^39 + x^38 + x^37 + x^35 + x^32 + x^31 + x^30 + x^28 + x^24 ...
-→ + x^22 + x^21 + x^18 + x^17 + x^16 + x^15 + x^14 + x^10 + x^8 + x^7 + x^5 + ...
-→ + x^3 + x^2 + x + 1

sage.rings.finite_rings.conway_polynomials.conway_polynomial \((p, n)\)

Return the Conway polynomial of degree \(n\) over \(\text{GF}(p)\).

If the requested polynomial is not known, this function raises a \texttt{RuntimeError} exception.

\textbf{INPUT:}

\begin{itemize}
  \item \(p\) – prime number
  \item \(n\) – positive integer
\end{itemize}

\textbf{OUTPUT:}

\begin{itemize}
  \item the Conway polynomial of degree \(n\) over the finite field \(\text{GF}(p)\), loaded from a table.
\end{itemize}

\textbf{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 \texttt{ConwayPolynomials()} object, which is the table of Conway polynomials used by this function.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: conway_polynomial(2,5) # needs conway_polynomials
x^5 + x^2 + 1
sage: conway_polynomial(101,5) # needs conway_polynomials
x^5 + x^4 + 99
sage: conway_polynomial(97,101) # needs conway_polynomials
Traceback (most recent call last):
... RuntimeError: requested Conway polynomial not in database.
\end{verbatim}

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

Check whether the Conway polynomial of degree \(n\) over \(\text{GF}(p)\) is known.

\textbf{INPUT:}

\begin{itemize}
  \item \(p\) – prime number
  \item \(n\) – positive integer
\end{itemize}

\textbf{OUTPUT:}

\begin{itemize}
  \item boolean: \texttt{True} if the Conway polynomial of degree \(n\) over \(\text{GF}(p)\) is in the database, \texttt{False} otherwise.
\end{itemize}
If the Conway polynomial is in the database, it can be obtained using the command `conway_polynomial(p, n)`.

EXAMPLES:

```python
sage: exists_conway_polynomial(2,3)  #...
needs conway_polynomials
True
sage: exists_conway_polynomial(2,-1)
False
sage: exists_conway_polynomial(97,200)
False
sage: exists_conway_polynomial(6,6)
False
```
• Index
• Module Index
• Search Page
sage.rings.algebraic_closure_finite_field, 126
sage.rings.finite_rings.conway_polynomials, 133
sage.rings.finite_rings.element_base, 62
sage.rings.finite_rings.element_givaro, 98
sage.rings.finite_rings.element_ntl_gf2e, 109
sage.rings.finite_rings.element_pari_ffelt, 87
sage.rings.finite_rings.finite_field_base, 47
sage.rings.finite_rings.finite_field_constructor, 39
sage.rings.finite_rings.finite_field_givaro, 93
sage.rings.finite_rings.finite_field_ntl_gf2e, 107
sage.rings.finite_rings.finite_field_pari_ffelt, 85
sage.rings.finite_rings.finite_field_prime_modn, 81
sage.rings.finite_rings.hom_finite_field, 74
sage.rings.finite_rings.hom_finite_field_givaro, 106
sage.rings.finite_rings.hom_prime_finite_field, 83
sage.rings.finite_rings.homset, 72
sage.rings.finite_rings.integer_mod, 15
sage.rings.finite_rings.integer_mod_ring, 1
sage.rings.finite_rings.residue_field, 115
Index
get_object() (sage.rings.finite_rings.integer_mod_ring.IntegerModFactory method), 4

ideal() (sage.rings.finite_rings.residue_field.ResidueField_generic method), 123
import_data() (sage.rings.finite_rings.element_ntl_gf2e.Cache_ntl_gf2e method), 110
inclusion() (sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField_generic method), 132
index() (sage.rings.finite_rings.homset.FiniteFieldHomset method), 72
Int_to_IntegerMod (class in sage.rings.finite_rings.integer_mod), 15
int_to_log() (sage.rings.finite_rings.element_givaro.Cache_givaro method), 102
int_to_log() (sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro method), 97
integer_representation() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 64
Integer_to_IntegerMod (class in sage.rings.finite_rings.integer_mod), 34
IntegerMod() (class in module sage.rings.finite_rings.integer_mod), 16
IntegerMod_gmp (class in sage.rings.finite_rings.integer_mod), 28
IntegerMod_hom (class in sage.rings.finite_rings.integer_mod), 29
IntegerMod_int (class in sage.rings.finite_rings.integer_mod), 29
IntegerMod_int64 (class in sage.rings.finite_rings.integer_mod), 32
IntegerMod_to_Integer (class in sage.rings.finite_rings.integer_mod), 33
IntegerMod_to_IntegerMod (class in sage.rings.finite_rings.integer_mod), 34
IntegerModFactory (class in sage.rings.finite_rings.integer_mod_ring), 1
IntegerModRing_generic (class in sage.rings.finite_rings.integer_mod_ring), 4
inverse() (sage.rings.finite_rings.hom_finite_field.FrobeniusEndomorphism_finite_field method), 77
inverses (sage.rings.finite_rings.integer_mod.NativeIntStruct attribute), 35
is_aut() (sage.rings.finite_rings.homset.FiniteFieldHomset method), 73
is_conway() (sage.rings.finite_rings.finite_field_base.FiniteField method), 55
is_field() (sage.rings.finite_rings.finite_field_base.FiniteField method), 55
is_field() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 7
is_FiniteField() (in module sage.rings.finite_rings.finite_field_base), 62
is_FiniteFieldElement() (in module sage.rings.finite_rings.element_base), 71
is_identity() (sage.rings.finite_rings.hom_finite_field.FrobeniusEndomorphism_finite_field method), 78
is_injective() (sage.rings.finite_rings.hom_finite_field.FiniteFieldHomomorphism_generic method), 76
is_injective() (sage.rings.finite_rings.hom_finite_field.FrobeniusEndomorphism_finite_field method), 78
is_injective() (sage.rings.finite_rings.integer_mod.Integer_to_IntegerMod method), 34
is_injective() (sage.rings.finite_rings.integer_mod.IntegerMod_to_IntegerMod method), 34
is_integral_domain() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 8
is_nilpotent() (sage.rings.finite_rings.integer_mod.IntegerMod_to_IntegerMod method), 18
is_noetherian() (sage.rings.finite_rings.integer_mod.IntegerModRing_generic method), 8
is_one() (sage.rings.finite_rings.element_givaro.FiniteField_givaroElement method), 103
is_one() (sage.rings.finite_rings.element_ntl_gf2e.FiniteField_ntl_gf2eElement method), 111
is_one() (sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt method), 88
is_one() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 18
is_one() (sage.rings.finite_rings.integer_mod.IntegerMod_gmp method), 28
is_one() (sage.rings.finite_rings.integer_mod.IntegerMod_int method), 30
is_one() (sage.rings.finite_rings.integer_mod.IntegerMod_int64 method), 32
is_perfect() (sage.rings.finite_rings.finite_field_base.FiniteField method), 55
is_prime_field() (sage.rings.finite_rings.finite_field_base.FiniteField method), 55
is_prime_field() (sage.rings.finite_rings.finite_field_prime_modn.FiniteField_prime_modn method), 82
is_prime_field() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 9
is_PrimeFiniteField() (in module sage.rings.finite_rings.finite_field_constructor), 47
is_primitive_root() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 18
is_square() (sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteFieldElement method), 129
is_square() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 65
is_square() (sage.rings.finite_rings.homset.FiniteFieldHomset method), 73
is_surjective() (sage.rings.finite_rings.hom_finite_field.FiniteFieldHomomorphism_generic method), 76
is_surjective() (sage.rings.finite_rings.hom_finite_field.FrobeniusEndomorphism_finite_field method), 78
is_surjective() (sage.rings.finite_rings.integer_mod.IntegerMod_to_IntegerMod method), 34
is_unique_factorization_domain() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 9
is_unit() (sage.rings.finite_rings.element_givaro.FiniteField_givaroElement method), 103
is_unit() (sage.rings.finite_rings.integer_mod.IntegerField_gmp method), 29
is_unit() (sage.rings.finite_rings.integer_mod.IntegerField_int method), 30
is_unit() (sage.rings.finite_rings.integer_mod.IntegerField_int64 method), 33
is_zero() (sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt method), 88
krull_dimension() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 9

K

L

late_import() (in module sage.rings.finite_rings.integer_mod_ring.finite_field_ntl_gf2e), 109
lift() (sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt method), 89
lift() (sage.rings.finite_rings.integer_mod.IntegerField_gmp method), 29
lift() (sage.rings.finite_rings.integer_mod.IntegerField_int method), 30
lift() (sage.rings.finite_rings.integer_mod.IntegerField_int64 method), 33
lift() (sage.rings.finite_rings.residue_field.ResidueField_generic method), 123
lift_centered() (sage.rings.finite_rings.integer_mod.IntegerField_abstract method), 19
lift_map() (sage.rings.finite_rings.residue_field.ResidueFieldHomomorphism_global method), 121
LiftingMap (class in sage.rings.finite_rings.residue_field), 116
list() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 65
list() (sage.rings.finite_rings.homset.FiniteFieldHomset method), 73
list_of_elements_of_multiplicative_group() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 9
log() (sage.rings.finite_rings.element_givaro.FiniteField_givaroElement method), 104
log() (sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt method), 112
log() (sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt method), 89
log() (sage.rings.finite_rings.integer_mod.IntegerField_abstract method), 20
log_to_int() (sage.rings.finite_rings.element_givaro.Cache_givaro method), 102
log_to_int() (sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro method), 97
lucas() (in module sage.rings.finite_rings.integer_mod), 36
lucas_q1() (in module sage.rings.finite_rings.integer_mod), 37

M
makeNativeIntStruct (in module sage.rings.finite_rings.integer_mod), 37
matrix() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 65
minimal_polynomial() (sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteFieldElement method), 129
minimal_polynomial() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 66
minimal_polynomial() (sage.rings.finite_rings.element_givaro.FiniteField_givaroElement method), 104
minimal_polynomial() (sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt method), 90
minimal_polynomial() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 21

Mod() (in module sage.rings.finite_rings.integer_mod), 35
mod() (in module sage.rings.finite_rings.integer_mod), 37
module
sage.rings.algebraic_closure_finite_field, 126
sage.rings.finite_rings.conway_polynomials, 133
sage.rings.finite_rings.element_base, 62
sage.rings.finite_rings.element_givaro, 98
sage.rings.finite_rings.element_ntl_gf2e, 109
sage.rings.finite_rings.element_pari_ffelt, 87
sage.rings.finite_rings.finite_field_base, 47
sage.rings.finite_rings.finite_field_constructor, 39
sage.rings.finite_rings.finite_field_givaro, 93
sage.rings.finite_rings.finite_field_ntl_gf2e, 107
sage.rings.finite_rings.finite_field_pari_ffelt, 85
sage.rings.finite_rings.finite_field_prime_modn, 81
sage.rings.finite_rings.hom_finite_field, 74
sage.rings.finite_rings.hom_finite_field_givaro, 106
sage.rings.finite_rings.hom_prime_finite_field, 83
sage.rings.finite_rings.homset, 72
sage.rings.finite_rings.integer_mod_ring, 15
sage.rings.finite_rings.integer_mod_ring.Mod, 1
sage.rings.finite_rings.residue_field, 115
modulus() (sage.rings.finite_rings.finite_field_base.FiniteField method), 56
modulus() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 9
modulus() (sage.rings.finite_rings.integer_mod_ring.IntegerModAbstract method), 21
multiplicative_generator() (sage.rings.finite_rings.finite_field_base.FiniteField method), 57
multiplicative_generator() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 10
multiplicative_generator() (sage.rings.finite_rings.integer_mod_ring.IntegerModRingGeneric method), 10
multiplicative_order() (sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteFieldElement method), 129
multiplicative_order() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 66
multiplicative_order() (sage.rings.finite_rings.element_givaro.FiniteField_givaroElement method), 104
multiplicative_order() (sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt method), 90
multiplicative_order() (sage.rings.finite_rings.integer_mod_ring.IntegerModAbstract method), 21
multiplicative_subgroups() (sage.rings.finite_rings.integer_mod_ring.IntegerMod-
Ring_generic method), 11

NativeIntStruct (class in sage.rings.finite_rings.integer_mod), 35

ngens() (sage.rings.algebraic_closure_field.AlgebraicClosureFiniteFieldMethod), 132

ngens() (sage.rings.finite_rings.finite_field_base.FiniteFieldMethod), 57

norm() (sage.rings.finite_rings.element_base.FinitePolyExtElementMethod), 67

norm() (sage.rings.finite_rings.integer_mod.IntegerModAbstractMethod), 21

nth_root() (sage.rings.algebraic_closure_field.AlgebraicClosureFiniteFieldElementMethod), 130

nth_root() (sage.rings.finite_rings.element_base.FinitePolyExtElementMethod), 67

nth_root() (sage.rings.finite_rings.integer_mod.IntegerModAbstractMethod), 22

O

order() (sage.rings.finite_rings.element_givaro.Cache_givaroMethod), 102

order() (sage.rings.finite_rings.element_ntl_gf2e.Cache_ntl_gf2eMethod), 110

order() (sage.rings.finite_rings.finite_field_base.FiniteFieldMethod), 57

order() (sage.rings.finite_rings.finite_field_givaro.FiniteFieldGivaroMethod), 97

order() (sage.rings.finite_rings.finite_field_prime_modn.FiniteFieldPrimeModnMethod), 109

order() (sage.rings.finite_rings.finite_field_prime_modn.FiniteFieldPrimeModnMethod), 82

order() (sage.rings.finite_rings.hom_field.FrobeniusEndomorphismFiniteFieldMethod), 78

order() (sage.rings.finite_rings.homset.FiniteFieldHomsetMethod), 74

order() (sage.rings.finite_rings.integer_mod_ring.IntegerModRingGenericMethod), 11

order_c() (sage.rings.finite_rings.element_givaro.Cache_givaroMethod), 102

P

polynomial() (sage.rings.finite_rings.conway_polynomials.PseudoConwayLatticeMethod), 134

polynomial() (sage.rings.finite_rings.element_givaro.FiniteFieldGivaroElementMethod), 104

polynomial() (sage.rings.finite_rings.element_ntl_gf2e.Cache_ntl_gf2eMethod), 110

polynomial() (sage.rings.finite_rings.element_ntl_gf2e.FiniteFieldNtlGf2eElementMethod), 113

polynomial() (sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElementPari_ffeltMethod), 90

polynomial() (sage.rings.finite_rings.finite_field_base.FiniteFieldMethod), 57

polynomial() (sage.rings.finite_rings.finite_field_prime_modn.FiniteFieldPrimeModnMethod), 83

polynomial() (sage.rings.finite_rings.integer_mod.IntegerModAbstractMethod), 23

polynomial_ring() (sage.rings.finite_rings.finite_field_base.FiniteFieldMethod), 58

power() (sage.rings.finite_rings.hom_field.FrobeniusEndomorphismFiniteFieldMethod), 78

precompute_table() (sage.rings.finite_rings.integer_mod.NativeIntStructMethod), 35

prime_subfield() (sage.rings.finite_rings.finite_field_givaro.FiniteFieldGivaroMethod), 98

prime_subfield() (sage.rings.finite_rings.finite_field_ntl_gf2e.FiniteFieldNtlGf2eMethod), 109

PseudoConwayLattice (class in sage.rings.finite_rings.conway_polynomials), 133

pth_power() (sage.rings.algebraic_closure_field.AlgebraicClosureFiniteFieldElementMethod), 130

pth_power() (sage.rings.finite_rings.finite_field_base.FinitePolyExtElementMethod), 68

pth_power() (sage.rings.finite_rings.element_base.FinitePolyExtElementMethod), 69

Q

quadratic_nonresidue() (sage.rings.finite_rings.integer_mod_ring.IntegerModRingGenericMethod), 11

random_element() (sage.rings.finite_rings.element_givaro.Cache_givaroMethod), 103

Index 147
random_element() (sage.rings.finite_rings.finite_field_base.FiniteField method), 58
random_element() (sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro method), 98
random_element() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 11
rational_reconstruction() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 23
reduction_map() (sage.rings.finite_rings.residue_field.ResidueField_generic method), 124
ReductionMap (class in sage.rings.finite_rings.residue_field), 116
repr (sage.rings.finite_rings.element_givaro.Cache_givaro attribute), 103
ResidueField_generic (class in sage.rings.finite_rings.residue_field), 122
ResidueFieldFactory (class in sage.rings.finite_rings.residue_field), 117
ResidueFieldHomomorphism_global (class in sage.rings.finite_rings.residue_field), 120
ResidueFiniteField_prime_modn (class in sage.rings.finite_rings.residue_field), 125
S
sage.rings.algebraic_closure_finite_field module, 126
dsage.rings.finite_rings.conway_polynomials module, 133
dsage.rings.finite_rings.element_base module, 62
dsage.rings.finite_rings.element_givaro module, 98
dsage.rings.finite_rings.element_ntl_gf2e module, 109
dsage.rings.finite_rings.element_pari_ffelt module, 87
dsage.rings.finite_rings.finite_field_base module, 47
dsage.rings.finite_rings.finite_field_constructor module, 39
dsage.rings.finite_rings.finite_field_givaro module, 93
dsage.rings.finite_rings.finite_field_ntl_gf2e module, 107
dsage.rings.finite_rings.finite_field_pari_ffelt module, 85
dsage.rings.finite_rings.finite_field_prime_modn module, 81
dsage.rings.finite_rings.hom FiniteField module, 74
dsage.rings.finite_rings.hom FiniteField_givaro module, 106
sage.rings.finite_rings.hom_primeFiniteField module, 83
sage.rings.finite_rings.homset module, 72
sage.rings.finite_rings.integer_mod module, 15
sage.rings.finite_rings.integer_mod_ring module, 1
sage.rings.finite_rings.residue_field module, 115
section() (sage.rings.finite_rings.homFiniteField.FiniteFieldHomomorphism_generic method), 76
section() (sage.rings.finite_rings.integer_mod.Integer_to_IntegerMod method), 35
section() (sage.rings.finite_rings.residue_field.ReductionMap method), 117
section() (sage.rings.finite_rings.residue_field.ResidueFieldHomomorphism_global method), 121
SectionFiniteFieldHomomorphism_generic (class in sage.rings.finite_rings.hom FiniteField), 79
SectionFiniteFieldHomomorphism_givaro (class in sage.rings.finite_rings.hom FiniteField_givaro), 106
SectionFiniteFieldHomomorphism_prime (class in sage.rings.finite_rings.hom_primeFiniteField), 84
some_elements() (sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField_generic method), 132
some_elements() (sage.rings.finite_rings.finite_field_base.FiniteField method), 59
sqrt() (sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteFieldElement method), 130
<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sqrt()</code></td>
<td>(sage.rings.finite_rings.element_base.FinitePolyExtElement method)</td>
<td>69</td>
</tr>
<tr>
<td><code>sqrt()</code></td>
<td>(sage.rings.finite_rings.element_givaro.FiniteField_givaroElement method)</td>
<td>104</td>
</tr>
<tr>
<td><code>sqrt()</code></td>
<td>(sage.rings.finite_rings.element_ntl_gf2e.FiniteField_ntl_gf2eElement method)</td>
<td>113</td>
</tr>
<tr>
<td><code>sqrt()</code></td>
<td>(sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt method)</td>
<td>91</td>
</tr>
<tr>
<td><code>sqrt()</code></td>
<td>(sage.rings.finite_rings.integer_mod.IntegerModElement method)</td>
<td>24</td>
</tr>
<tr>
<td><code>sqrt()</code></td>
<td>(sage.rings.finite_rings.integer_mod.IntegerMod_int method)</td>
<td>30</td>
</tr>
<tr>
<td><code>square_root()</code></td>
<td>(sage.rings.finite_rings.element_base.FinitePolyExtElement method)</td>
<td>69</td>
</tr>
<tr>
<td><code>square_root()</code></td>
<td>(sage.rings.finite_rings.integer_mod.IntegerModAbstract method)</td>
<td>26</td>
</tr>
<tr>
<td><code>square_root_mod_prime()</code></td>
<td>(in module sage.rings.finite_rings.integer_mod)</td>
<td>37</td>
</tr>
<tr>
<td><code>square_root_mod_prime_power()</code></td>
<td>(in module sage.rings.finite_rings.integer_mod)</td>
<td>38</td>
</tr>
<tr>
<td><code>square_roots_of_one()</code></td>
<td>(sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method)</td>
<td>12</td>
</tr>
<tr>
<td><code>subfield()</code></td>
<td>(sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField_generic method)</td>
<td>132</td>
</tr>
<tr>
<td><code>subfield()</code></td>
<td>(sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method)</td>
<td>59</td>
</tr>
<tr>
<td><code>subfields()</code></td>
<td>(sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method)</td>
<td>60</td>
</tr>
<tr>
<td><code>table</code></td>
<td>(sage.rings.finite_rings.integer_mod.NativeIntStruct attribute)</td>
<td>36</td>
</tr>
<tr>
<td><code>to_bytes()</code></td>
<td>(sage.rings.finite_rings.element_base.FinitePolyExtElement method)</td>
<td>70</td>
</tr>
<tr>
<td><code>to_bytes()</code></td>
<td>(sage.rings.finite_rings.element_base.FiniteRingElement method)</td>
<td>71</td>
</tr>
<tr>
<td><code>to_integer()</code></td>
<td>(sage.rings.finite_rings.element_base.FinitePolyExtElement method)</td>
<td>70</td>
</tr>
<tr>
<td><code>trace()</code></td>
<td>(sage.rings.finite_rings.element_base.FinitePolyExtElement method)</td>
<td>71</td>
</tr>
<tr>
<td><code>trace()</code></td>
<td>(sage.rings.finite_rings.element_ntl_gf2e.FiniteField_ntl_gf2eElement method)</td>
<td>113</td>
</tr>
<tr>
<td><code>trace()</code></td>
<td>(sage.rings.finite_rings.integer_mod.IntegerModAbstract method)</td>
<td>27</td>
</tr>
<tr>
<td><code>unit_gens()</code></td>
<td>(sage.rings.finite_rings.integer_mod.IntegerModRing_generic method)</td>
<td>12</td>
</tr>
<tr>
<td><code>unit_group()</code></td>
<td>(sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method)</td>
<td>13</td>
</tr>
<tr>
<td><code>unit_group_exponent()</code></td>
<td>(sage.rings.finite_rings.integer_field_base.FiniteField method)</td>
<td>60</td>
</tr>
<tr>
<td><code>unit_group_exponent()</code></td>
<td>(sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method)</td>
<td>14</td>
</tr>
<tr>
<td><code>unit_group_order()</code></td>
<td>(sage.rings.finite_rings.integer_mod_ring.IntegerModRingGeneric method)</td>
<td>14</td>
</tr>
<tr>
<td><code>unpickle_Cache_givaro()</code></td>
<td>(in module sage.rings.finite_rings.element_givaro)</td>
<td>105</td>
</tr>
<tr>
<td><code>unpickleFiniteField_ext()</code></td>
<td>(in module sage.rings.finite_rings.finite_field_base)</td>
<td>62</td>
</tr>
<tr>
<td><code>unpickleFiniteField_givaroElement()</code></td>
<td>(in module sage.rings.finite_rings.element_givaro)</td>
<td>105</td>
</tr>
<tr>
<td><code>unpickleFiniteField_prm()</code></td>
<td>(in module sage.rings.finite_rings.finite_field_base)</td>
<td>62</td>
</tr>
<tr>
<td><code>unpickleFiniteFieldElement_pari_ffelt()</code></td>
<td>(in module sage.rings.finite_rings.element_pari_ffelt)</td>
<td>92</td>
</tr>
<tr>
<td><code>unpickleFiniteField_ntrl_gf2eElement()</code></td>
<td>(in module sage.rings.finite_rings.integer_mod_ring)</td>
<td>114</td>
</tr>
<tr>
<td><code>valuation()</code></td>
<td>(sage.rings.finite_rings.integer_mod.IntegerModAbstract method)</td>
<td>27</td>
</tr>
<tr>
<td><code>weight()</code></td>
<td>(sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method)</td>
<td>114</td>
</tr>
<tr>
<td><code>zeta()</code></td>
<td>(sage.rings.finite_rings.integer_mod_ring.IntegerModRingGeneric method)</td>
<td>61</td>
</tr>
<tr>
<td><code>zeta_order()</code></td>
<td>(sage.rings.finite_rings.integer_field_base.FiniteField method)</td>
<td>61</td>
</tr>
</tbody>
</table>

**Index**

149