CONTENTS

1 Finite Rings ........................................ 1
2 Finite Fields ....................................... 41
3 Prime Fields ....................................... 83
4 Finite Fields Using Pari ............................. 87
5 Finite Fields Using Givaro ......................... 95
6 Finite Fields of Characteristic 2 Using NTL ...... 111
7 Miscellaneous ...................................... 119
8 Indices and Tables ................................ 143
Python Module Index ................................. 145
Index .................................................. 147
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 \(F_p\). In particular, there is a canonical map to \(F_p\), 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 \texttt{sage.rings.finite_rings.integer_mod_ring.IntegerModFactory}

Bases: \texttt{UniqueFactory}

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

\textbf{INPUT}:

- \texttt{order} – integer (default: 0); positive or negative
- \texttt{is\_field} – bool (default: False); assert that the order is prime and hence the quotient ring belongs to the category of fields
- \texttt{category} (optional) - the category that the quotient ring belongs to.

\textbf{Note}: The optional argument \texttt{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 \texttt{is\_field} is true, then a previously created instance of the quotient ring will be updated to be in the category of fields.

\textbf{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.

\textbf{EXAMPLES}:

\begin{verbatim}
sage: IntegerModRing(15)
Ring of integers modulo 15
sage: IntegerModRing(7)
Ring of integers modulo 7
sage: IntegerModRing(-100)
Ring of integers modulo 100
\end{verbatim}

Note that you can also use \texttt{Integers}, which is a synonym for \texttt{IntegerModRing}.

\begin{verbatim}
sage: Integers(18)
Ring of integers modulo 18
sage: Integers() is Integers(0) is ZZ
True
\end{verbatim}

\textbf{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 \texttt{GF()}. If the user is sure that the modulus is prime and wants to avoid a primality test, (s)he can provide \texttt{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 \texttt{Fields()} during initialisation.

\textbf{EXAMPLES}:

\begin{verbatim}
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
\end{verbatim}
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 you 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:

```python
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 \(\text{CommutativeRings}()\) (the default)

OUTPUT:

The ring of integers modulo \(N\).

EXAMPLES:

First we compute with integers modulo 29.

```python
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, 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)]
```

(continues on next page)
Finite Rings, Release 10.2

[1, 15, 1, 15, 1, 15, 1, 15, 1, 15, 1, 15, 1, 15, 1]  
sage: [powb(i) for i in range(15)]  
[1, 5, 9, 13, 1, 5, 9, 13, 1, 5, 9, 13, 1, 5, 9]  
sage: a.multiplicative_order()  
2  
sage: b.multiplicative_order()  
4  
sage: TestSuite(Z16).run()  

Saving and loading:

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

Testing ideals and quotients:

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

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

cardinality()  
Return the cardinality of this ring.

EXAMPLES:

sage: Zmod(87).cardinality()  
87  

characteristic()  
EXAMPLES:

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

degree()  
Return 1.

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

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

Return an algebraic extension of self. See \texttt{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 \texttt{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 \texttt{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:**

```python
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:

```python
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:

```python
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:

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

**is_integral_domain**(proof=None)

Return True if and only if the order of **self** is prime.
EXAMPLES:

```python
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'>
```

(continues on next page)
modulus()

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

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

EXAMPLES:

```python
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:
Finite Rings, Release 10.2

```
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()
# Needs sage.libs.pari
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()
# Needs sage.libs.pari
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 $(\mathbb{Z}/N\mathbb{Z})^\ast$.

**EXAMPLES:**

```
sage: # 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
```

(continues on next page)
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:**

```python
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:

```python
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:**

```python
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, 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:

```python
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
```

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

```python
sage: A = Zmod(55)
sage: A.unit_gens(algorithm='sage')
needs sage.groups
(12, 46)
sage: A.unit_gens(algorithm='pari')
needs sage.groups sage.libs.pari
(2, 21)
```

`unit_group(algorithm='sage')`

Return the unit group of self.

INPUT:

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

OUTPUT:

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

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

EXAMPLES:

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

```python
sage: # needs sage.groups
sage: A = Zmod(15)
sage: G = A.unit_group(); G
Multiplicative Abelian group isomorphic to C2 x C4
```
Here are two examples where the cyclic factors are isomorphic, but are ordered differently and have different generators:

```plaintext
sage: # needs sage.groups
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
# needs sage.libs.pari
Multiplicative Abelian group isomorphic to C4 x C2 x C2
sage: H.gens_values()
# needs sage.libs.pari
(17, 31, 21)
```

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

```plaintext
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 C140 x C2
sage: A = Zmod(30.factorial())
sage: A.unit_group()  # needs sage.groups
Multiplicative Abelian group isomorphic to
```
unit_group_exponent()

EXAMPLES:

```python
sage: R = IntegerModRing(17)
sage: R.unit_group_exponent()                      # needs sage.groups
16
sage: R = IntegerModRing(18)
sage: R.unit_group_exponent()                      # needs sage.groups
6
```

unit_group_order()

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

EXAMPLES:

```python
sage: R = Integers(500)
sage: R.unit_group_order()                        # needs sage.groups
200
```

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

INPUT:

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

EXAMPLES:

```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`.
**Finite Rings, Release 10.2**

- `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

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

Bases: `IntegerMod_hom`

**EXAMPLES:**

We make sure it works for every type.

```python
sage: from sage.rings.finite_rings.integer_mod import Int_to_IntegerMod
sage: Rs = [Integers(2**k) for k in range(1,50,10)]
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_int'>, 
 <class 'sage.rings.finite_rings.integer_mod.IntegerMod_int64'>, 
 <class 'sage.rings.finite_rings.integer_mod.IntegerMod_gmp'>, 
 <class 'sage.rings.finite_rings.integer_mod.IntegerMod_gmp'>]
sage: fs = [Int_to_IntegerMod(R) for R in Rs]
sage: [f(-1) for f in fs]
[1, 2047, 2097151, 2147483647, 2199023255551]
```

`sage.rings.finite_rings.integer_mod.IntegerMod(parent, value)`

Create an integer modulo $n$ with the given parent.

This is mainly for internal use.

**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:** 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 `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:**

```python
sage: k = GF(3)
sage: a = k.gen()
sage: a.charpoly('x')
x + 2
sage: a + 2
0
```

**AUTHORS:**

- Craig Citro

**crt**(other)

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

**EXAMPLES:**

```python
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
```
AUTHORS:

- Robert Bradshaw

divides(other)

Test whether self divides other.

EXAMPLES:

```python
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
```

generalised_log()

Return integers \([n_1, \ldots, n_d]\) such that

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

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

EXAMPLES:

```python
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
```

See also:

The method \(\log()\).

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

is_nilpotent()

Return True if self is nilpotent, i.e., some power of self is zero.

EXAMPLES:
sage: a = Integers(90384098234^3)
sage: factor(a.order())  # needs sage.libs.pari
2^3 * 191^3 * 236607587^3
sage: b = a(2*191)
sage: b.is_nilpotent()
False
sage: b = a(2*191*236607587)
sage: b.is_nilpotent()
True

ALGORITHM: Let \( m \geq \log_2(n) \), where \( n \) is the modulus. Then \( x \in \mathbb{Z}/n\mathbb{Z} \) is nilpotent if and only if \( x^m = 0 \).

PROOF: This is clear if you reduce to the prime power case, which you can do via the Chinese Remainder Theorem.

We could alternatively factor \( n \) and check to see if the prime divisors of \( n \) all divide \( x \). This is asymptotically slower :-).

is_one()

is_primitive_root()

Determines whether this element generates the group of units modulo \( n \).

This is only possible if the group of units is cyclic, which occurs if \( n \) is 2, 4, a power of an odd prime or twice a power of an odd prime.

EXAMPLES:

```plaintext
sage: mod(1, 2).is_primitive_root()
True
sage: mod(3, 4).is_primitive_root()
True
sage: mod(2, 7).is_primitive_root()
False
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:

```plaintext
sage: Mod(3, 17).is_square()
False
sage: Mod(9, 17).is_square()
True
sage: Mod(9, 17*19^2).is_square()  # needs sage.libs.pari
True
sage: Mod(-1, 17^30).is_square()
```

(continues on next page)
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 \(\text{ord}_p(\text{self})\) must be even or greater than \(k\).

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

AUTHORS:
- Robert Bradshaw

is_unit()

```
lift_centered()
```

Lift self to a centered congruent integer.

OUTPUT:

The unique integer \(i\) such that \(-n/2 < i \leq n/2\) and \(i = \text{self mod } n\) (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()
-49
sage: Mod(-1,3^100).lift_centered()
-1
```

log\((b=None)\)

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 \texttt{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: x^a
4294967356
```

An example with a highly composite modulus:

```python
sage: m = 2^99 * 77^7 * 123456789 * 13712923537615486607^2
sage: (Mod(5,m)^5735816763073854953388147237921).log(5)
#...
```

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

```python
sage: Mod(3, 7).log(Mod(2, 7))
#...
Traceback (most recent call last):
...
ValueError: no logarithm of 3 found to base 2 modulo 7
sage: a = Mod(16, 100); b = Mod(4, 100)
sage: a.log(b)
Traceback (most recent call last):
...
ValueError: logarithm of 16 is not defined since it is not a unit modulo 100
```

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:

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

minpoly($var=x$)

Returns the minimal polynomial of this element.

EXAMPLES:

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

modulus()

EXAMPLES:

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

multiplicative_order()

Returns the multiplicative order of self.

EXAMPLES:

```python
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
```

norm()

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

EXAMPLES:

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

AUTHORS:

- Craig Citro
\textbf{nth\_root}(n, \text{extend=False, all=False, algorithm=None, cunningham=False})

Returns an \(n\)th root of self.

\begin{itemize}
  \item n - integer \(\geq 1\)
  \item extend - bool (default: True); if True, return an \(n\)th root in an extension ring, if necessary. Otherwise, raise a \texttt{ValueError} if the root is not in the base ring. Warning: this option is not implemented!
  \item all - bool (default: False); if True, return all \(n\)th roots of self, instead of just one.
  \item 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.
  \item 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 \texttt{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 \texttt{sage -i cunningham\_tables}
\end{itemize}

\textbf{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 \texttt{ValueError} (if extend is False) or a \texttt{NotImplementedError} (if extend is True).

\begin{tabular}{|l|}
  \hline
  \textbf{Warning:} The ‘extend’ option is not implemented (yet).
  \hline
\end{tabular}

\textbf{NOTE:}

\begin{itemize}
  \item If \(n = 0\):
    \begin{itemize}
      \item if all=True:
        \begin{itemize}
          \item if self=1: all nonzero elements of the parent are returned in a list. Note that this could be very expensive for large parents.
          \item otherwise: an empty list is returned
        \end{itemize}
      \item otherwise: an empty list is returned
    \end{itemize}
  \item If \(n < 0\):
    \begin{itemize}
      \item if self is invertible, the \((-n)\)th root of the inverse of self is returned
      \item otherwise a \texttt{ValueError} is raised or empty list returned.
    \end{itemize}
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
 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
\end{verbatim}

(continues on next page)
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:

sage: k = GF(7)
sage: a = k.gen(); a
1
sage: a.polynomial()
1
sage: type(a.polynomial())
<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

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
```

```
def 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)
```

(continues on next page)
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, 126765060022829401496703205651]
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)
```

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

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 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() 14
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

(continues on next page)
We compute all square roots in several cases:

```python
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]
```

```python
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, 193097, 213348, 246303, 294637, 327592]

sage: [x^2 for x in v]
[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
```

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

sage: t = FiniteField(next_prime(2^100))(2)
sage: t.sqrt(extend=False, all=True)
[]
```

Modulo a power of 2:

```python
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]
```

```
trace()

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

EXAMPLES:
```
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 $(a*b).valuation(n)$ is not always the same as $a.valuation(n) + b. valuation(n)$

```
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 $\mathbb{Z}/n\mathbb{Z}$ 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 $\mathbb{Z}/N\mathbb{Z}$ of the ideal generated by self and other.

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

EXAMPLES:
Finite Rings, Release 10.2

```python
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:

```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
```

```python
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
```
\textbf{gcd(}\mathtt{other}\text{)}

Greatest common divisor

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

**INPUT:**

- \texttt{other} – an element of the same ring as this one.

**EXAMPLES:**

\begin{verbatim}
sage: R = Zmod(60); S = Zmod(72)
sage: a = R(40).gcd(S(30)); a
2
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
\end{verbatim}

\textbf{is_one()}

Returns \texttt{True} if this is 1, otherwise \texttt{False}.

**EXAMPLES:**

\begin{verbatim}
sage: mod(6,5).is_one()
True
sage: mod(0,5).is_one()
False
sage: mod(1, 1).is_one()
True
\end{verbatim}

\textbf{is_unit()}

Return \texttt{True} iff this element is a unit

**EXAMPLES:**

\begin{verbatim}
sage: a=Mod(23,100)
sage: a.is_unit()
True
sage: a=Mod(24,100)
sage: a.is_unit()
False
\end{verbatim}

\textbf{lift()}

Lift an integer modulo $n$ to the integers.

**EXAMPLES:**

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

(continues on next page)
Univariate Quotient Polynomial Ring in \(\sqrt{359}\) over Ring of integers modulo 360 with modulus \(x^2 + 1\)

\[
sage: y^2
\]

We compute all square roots in several cases:

\[
sage: R = 
\text{Integers}(5*2^3*3^2); 
R 
\]

Ring of integers modulo 360

\[
sage: R(40).sqrt(all=True)
\]

[20, 160, 200, 340]

\[
sage: [x \text{ for } x \text{ in } R \text{ if } x^2 == 40] \quad \# \text{ Brute force verification}
\]

[20, 160, 200, 340]

\[
sage: R(1).sqrt(all=True)
\]

[1, 19, 71, 89, 91, 109, 161, 179, 181, 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]

\[
sage: # \text{ needs sage.libs.pari}
\]

\[
sage: R = 
\text{Integers}(5*13^3*37); 
R 
\]

Ring of integers modulo 406445

\[
sage: v = R(-1).sqrt(all=True); 
v 
\]

[78853, 111808, 160142, 193097, 213348, 246303, 294637, 327592]

\[
sage: [x^2 \text{ for } x \text{ in } v]
\]

[406444, 406444, 406444, 406444, 406444, 406444, 406444, 406444]

\[
sage: v = R(169).sqrt(all=True); 
\text{min}(v), \text{-max}(v), \text{len}(v)
\]

(13, 13, 104)

\[
sage: all(x^2 == 169 \text{ for } x \text{ in } v)
\]

True

Modulo a power of 2:

\[
sage: R = 
\text{Integers}(2^7); 
R 
\]

Ring of integers modulo 128

\[
sage: a = R(17)
\]

\[
sage: a.sqrt()
\]

23

\[
sage: a.sqrt(all=True)
\]

[23, 41, 87, 105]

\[
sage: [x \text{ for } x \text{ in } R \text{ if } x^2==17]
\]

[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:**

\[
sage: a = \text{Mod}(10,3^{10}); 
a
\]

10

---

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

(continues on next page)
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(0)) 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'>,
    <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, ... → 10460353202]

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

1.2. Elements of \(\mathbb{Z}/n\mathbb{Z}\)
class sage.rings.finite_rings.integer_mod.Integer_to_IntegerMod

Bases: IntegerMod_hom

Fast \mathbb{Z} \rightarrow \mathbb{Z}/n\mathbb{Z} morphism.

EXAMPLES:

We make sure it works for every type.

```python
sage: from sage.rings.finite_rings.integer_mod import Integer_to_IntegerMod
sage: Rs = [Integers(10), Integers(10^5), Integers(10^10)]
[<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:

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

is_surjective()  
Return whether this morphism is surjective.

EXAMPLES:

```python
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:

```python
sage: x = Mod(12345678, 32098203845329048)
sage: x
12345678
sage: x^100
1017322209155072
```

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 \( n \):
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:

```python
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:

```python
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:

```python
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($k, P, Q=1, n=None$)
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:

```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:

```sage
tax = Mod(12345678, 32098203845329048)
tax
12345678
tax^100
1017322209155072
```

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
tax = randint(-100, 100)
tax
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: \( \sqrt{a} = \zeta_i a \) where \( \zeta = (2a)^{(p-5)/8} \), \( i = \sqrt{-1} \).
- \( p \) mod 16 = 9: Similar, work in a bi-quadratic extension of \( \mathbb{F}_p \) for small \( p \), Tonelli and Shanks for large \( p \).
- \( p \) mod 16 = 1: Tonelli and Shanks.

REFERENCES:

- [Mul2004]
- [Atk1992]
- [Pos1988]

AUTHORS:

- Robert Bradshaw

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 \( \sqrt{u} \) mod \( p \) via Newton’s method whenever \( p \) is odd and by a variant of Hensel lifting for \( p = 2 \).

AUTHORS:

- Robert Bradshaw
• Lorenz Panny (2022): polynomial-time algorithm for $p = 2$

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 case 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 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
sage: k = GF(9, 'a')
sage: 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
sage: k = GF(3); type(k)
<class 'sage.rings.finite_rings.finite_field_prime_modn.FiniteField_prime_modn_with_category'>
sage: k.base_ring()
Finite Field of size 3
```
sage: # needs sage.libs.linbox
sage: k = GF(9, 'alpha'); type(k)
<class 'sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro_with_category'>
sage: 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)

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).
 Finite Fields

\begin{itemize}
\item 'pari' or 'pari_ffelt' – PARI's FFELT type (only for extension fields).
\item `elem_cache` – (default: order < 500) cache all elements to avoid creation time; ignored unless `impl='givaro'`
\item `repr` – (default: 'poly') ignored unless `impl='givaro'`; controls the way elements are printed to the user:
  \begin{itemize}
  \item 'log': repr is `log_repr()`
  \item 'int': repr is `int_repr()`
  \item 'poly': repr is `poly_repr()`
  \end{itemize}
\item `check_irreducible` – verify that the polynomial modulus is irreducible
\item `proof` – bool (default: True): if True, use provable primality test; otherwise only use pseudoprimitivity test.
\end{itemize}

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

EXAMPLES:

\begin{Verbatim}
\texttt{sage: k.<a> = FiniteField(9); k}
\texttt{Finite Field in a of size 3^2}
\texttt{sage: parent(a)}
\texttt{Finite Field in a of size 3^2}
\texttt{sage: charpoly(a, 'y')}
\texttt{y^2 + 2*y + 2}
\end{Verbatim}

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

\begin{Verbatim}
\texttt{sage: k = FiniteField(10^1000 + 453, proof=False)}
\texttt{sage: k = FiniteField((10^1000 + 453)^2, 'a', proof=False)}
\texttt{# long time --~ \rightarrow about 5 seconds}
\end{Verbatim}

\begin{Verbatim}
\texttt{sage: F.<x> = GF(5)}
\texttt{sage: K.<a> = GF(5^45, name='a', modulus=x^5 - x +1 )}
\texttt{sage: f = K.modulus(); f}
\texttt{x^5 + 4*x + 1}
\texttt{sage: type(f)}
\texttt{<class 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>}
\end{Verbatim}

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

\begin{Verbatim}
\texttt{sage: K.<a> = GF(5^45)}
\texttt{sage: a=multiplicative_order()}
\texttt{7105427357601001858711242675781}
\texttt{sage: a.is_square()}
\texttt{True}
\texttt{sage: K.<b> = GF(5^45, modulus="primitive")}
\end{Verbatim}
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 $\mathbb{GF}(p)[x]$ 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))
  ... 
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')
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:
```
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...
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:
Moreover, repr and elem_cache are ignored when not using givaro:

```python
sage: GF.create_key_and_extra_args(16, 'a', impl='ntl', repr='poly') # needs sage.libs.ntl
((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.nntl
((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.nntl
True
```

We handle extra arguments for the givaro finite field and create unique objects for their defaults:

```python
sage: GF(25, impl='givaro') is GF(25, impl='givaro', repr='poly') # needs sage.libs.linbox
True
sage: GF(25, impl='givaro') is GF(25, impl='givaro', elem_cache=True) # needs sage.libs.linbox
True
sage: GF(625, impl='givaro') is GF(625, impl='givaro', elem_cache=False) # needs sage.libs.linbox
True
```

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:

```python
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, True), {})
```

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

**EXAMPLES:**

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

We try to create finite fields with various implementations:

```python
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.nntl
sage: k = GF(2, impl='pari')
```

(continues on next page)
... Value Error: the degree must be at least 2
sage: k = GF(2, impl='supercalifragilisticexpialidocious')
Traceback (most recent call last):
...
Value Error: no such finite field implementation:
   'supercalifragilisticexpialidocious'
sage: k.<a> = GF(2^15, impl='modn')
Traceback (most recent call last):
...
Value Error: 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.libsntl
sage: k.<a> = GF(2^15, impl='pari')
sage: k.<a> = GF(3^60, impl='modn')
Traceback (most recent call last):
...
Value Error: the 'modn' implementation requires a prime order
sage: k.<a> = GF(3^60, impl='givaro') # needs sage.libs.linbox
Traceback (most recent call last):
...
Value Error: q must be < 2^16
sage: k.<a> = GF(3^60, impl='ntl') # needs sage.libsntl
Traceback (most recent call last):
...
Value Error: 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)
Traceback (most recent call last):
  ... DeprecationWarning: the function is_PrimeFiniteField is depreciate;
   use isinstance(x, sage.rings.finite_rings.finite_field_base.FiniteField) and x.is_  
   prime_field() instead
See https://github.com/sagemath/sage/issues/32664 for details.
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 self. Note that mathematically speaking, this is only unique up to non-unique
isomorphism. To obtain canonically defined algebraic closures, one needs an algorithm that also provides
a canonical isomorphism between any two algebraic closures constructed using the algorithm.

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

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

EXAMPLES:

```
sage: F = GF(5).algebraic_closure()
sage: F
Algebraic closure of Finite Field of size 5
sage: F.gen(3)
Z3
```

The default name is ‘z’ but you can change it through the option name:

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

Because Sage currently only implements algebraic closures using a non-unique definition (see above), it is
currently impossible to implement pickling in such a way that a pickled and unpickled element compares
equal to the original:
sage: F = GF(7).algebraic_closure()
sage: x = F.gen(2)
sage: loads(dumps(x)) == x
False

Note: This is currently only implemented for prime fields.

cardinality()
Return the cardinality of self.
Same as order().
EXAMPLES:

sage: GF(997).cardinality()
997

collection()
Return the construction of this finite field, as a ConstructionFunctor and the base field.
EXAMPLES:

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

The implementation is taken into account, by github issue #15223:

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

**EXAMPLES:**

```sage
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: $\text{Tr}(e_id_j) = \delta_{i,j}, 0 \leq i,j \leq n-1$.

```sage
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 + 2*a + 2, 2*a^3 + 2*a^2 + 2]
sage: d = F.dual_basis(e, check=True); d
[3*a^3 + 4*a^2 + 6*a + 2, 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
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 + a^4 + 2*a^3 + 5*a^2 + 4*a + 2,
  4*a^6 + 5*a^5 + 5*a^4 + 4*a^3 + 5*a^2 + 3 + a + 6,
  5*a^7 + a^6 + 2*a^5 + 5*a^4 + 4*a^3 + 5*a^2 + 2*a + 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 + 4*a^3 + 5*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
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
sage: F.dual_basis([a^0, a, a^0 + a], check=True)  # needs sage.modules
Traceback (most recent call last):
```

(continues on next page)
... ValueError: value of 'basis' keyword is not a basis

AUTHOR:
• Thomas Gagne (2015-06-16)

extension(modulus, 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:

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

An example using the map argument:

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

(continues on next page)
from Finite Field of size 5
to Finite Field in b of size 5^2

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

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

**factored_order()**

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

**EXAMPLES:**

```python
sage: 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:**

```python
sage: 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:**
Vector space of dimension 3 over Finite Field of size 3

Vector space of dimension 2 over Finite Field in z3 of size 2^3

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

**frobenius_endomorphism**(n=1)

**INPUT:**

- n – an integer (default: 1)
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()  # needs sage.modules
sage: Frob(a) == a^3  # needs sage.modules
True
```

We can specify a power:

```python
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:

```python
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:

```python
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, 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)
```

(continues on next page)
\[100a^3 + 37a^2 + 12a + 6\]
sage: F.from_integer(n) in F
True
sage: F.from_integer(n, reverse=True)
6a^3 + 12a^2 + 37a + 100

**galois_group()**

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

**EXAMPLES:**

```python
sage: # needs sage.groups
sage: G = GF(3^6).galois_group(); G
Galois group C6 of GF(3^6)
sage: F = G.gen()
sage: F^2
Frob^2
sage: F^6
1
```

**gen()**

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

**EXAMPLES:**

```python
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:**

```python
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:**

```python
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: GF(2).is_perfect()
True
```
For prime fields, this returns $x - 1$ unless a custom modulus was given when constructing this field:

```python
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:**
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 \(F_p\) in the variable name.

See also:
Except for the name argument, this is identical to the \(modulus()\) method.

EXAMPLES:

```python
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.libs.ntl
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^8 + 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:

```python
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:**

```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
```

**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
# needs sage.modules
True
```

Passes extra positional or keyword arguments through:

```python
sage: k.random_element(prob=0)
# needs sage.modules
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
# needs sage.modules
[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:**
subfields \( \text{(degree=0, name=None)} \)

Return all subfields of \( \text{self} \) of the given degree, or all possible degrees if degree is 0.

The subfields are returned as absolute fields together with an embedding into \( \text{self} \).

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:

```python
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')
sage: k.order()  
1024
sage: 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)
sage: k.zeta()  
3
sage: k.zeta().multiplicative_order()  
6
sage: k.zeta(3)  
2
sage: k.zeta(3).multiplicative_order()  
3
sage: k = GF(49, 'a')
sage: k.zeta().multiplicative_order()  
48
sage: k.zeta(6)  
3
sage: k.zeta(5)
Traceback (most recent call last):  
... IndexError: 5 is not invertible in Finite Field in 5 over Finite Field in 7 of size 2^2

(continues on next page)
ValueError: no 5th root of unity in Finite Field in a of size 7^2

Even more examples:

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

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

```python
sage: k.<a> = GF(2^2000)
sage: p = 887794514874294500114604143902514703409869050359101317733635669441651752731018138001
sage: z = k.zeta(p)
sage: z
a^1999 + a^1996 + a^1995 + a^1994 + ... + a^7 + a^5 + a^4 + 1
sage: z ^ p
1
```

**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
sage: is_FiniteField(GF(9, 'a'))
```

Note that the integers modulo \( n \) are not backed by the finite field type:
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:

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

  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:**

```
sage: from sage.rings.finite_rings.element_base import FinitePolyExtElement
sage: k.<a> = FiniteField(19^2)
sage: parent(a)
Finite Field in a of size 19^2
sage: b = a**20
sage: p = FinitePolyExtElement.charpoly(b, "x", algorithm="pari")
```

```
# needs sage.modules
```

```
sage: q = FinitePolyExtElement.charpoly(b, "x", algorithm="matrix")  # needs sage.modules
```

```
q == p
# needs sage.modules
```

```
True
```

```
sage: p
x^2 + 15*x + 4
sage: factor(p)
(x + 17)^2
sage: b.minpoly('x')
x + 17
```

**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 morphism.

**EXAMPLES:**

```
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:**

```python
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)
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
```

**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:**

```python
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.libsntl
sage: (a**2).is_square()
  # needs sage.libsntl
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:

```python
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:

```python
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

EXAMPLES:

```python
sage: # needs sage.modules
sage: k.<a> = GF(2^4)
sage: b = k.random_element()
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='y')**

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

EXAMPLES:

```python
sage: k.<a> = FiniteField(3^4)
sage: parent(a)
Finite Field in a of size 3^4
sage: b=a^20;p=charpoly(b,'y');p
y^4 + 2*y^2 + 1
sage: factor(p)
(y^2 + 1)^2
sage: b.minimal_polynomial('y')
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
gsage: k.<a> = FiniteField(19^2)
sage: parent(a)
Finite Field in a of size 19^2
gsage: b=a**20
gsage: p=FinitePolyExtElement.minpoly(b,"x", algorithm="pari")
gsage: q=FinitePolyExtElement.minpoly(b,"x", algorithm="matrix")
gsage: q == p
True
gsage: p
x + 17
```

**multiplicative_order()**

Return the multiplicative order of this field element.

**EXAMPLES:**

```python
sage: S.<a> = GF(5^3); S
Finite Field in a of size 5^3
gsage: a.multiplicative_order()
124
gsage: (a^8).multiplicative_order()
31
gsage: 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:**

```python
sage: S.<b> = GF(5^2); S
Finite Field in b of size 5^2
gsage: b.norm()
2
gsage: 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 \(\geq 1\)
- \(\text{extend}\) -- 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!
- \(\text{all}\) -- bool (default: False); if True, return all \(n\)th roots of self, instead of just one.
- \(\text{algorithm}\) -- 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 NotImplementedError (if extend is True).

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

EXAMPLES:

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: K.<a> = GF(625)
sage: (3*a^2+a+1).nth_root(13)**13
3*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
\texttt{sage}: b.nth\_root\(5\), all = \texttt{True})

[]
\texttt{sage}: b.nth\_root(3, all = \texttt{True})

[14*a + 18, 10*a + 13, 5*a + 27]
\texttt{sage}: k.\langle a\rangle = GF(29^5)
\texttt{sage}: b = a^2 + 5*a + 1
\texttt{sage}: b.nth\_root(5)

19*a^4 + 2*a^3 + 2*a^2 + 16*a + 3
\texttt{sage}: b.nth\_root(7)

Traceback (most recent call last):
...
ValueError: no nth root
\texttt{sage}: b.nth\_root(4, all=\texttt{True})

[]

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 C int type)

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

EXAMPLES:

\texttt{sage}: F.\langle a\rangle = GF(29^2)
\texttt{sage}: z = a^2 + 5*a + 1
\texttt{sage}: z.pth\_power()

19*a + 20
\texttt{sage}: z.pth\_power(10)

10*a + 28
\texttt{sage}: z.pth\_power(-10) == z

True
\texttt{sage}: F.\langle b\rangle = GF(2^12)
\texttt{sage}: y = b^3 + b + 1
\texttt{sage}: y == (y.pth\_power(-3))^2^3

True
\texttt{sage}: y.pth\_power(2)

b^7 + b^6 + b^5 + b^4 + b^3 + b

\texttt{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:**

```python
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:**

```python
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^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.

  ```python
  Warning: This option is not implemented!
  ```

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

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

**EXAMPLES:**

```python
sage: F = FiniteField(7^2, 'a')
sage: F.(a).square_root() 4
sage: F.(3).square_root() 2*a + 6
sage: F.(3).square_root()**2 3
sage: F.(4).square_root() 2
sage: K = FiniteField(7^3, 'alpha', impl='pari_ffelt')
sage: K.(3).square_root()
Traceback (most recent call last):
  ...
ValueError: must be a perfect square.
```
**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 reverse is set to True (default: False), the list of coefficients is reversed prior to evaluation.

Inverse of `sage.rings.finite_rings.finite_field_base.FiniteField.from_integer()`.

**EXAMPLES:**

```python
sage: F.<t> = GF(7^5)
sage: F(5).to_integer() 5
sage: t.to_integer() 7
sage: (t^2).to_integer() 49
sage: (t^2+1).to_integer() 50
sage: (t^2+t+1).to_integer() 57
```

```python
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:**

```python
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`

`sage.rings.finite_rings.element_base.is_FiniteFieldElement(x)`

Return True if \( x \) is a finite field element.
This function is deprecated.

EXAMPLES:

```python
sage: from sage.rings.finite_rings.element_base import is_FiniteFieldElement
sage: is_FiniteFieldElement(1)
Doctest:...: DeprecationWarning: the function is_FiniteFieldElement is deprecated;
→ use isinstance(x, sage.structure.element.FieldElement) and x.parent().is_finite() instead
See https://github.com/sagemath/sage/issues/32664 for details.
False
sage: is_FiniteFieldElement(IntegerRing())
False
sage: is_FiniteFieldElement(GF(5)(2))
True
```

2.4 Homset for finite fields

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

EXAMPLES:

```python
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:

```python
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
```

```
```
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^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
```

(continues on next page)
Defn: 1 |--> 1
]
sage: Hom(k2, k1).list()
[
Ring morphism:
    From: Finite Field of size 1009
    To:   Finite Field of size 1009
    Defn: 11 |--> 11
]

sage: k1.<a> = GF(1009^2, modulus="first_lexicographic")
sage: k2.<b> = GF(1009^2, modulus="conway")
sage: Hom(k1, k2).list()
[
Ring morphism:
    From: Finite Field in a of size 1009^2
    To:   Finite Field in b of size 1009^2
    Defn: a |--> 290*b + 864,
Ring morphism:
    From: Finite Field in a of size 1009^2
    To:   Finite Field in b of size 1009^2
    Defn: a |--> 719*b + 145
]

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
Ring morphism:
  From: Finite Field in t of size 3^7
  To:   Finite Field in T of size 3^21
  Defn: t |--> T^20 + 2*T^18 + T^16 + 2*T^13 + T^9 + 2*T^8 + T^7 + T^6 + T^5 + T^3 + 2*T^2 + T
sage: f(t)
T^20 + 2*T^18 + T^16 + 2*T^13 + T^9 + 2*T^8 + T^7 + T^6 + T^5 + T^3 + 2*T^2 + T

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

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

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

sage: k.<t> = GF(5^6)
sage: K.<T> = GF(5^{11})
sage: FiniteFieldHomomorphism_generic(Hom(k, K))
Traceback (most recent call last):
  ... ValueError: No embedding of Finite Field in t of size 5^6 into Finite Field in T of size 5^{11}

Construction of Frobenius endomorphisms:

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

Some basic arithmetics is supported:

sage: Frob^2
Frobenius endomorphism t |---> t^{7*2} on Finite Field in t of size 7^{14}

(continues on next page)
```python
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:

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

And that comparisons work:

```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
        sage: from sage.rings.finite_rings.hom_finite_field import _
        →FiniteFieldHomomorphism_generic
        sage: k.<t> = GF(3^3)
        sage: K.<T> = GF(3^9)
        sage: f = FiniteFieldHomomorphism_generic(Hom(k, K))
        sage: f.is_injective()
        True
        ```
    is_surjective()
        Return True if this embedding is surjective (and hence an isomorphism.
        EXAMPLES:
        ```python
        sage: from sage.rings.finite_rings.hom_finite_field import _
        →FiniteFieldHomomorphism_generic
        sage: k.<t> = GF(3^3)
        sage: K.<T> = GF(3^9)
        ```
```

(continues on next page)
sage: f = FiniteFieldHomomorphism_generic(Hom(k, K))
sage: f.is_surjective()
False
sage: g = FiniteFieldHomomorphism_generic(Hom(k, k))
sage: g.is_surjective()
True

section()

Return the inverse of this embedding.

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

EXAMPLES:

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

class sage.rings.finite_rings.homFiniteField.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:
Finite Rings, Release 10.2

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

**inverse()**

Return the inverse of this Frobenius endomorphism.

**EXAMPLES:**

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

**is_identity()**

Return True if this morphism is the identity morphism.

**EXAMPLES:**

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

**is_injective()**

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

**EXAMPLES:**

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

**is_surjective()**

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

**EXAMPLES:**

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

(continues on next page)
order()

Return the order of this endomorphism.

EXAMPLES:

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

power()

Return an integer $n$ such that this endomorphism is the $n$-th power of the absolute (arithmetric) 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.

2.5. Finite field morphisms 81
3.1 Finite prime fields

AUTHORS:

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

```python
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()
5
```

def polynomial(name=None)

Returns the polynomial name.

EXAMPLES:

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

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 :-).

def 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:
```plaintext
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:

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

(continues on next page)
Finite Rings, Release 10.2

(continued from previous page)

```python
sage: a.parent()
Finite Field in a of size 3^2
sage: a.charpoly('x')
x^2 + 2*x + 2
sage: [a^i for i in range(8)]
[1, a, a + 1, 2*a + 1, 2, 2*a, 2*a + 2, a + 2]

Next we compute with a finite field of order 16:

```python
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:

```python
sage: K = FiniteField(7^10, 'b', impl='pari_ffelt')
sage: loads(K.dumps()) == K
True
sage: K = FiniteField(10007^10, 'a', impl='pari_ffelt')
sage: loads(K.dumps()) == K
True
```

**Element**

alias of `FiniteFieldElement_pari_ffelt`

**characteristic()**

Return the characteristic of `self`.

EXAMPLES:

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

**degree()**

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

EXAMPLES:

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

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

INPUT:

\begin{itemize}
\item \( n \) – must be 0
\end{itemize}

OUTPUT:

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

\begin{center}
\textbf{Warning: } This generator is not guaranteed to be a generator for the multiplicative group. To obtain the latter, use \texttt{multiplicative_generator()} or use the \texttt{modulus=\textquoteleft primitive\textquoteleft} option when constructing the field.
\end{center}

EXAMPLES:

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

\section{4.2 Finite field elements implemented via PARI\textquoteleft s FFELT type}

AUTHORS:

\begin{itemize}
\item Peter Bruin (June 2013): initial version, based on \texttt{element\_ext\_pari.py} by William Stein et al. and \texttt{element\_ntl\_gf2e.pyx} by Martin Albrecht.
\end{itemize}

\begin{verbatim}
class sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt
    Bases: FinitePolyExtElement

    An element of a finite field implemented using PARI.

    EXAMPLES:

    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'>
\end{verbatim}

\texttt{charpoly}(\texttt{var='x'}

Return the characteristic polynomial of \texttt{self}.

INPUT:

\begin{itemize}
\item \texttt{var} – string (default: \textquoteleft x\textquoteleft): variable name to use.
\end{itemize}

EXAMPLES:
Finite Rings, Release 10.2

```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)

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

INPUT:

• base – non-zero field element

OUTPUT:

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

EXAMPLES:

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

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

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

```
sage: F.<a> = GF(3^10, impl='pari_ffelt')
sage: a.log(-1)
Traceback (most recent call last):
...
```

ArithmeticError: element a does not lie in group generated by 2
sage: a.log(0)
Traceback (most recent call last):
...
ArithmeticError: discrete logarithm with base 0 is not defined
sage: F(0).log(1)
Traceback (most recent call last):
...
ArithmeticError: discrete logarithm of 0 is not defined

```
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:
```
sage: k.<a> = FiniteField(3^2, impl='pari_ffelt')
sage: pol = a.polynomial()
sage: pol
a
sage: parent(pol)
Univariate Polynomial Ring in a over Finite Field of size 3
```
```
sage: k = FiniteField(3^4, 'alpha', impl='pari_ffelt')
sage: a = k.gen()
sage: a.polynomial()
```
Finite Rings, Release 10.2

(continued from previous page)

alpha

\begin{verbatim}
sage: (a**2 + 1).polynomial('beta')
beta^2 + 1
sage: (a**2 + 1).polynomial().parent()
Univariate Polynomial Ring in alpha over Finite Field of size 3
sage: (a**2 + 1).polynomial('beta').parent()
Univariate Polynomial Ring in beta over Finite Field of size 3
\end{verbatim}

\section{pth_power ($k=1$)}

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

\begin{itemize}
\item \texttt{k} – integer (default: 1); must fit in a C \texttt{int}
\end{itemize}

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

\section{sqrt ($extend=False$, $all=False$)}

Return a square root of \texttt{self}, if it exists.

\begin{itemize}
\item \texttt{extend} – bool (default: False)
\end{itemize}

\textbf{Warning:} This option is not implemented.

\begin{itemize}
\item \texttt{all} - bool (default: False)
\end{itemize}

\textbf{Warning:} The \texttt{extend} option is not implemented (yet).

\section{EXAMPLES:}

\begin{verbatim}
sage: F = FiniteField(7^2, 'a', impl='pari_ffelt')
sage: F(2).sqrt()
4
sage: F(3).sqrt() in (2^5*F.gen() + 6, 5^5*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()  # can't square root in the base field
Traceback (most recent call last):
\end{verbatim}

(continues on next page)
... 
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
```
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
sage: f = k.modulus(); f
x^8 + x^4 + x^3 + x^2 + 1
```
You may enforce a modulus:

```python
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:

```python
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:

```python
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:

```python
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 $a \times b - c$.

**INPUT:**

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

**EXAMPLES:**

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

**INPUT:**

- $a, b, c$ – `FiniteField_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:**

- \( a, b, c \) – *FiniteField_givaroElement*

**EXAMPLES:**

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

**characteristic()**

Return the characteristic of this field.

**EXAMPLES:**

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

**degree()**

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

**OUTPUT:**

Integer – the degree

**EXAMPLES:**

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

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

Deprecated: Use `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:**

```sage
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:
\textbf{sage: }k\text{.frobenius\_endomorphism}(2)
\begin{verbatim}
Frobenius endomorphism t |--> t^{3^2} on Finite Field in t of size 3^5
\end{verbatim}

The result is simplified if possible:
\textbf{sage: }k\text{.frobenius\_endomorphism}(6)
\begin{verbatim}
Frobenius endomorphism t |--> t^3 on Finite Field in t of size 3^5
\end{verbatim}
\textbf{sage: }k\text{.frobenius\_endomorphism}(5)
\begin{verbatim}
Identity endomorphism of Finite Field in t of size 3^5
\end{verbatim}

Comparisons work:
\textbf{sage: }k\text{.frobenius\_endomorphism}(6) == \text{Frob}
\begin{verbatim}
True
\end{verbatim}
\textbf{sage: }from\ sage\\ categories\.morphism\ import\ IdentityMorphism\n\textbf{sage: }k\text{.frobenius\_endomorphism}(5) == \text{IdentityMorphism}(k)
\begin{verbatim}
True
\end{verbatim}

\textbf{AUTHOR:}
\begin{itemize}
  \item Xavier Caruso (2012-06-29)
\end{itemize}

\textbf{\texttt{from\_integer}}\texttt{(}n\texttt{)}

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

\textbf{EXAMPLES:}
\textbf{sage: }k.<a> = GF(2^8)
\textbf{sage: }k\text{.from\_integer}(8)
\begin{verbatim}
a^3
\end{verbatim}
\textbf{sage: }e = k\text{.from\_integer}(151); e
\begin{verbatim}
a^7 + a^4 + a^2 + a + 1
\end{verbatim}
\textbf{sage: }2^7 + 2^4 + 2^2 + 2 + 1
\begin{verbatim}
151
\end{verbatim}

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

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

\textbf{INPUT:}
\begin{itemize}
  \item \(n\) – must be 0
\end{itemize}

\textbf{OUTPUT:}
An element \(a\) of \texttt{self} such that \texttt{self.modulus()}(\(a\)) == 0.

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

\textbf{EXAMPLES:}
\textbf{sage: }k = GF(3^4, 'b'); k\text{.gen()}
\begin{verbatim}
b
\end{verbatim}
sage: k.gen(1)
Traceback (most recent call last):
...
IndexError: only one generator
sage: F = FiniteField(31, impl='givaro')
sage: F.gen()
1

**int_to_log**(\(n\))

Given an integer \(n\) this method returns \(i\) where \(i\) satisfies \(g^i = n \mod p\) where \(g\) is the generator and \(p\) is the characteristic of \(\text{self}\).

**INPUT:**

- \(n\) – integer representation of an 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 \(g^i = i\) 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
```
Finite Rings, Release 10.2

```
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:

```
sage: GF(3^4, 'b').prime_subfield()
Finite Field of size 3
sage: S.<b> = GF(5^2); S
Finite Field in b of size 5^2
sage: S.prime_subfield()
Finite Field of size 5
sage: type(S.prime_subfield())
<class 'sage.rings.finite_rings.finite_field_prime_modn.FiniteField_prime_modn_with_category'>
```

random_element(*args, **kwds)

Return a random element of self.

EXAMPLES:

```
sage: k = GF(23**3, 'a')
sage: e = k.random_element()
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:

```
sage: k = GF(5); type(k)
<class 'sage.rings.finite_rings.finite_field_prime_modn.FiniteField_prime_modn_with_category'>
```

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

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

You may enforce a modulus:

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

You may enforce a random modulus:

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

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

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

\texttt{a\_times\_b\_minus\_c}(a, b, c)

Return \(a \times b - c\).

INPUT:

\begin{itemize}
  \item a, b, c – \texttt{FiniteField\_givaroElement}
\end{itemize}

EXAMPLES:

\begin{verbatim}
sage: k.<a> = GF(3**3)
sage: k._cache.a_times_b_minus_c(a,a,k(1))
a^2 + 2
\end{verbatim}

\texttt{a\_times\_b\_plus\_c}(a, b, c)

Return \(a \times b + c\).

This is faster than multiplying \(a\) and \(b\) first and adding \(c\) to the result.

INPUT:

\begin{itemize}
  \item a, b, c – \texttt{FiniteField\_givaroElement}
\end{itemize}

EXAMPLES:

\begin{verbatim}
sage: k.<a> = GF(2**8)
sage: k._cache.a_times_b_plus_c(a,a,k(1))
a^2 + 1
\end{verbatim}
c_minus_a_times_b(a, b, c)
Return c - a*b.

INPUT:
  • a, b, c – FiniteField_givaroElement

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

characteristic()
Return the characteristic of this field.

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

element_from_data(e)
Coerces several data types to self.

INPUT:
  • e – data to coerce in.

EXAMPLES:
```
sage: k = GF(3^8, 'a')
sage: type(k)
<class 'sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro_with_category'>
sage: e = k.vector_space(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 F_p.

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

(continues on next page)
\[ a^7 + a^4 + a^2 + a + 1 \]

**sage:** \[ 2^7 + 2^4 + 2^2 + 2 + 1 \]

\[ 151 \]

**gen()**

Return a generator of the field.

**EXAMPLES:**

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

**int_to_log(n)**

Given an integer \( n \) this method returns \( i \) where \( i \) satisfies \( g^i = n \mod p \) where \( g \) is the generator and \( p \) is the characteristic of \( \text{self} \).

**INPUT:**

- \( n \) – integer representation of an finite field element

**OUTPUT:**

log representation of \( n \)

**EXAMPLES:**

```python
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 \( g^i = n \) 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._cache.log_to_int(4)
16
sage: k._cache.log_to_int(20)
180
```

**order()**

Return the order of this field.

**EXAMPLES:**
...
**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(3^4); k
Finite Field in a of size 3^4
sage: a.is_unit()
True
sage: k(0).is_unit()
False
```

**log(base)**

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

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

**EXAMPLES:**

```sage
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
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:**
```python
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)
sage: k(3).sqrt()
Traceback (most recent call last):
  ... ValueError: must be a perfect square.
```

**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
```

(continues on next page)

5.2. Givaro finite field elements 107
Finite Rings, Release 10.2

(continued from previous page)

\[ \begin{align*}
 & a \\
 & a + 1 \\
 & 1
\end{align*} \]

```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:**

```python
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
```

(continues on next page)
Ring morphism:
  From: Finite Field in t_fixed of size 5^2
  To:   Finite Field in t of size 5^6
  Defn: t_fixed |--> 4*t^5 + 2*t^4 + 4*t^2 + t

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

class
sage.rings.finite_rings.hom_finite_field_givaro.SectionFiniteFieldHomomorphism_givaro
  Bases: SectionFiniteFieldHomomorphism_generic
CHAPTER
SIX

FINITE FIELDS OF CHARACTERISTIC 2 USING NTL

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')
sage: k.modulus()
```

(continues on next page)
\[
x^{211} + x^{11} + x^{10} + x^{8} + 1
\]
\[
sage: k.<a> = GF(2^{211}, \text{modulus}='conway')
\]
\[
sage: k.modulus()
\]
\[
x^{211} + x^{9} + x^{6} + x^{5} + x^{3} + x + 1
\]
\[
sage: k.<a> = GF(2^{23}, \text{modulus}='conway')
\]
\[
sage: a.multiplicative_order() == k.order() - 1
True
\]

**characteristic()**

Return the characteristic of self which is 2.

**EXAMPLES:**

\[
sage: k.<a> = GF(2^{16}, \text{modulus}='random')
\]
\[
sage: k.characteristic()
2
\]

**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:**

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

**gen(n=0)**

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

**INPUT:**

- `n` – must be 0

**OUTPUT:**

An element \(a\) of self such that self.modulus()(a) == 0.
Warning: This generator is not guaranteed to be a generator for the multiplicative group. To obtain the latter, use multiplicative_generator() or use the modulus="primitive" option when constructing the field.

EXAMPLES:

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

order()

Return the cardinality of this field.

EXAMPLES:

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

prime_subfield()

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

EXAMPLES:

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

```
sage.rings.finite_rings.finite_field_ntl_gf2e.late_import()
Imports various modules after startup.
EXAMPLES:
```
sage: sage.rings.finite_rings.finite_field_ntl_gf2e.late_import()
sage: sage.rings.finite_rings.finite_field_ntl_gf2e.GF2 is None # indirect doctest
False
```

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

fetch_int(number)
    Given an integer less than $p^n$ with base 2 representation $a_0 + a_1 \cdot 2 + \cdots + a_k 2^k$, this returns $a_0 + a_1 x + \cdots + a_k x^k$, where $x$ is the generator of this finite field.
    INPUT:
    • number – an integer, of size less than the cardinality
    EXAMPLES:
    
    sage: k.<a> = GF(2^48)
sage: k._cache.fetch_int(2^33 + 2 + 1)
a^33 + a + 1

import_data(e)
    EXAMPLES:
    
    sage: k.<a> = GF(2^17)
sage: V = k.vector_space(map=False)
sage: v = [1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0]
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.
    EXAMPLES:
    
    sage: k.<a> = GF(2^64)
sage: k._cache.order()
    18446744073709551616

polynomial()
    Returns the list of 0’s and 1’s giving the defining polynomial of the field.
    EXAMPLES:
    
    sage: k.<a> = GF(2^20,modulus="minimal_weight")
sage: k._cache.polynomial()
    [1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
charpoly(var='x')

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

**INPUT:**

- var – string (default: 'x')

**OUTPUT:**

polynomial

**EXAMPLES:**

```
sage: k.<a> = GF(2^8, impl="ntl")
sage: b = a^3 + a
sage: b.minpoly()
x^4 + x^3 + x^2 + x + 1
sage: b.charpoly()
x^8 + x^6 + x^4 + x^2 + 1
sage: b.charpoly().factor()
(x^4 + x^3 + x^2 + x + 1)^2
sage: b.charpoly('Z')
Z^8 + Z^6 + Z^4 + Z^2 + 1
```

is_one()

Return True if self == k(1).

Equivalent to self != k(0).

**EXAMPLES:**

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

is_square()

Return True as every element in \( F_{2^n} \) is a square.

**EXAMPLES:**

```
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:**

6.2. Elements of finite fields of characteristic 2 115
sage: k.<a> = GF(2^17)
sage: a.is_unit()
True
sage: k(0).is_unit()
False

\textbf{log} (base)

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

\textbf{INPUT}:

\begin{itemize}
  \item base -- finite field element
\end{itemize}

\textbf{OUTPUT}:

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

\textbf{ALGORITHM}: pari:fflog

\textbf{EXAMPLES}:

\begin{sage}
    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
\end{sage}

Big instances used to take a very long time before \texttt{github issue #32842}:

\begin{sage}
    sage: g = GF(2^61).gen()
    sage: g.log(g^7)
    1976436865040309101
\end{sage}

\textbf{AUTHORS}:

\begin{itemize}
  \item David Joyner and William Stein (2005-11)
  \item Lorenz Panny (2021-11): use PARI's pari:fflog instead of \texttt{sage.groups.generic.discrete_log()}
\end{itemize}

\textbf{minpoly} (var='x')

Return the minimal polynomial of self, which is the smallest degree polynomial $f \in F_2[x]$ such that $f(self) == 0$.

\textbf{INPUT}:

\begin{itemize}
  \item var -- string (default: 'x')
\end{itemize}

\textbf{OUTPUT}:

polynomial

\textbf{EXAMPLES}:

\begin{sage}
    sage: K.<a> = GF(2^100)
    sage: f = a.minpoly(); f
    x^100 + x^109 + x^108 + x^107 + x^106 + x^105 + x^102 + x^101 + x^100 + x^99 + x^98 + x^97 + x^96 + x^95 + x^94 + x^93 + x^92 + x^91 + x^90 + x^89 + x^88 + x^87 + x^86 + x^85 + x^84 + x^83 + x^82 + x^81 + x^80 + x^79 + x^78 + x^77 + x^76 + x^75 + x^74 + x^73 + x^72 + x^71 + x^70 + x^69 + x^68 + x^67 + x^66 + x^65 + x^64 + x^63 + x^62 + x^61 + x^60 + x^59 + x^58 + x^57 + x^56 + x^55 + x^54 + x^53 + x^52 + x^51 + x^50 + x^49 + x^48 + x^47 + x^46 + x^45 + x^44 + x^43 + x^42 + x^41 + x^40 + x^39 + x^38 + x^37 + x^36 + x^35 + x^34 + x^33 + x^32 + x^31 + x^30 + x^29 + x^28 + x^27 + x^26 + x^25 + x^24 + x^23 + x^22 + x^21 + x^20 + x^19 + x^18 + x^17 + x^16 + x^15 + x^14 + x^13 + x^12 + x^11 + x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
\end{sage}
\[\overline{20} + x^{19} + x^{16} + x^{15} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^3 + 1\]

```
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:**

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

`sqt(all=False, extend=False)`

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

**EXAMPLES:**

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

This failed before github issue #4899:

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

`trace()`

Return the trace of `self`.

**EXAMPLES:**

```
sage: K.<a> = GF(2^25)
sage: a.trace()
0
```

(continues on next page)
sage: a.charpoly()
\texttt{x^25 + x^8 + x^6 + x^2 + 1}

\texttt{sage: parent(a.trace())}
Finite Field of size 2

\texttt{sage: b = a+1}
\texttt{sage: b.trace()}
1

\texttt{sage: b.charpoly()[1]}
1

\textbf{weight()}

Returns the number of non-zero coefficients in the polynomial representation of \texttt{self}.

\textbf{EXAMPLES:}

\texttt{sage: K.<a> = GF(2^21)}
\texttt{sage: a.weight()}
1

\texttt{sage: (a^5+a^2+1).weight()}
3

\texttt{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

\texttt{sage: b.weight()}
18

\texttt{sage.rings.finite_rings.element_ntl_gf2e.unpickleFiniteField_ntl_gf2eElement}(\texttt{parent, elem})

\textbf{EXAMPLES:}

\texttt{sage: k.<a> = GF(2^20)}
\texttt{sage: e = k.random_element()}
\texttt{sage: f = loads(dumps(e)) \# indirect doctest}
\texttt{sage: e == f}
\texttt{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:

```python
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):

```python
sage: # needs sage.rings.number_field
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 $\mathbb{Z}$:

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

And for polynomial rings over finite fields:

```python
sage: # needs sage.rings.finite_rings
sage: R.<t> = GF(5)[]
sage: I = R.ideal(t^2 + 2)
sage: k = ResidueField(I); k
```

(continues on next page)
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 $\mathbb{Z}Z$
- 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: sage.categories.map

    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 in Number Field in a with defining polynomial x^3 + 2
    sage: L(F.0^2)
    3*a + 1
    sage: L(3*a + 1) == F.0^2
    True

    sage: # needs sage.rings.finite_rings
    sage: R.<t> = GF(13)[]
    sage: P = R.ideal(8*t^12 + 9*t^11 + 11*t^10 + 2*t^9 + 11*t^8
    ....:     + 3*t^7 + 12*t^6 + t^4 + 7*t^3 + 5*t^2 + 12*t + 1)
    sage: k.<a> = P.residue_field()
    sage: k.lift_map()
    Lifting map:
        From: Residue field in a of Principal ideal (t^12 + 6*t^11 + 3*t^10
        + 10*t^9 + 3*t^8 + 2*t^7 + 8*t^6 + 5*t^4 + 9*t^3 + 12*t^2 + 8*t + 5) of
        Univariate Polynomial Ring in t over Finite Field of size 13
        To: Univariate Polynomial Ring in t over Finite Field of size 13
```

```python
class sage.rings.finite_rings.residue_field.ReductionMap
    Bases: sage.categories.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:
```
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 a 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 b 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)[]; h = t^5 + t^2 + 1
sage: k.<a> = R.residue_field(h) (continues on next page)
sage: K = R.fraction_field()
sage: f = k.convert_map_from(K)
sage: f.section()  # needs sage.libsntl
Lifting map:
    From: Residue field in a of Principal ideal (t^5 + t^2 + 1) of
        Univariate Polynomial Ring in t over Finite Field of size 2 (using GF2X)
    To:     Fraction Field of
        Univariate Polynomial Ring in t over Finite Field of size 2 (using GF2X)

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.
• \( \text{names} \) – 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)
An example where the generator of the number field doesn’t generate the residue class field:

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

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

```sage
# 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 ZZ[a] in the maximal order for all a:
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:**

```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)
```

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

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

**EXAMPLES:**

```sage
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) is ResidueField(P)  # indirect doctest  # needs sage.rings.number_field
True
```

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

**Bases:** `RingHomomorphism`

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

**EXAMPLES:**

```sage: # 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)
```
```python
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 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
  To: Residue field in a of Principal ideal (t^2 + 5) of
      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:
```
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:

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

```python
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.rings.finite_rings
sage: R.<t> = GF(29)[]; P = R.ideal(t^2 + 2); k.<a> = ResidueField(P); k
Residue field in a of Principal ideal (t^2 + 2) of Univariate Polynomial Ring in t over Finite Field of size 29
sage: type(k)  # 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:

sage: # needs sage.rings.number_field
sage: K.<z> = CyclotomicField(7)

sage: P = K.factor(17)[0][0]

sage: k = K.residue_field(P); k
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:

sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')

sage: K.<a> = NumberField(x^3 + x + 1)

sage: P = K.ideal(29).factor()[0][0]

sage: k = K.residue_field(P) # indirect doctest
sage: k.ideal() is P

(continues on next page)
True
sage: p = next_prime(2^40); p
1099511627791
sage: k = K.residue_field(K.prime_above(p))
sage: k.ideal().norm() == p
True

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

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

EXAMPLES:

sage: # 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: c = OK(a)
sage: b = k(a)
sage: k.lift(13*b + 5)
13*a + 5
sage: k.lift(12821*b + 918)
3*a + 19

sage: # 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: k.lift(a^2 + 5)
t^2 + 5

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

EXAMPLES:

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

```
sage: f.domain()
Residue field of Fractional ideal (a - 2)
sage: f.codomain()
Maximal Order in Number Field in a with defining polynomial \( x^3 - 3 \) with \( a = 1.442249570307409 \)
```

```
sage: f(k.0)
1
```
```
# 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.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
sage: f(a^2 + 5)
t^2 + 5
```

**reduction_map()**

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

**EXAMPLES:**

```
sage: # needs sage.rings.number_field sage.symbolic
sage: I = QQ[2^(1/3)].factor(2)[0][0]; I
Fractional ideal (a)
sage: k = I.residue_field(); k
Residue field of Fractional ideal (a)
sage: pi = k.reduction_map(); pi
Partially defined reduction map:
    From: Number Field in a with defining polynomial \( x^3 - 2 \) with \( a = 1.259921049894873 \)
    To: Residue field of Fractional ideal (a)
sage: pi.domain()
Number Field in a with defining polynomial \( x^3 - 2 \) with \( a = 1.259921049894873 \)
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)
```

(continues on next page)
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

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
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 `AlgebraicClosure FiniteField_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

```python
sage.rings.algebraic_closureFiniteField.AlgebraicClosureFiniteField(base_ring, name, category=\text{None}, implementation=\text{None}, \text{**kwds})
```

Construct an algebraic closure of a finite field.

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

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

EXAMPLES:

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

```python
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
sage: type(F.gen(2))
<class 'sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField_element_class'>
```

**asFiniteFieldElement(minimal=False)**

Return self as a finite field element.

**INPUT:**

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

**OUTPUT:**

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

**EXAMPLES:**

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

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

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

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:

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

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

EXAMPLES:

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

sage: a = F(1).change_level(3); a
1
sage: a.change_level(2)
1
sage: F.gen(3).change_level(1)
Traceback (most recent call last):
...
ValueError: z3 is not in the image of Ring morphism:
  From: Finite Field of size 3
  To: Finite Field in z3 of size 3^3
  Defn: 1 |--> 1

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

EXAMPLES:

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

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

EXAMPLES:

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

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

EXAMPLES:

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

**multiplicative_order()**
Return the multiplicative order of self.

EXAMPLES:

```python
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:

```python
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
```
Todo: This function could probably be made faster.

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

Return the $p^k$-th power of \texttt{self}, where $p$ is the characteristic of \texttt{self.parent()}. EXAMPLES:

\begin{verbatim}
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
\end{verbatim}

\textbf{pth\_root}(k=1)

Return the unique $p^k$-th root of \texttt{self}, where $p$ is the characteristic of \texttt{self.parent()}. EXAMPLES:

\begin{verbatim}
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(3).parent() is K
True
\end{verbatim}

\textbf{sqrt}(all=False)

Return a square root of \texttt{self}.

If the optional keyword argument \texttt{all} is set to \texttt{True}, return a list of all square roots of \texttt{self} instead. EXAMPLES:

\begin{verbatim}
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()
\end{verbatim}
class sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField_generic(base_ring, name, category=None)

Bases: Field

Algebraic closure of a finite field.

Element

alias of AlgebraicClosureFiniteFieldElement

algebraic_closure()

Return an algebraic closure of self.

This always returns self.

EXAMPLES:

```
sage: from sage.rings.algebraic_closure_finite_field import AlgebraicClosureFiniteField
sage: F = AlgebraicClosureFiniteField(GF(5), 'z')
sage: F.algebraic_closure() is F
True
```

characteristic()

Return the characteristic of self.

EXAMPLES:

```
sage: from sage.rings.algebraic_closure_finite_field import AlgebraicClosureFiniteField
sage: p = next_prime(1000)
sage: F = AlgebraicClosureFiniteField(GF(p), 'z')
sage: F.characteristic() == p
True
```

gen(n)

Return the \( n \)-th generator of self.

EXAMPLES:

```
sage: from sage.rings.algebraic_closure_finite_field import AlgebraicClosureFiniteField
sage: F = AlgebraicClosureFiniteField(GF(5), 'z')
sage: F.gen(2)
z2
```

gens()

Return a family of generators of self.

OUTPUT:
• a Family, indexed by the positive integers, whose $n$-th element is `self.gen(n)`.

EXAMPLES:

```python
sage: from sage.rings.algebraic_closureFiniteField 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:

```python
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:

```python
sage: from sage.rings.algebraic_closureFiniteField import AlgebraicClosureFiniteField
sage: AlgebraicClosureFiniteField(GF(5), 'z').ngens()
+Infinity
```

`some_elements()`

Return some elements of this field.

EXAMPLES:

```python
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(base_ring, name, category=None, lattice=None, use_database=True):

Bases: WithEqualityById, AlgebraicClosureFiniteField_generic

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

EXAMPLES:

sage: F = GF(5).algebraic_closure(implementation='pseudo_conway')
sage: F.cardinality()
+Infinity
sage: F.algebraic_closure() is F
True
sage: x = F(3).nth_root(12); x
z4^3 + z4^2 + 4*z4
sage: x**12
3

7.3 Routines for Conway and pseudo-Conway polynomials

AUTHORS:
- David Roe
- Jean-Pierre Flori
- Peter Bruin

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

Bases: WithEqualityById, SageObject

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

The Conway polynomial $f_n$ of degree $n$ over $F_p$ is defined by the following four conditions:
- $f_n$ is irreducible.
- In the quotient field $F_p[x]/(f_n)$, the element $x \mod f_n$ generates the multiplicative group.
• The minimal polynomial of \( (x \mod f_n)^{p^m-1} \) equals the Conway polynomial \( f_m \), for every divisor \( m \) of \( n \).

• \( f_n \) is lexicographically least among all such polynomials, under a certain ordering.

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

**INPUT:**

• \( p \) – prime number

• \( \text{use\_database} \) – boolean. If \text{True}, use actual Conway polynomials whenever they are available in the database. If \text{False}, always compute pseudo-Conway polynomials.

**EXAMPLES:**

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

**check\_consistency\( (n) \)**

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

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
sage: from sage.rings.finite_rings.conway_polynomials import PseudoConwayLattice
sage: PCL = PseudoConwayLattice(2, use_database=False)
sage: PCL.check_consistency(6)  
sage: PCL.check_consistency(60)  # long 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:**

```python
sage: # needs sage.rings.finite_rings
sage: from sage.rings.finite_rings.conway_polynomials import PseudoConwayLattice
sage: PCL = PseudoConwayLattice(2, use_database=False)
sage: PCL.polynomial(3)  
x^3 + x + 1
sage: PCL.polynomial(4)  
x^4 + x^3 + 1
sage: PCL.polynomial(60)  
```

(continues on next page)
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 \ RuntimeException \ exception.

**INPUT:**

- \( p \) – prime number
- \( n \) – positive integer

**OUTPUT:**

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

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

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

**EXAMPLES:**

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

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

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

**INPUT:**

- \( p \) – prime number
- \( n \) – positive integer

**OUTPUT:**

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

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

**EXAMPLES:**
<table>
<thead>
<tr>
<th>sage: exists_conway_polynomial(2,3)</th>
<th>#~ needs conway_polynomials</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td></td>
</tr>
<tr>
<td>sage: exists_conway_polynomial(2,-1)</td>
<td>False</td>
</tr>
<tr>
<td>sage: exists_conway_polynomial(97,200)</td>
<td>False</td>
</tr>
<tr>
<td>sage: exists_conway_polynomial(6,6)</td>
<td>False</td>
</tr>
</tbody>
</table>
CHAPTER
EIGHT

INDICES AND TABLES

• Index
• Module Index
• Search Page
sage.rings.algebraic_closure_finite_field,
131
sage.rings.finite_rings.conway_polynomials,
138
sage.rings.finite_rings.element_base, 65
sage.rings.finite_rings.element_givaro, 100
sage.rings.finite_rings.element_ntl_gf2e, 113
sage.rings.finite_rings.element_pari_ffelt, 89
sage.rings.finite_rings.finite_field_base, 50
sage.rings.finite_rings.finite_field_constructor, 41
sage.rings.finite_rings.finite_field_givaro, 95
sage.rings.finite_rings.finite_field_ntl_gf2e, 111
sage.rings.finite_rings.finite_field_pari_ffelt, 87
sage.rings.finite_rings.finite_field_prime_modn, 83
sage.rings.finite_rings.hom_finite_field, 76
sage.rings.finite_rings.hom_finite_field_givaro, 108
sage.rings.finite_rings.hom_prime_finite_field, 85
sage.rings.finite_rings.homset, 74
sage.rings.finite_rings.integer_mod, 15
sage.rings.finite_rings.integer_mod_ring, 1
sage.rings.finite_rings.residue_field, 119
INDEX

A

a_times_b_minus_c()
(sage.rings.finite_rings.element_givaro.Cache_givaro method), 102

a_times_b_minus_c()
(sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro method), 96

a_times_b_plus_c()
(sage.rings.finite_rings.element_givaro.Cache_givaro method), 102

a_times_b_plus_c()
(sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro method), 96

additive_order()
(sage.rings.finite_rings.element_base.FinitePolyExtElement method), 65

additive_order()
(sage.rings.finite_rings.integer_mod_ring.IntegerMod_abstract method), 17

algebraic_closure()
(sage.rings.finite_rings.finite_field_base.FiniteField method), 50

algebraic_closure()
(sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField pseudo_conway method), 138

AlgebraicClosureFiniteField() (in module sage.rings.algebraic_closure_finite_field), 131

AlgebraicClosureFiniteField_generic (class in sage.rings.algebraic_closure_finite_field), 136

AlgebraicClosureFiniteField_pseudo_conway (class in sage.rings.algebraic_closure_finite_field), 138

AlgebraicClosureFiniteFieldElement (class in sage.rings.algebraic_closure_finite_field), 132

asFiniteFieldElement()
(sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteFieldElement method), 132

C

c_minus_a_times_b()
(sage.rings.finite_rings.element_givaro.Cache_givaro method), 102

c_minus_a_times_b()
(sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro method), 96

Cache_base (class in sage.rings.finite_rings.element_base),
construction() (sage.rings.finite_rings.residue_field.ResidueField_generic, method), 83
conway_polynomial() (in module sage.rings.finite_rings.conway_polynomials), 140
crt() (sage.rings.finite_rings.element_givaro.Cache_givaro method), 103
crt() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic, method), 7
crt() (in module sage.rings.finite_rings.integer_mod_ring), 15
crt() (sage.rings.finite_rings.integer_mod.IntegerMod_abs_zero, method), 17
degree() (sage.rings.finite_rings.element_givaro.Cache_givaro method), 140
degree() (sage.rings.finite_rings.element_ntl_gf2e.Cache_ntl_gf2e method), 114
degree() (sage.rings.finite_rings.element_givaro.Cache_givaro method), 103
degree() (sage.rings.finite_rings.integer_mod.IntegerMod_abs_zero, method), 17
degree() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic, method), 7
degree() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic, method), 54
degree() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic, method), 112
degree() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic, method), 113
degree() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic, method), 97
degree() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic, method), 53

divides() (sage.rings.finite_rings.integer_mod.IntegerMod_base, method), 6
dual_basis() (sage.rings.finite_rings.element_givaro.Cache_givaro method), 103
dual_basis() (sage.rings.finite_rings.element_givaro.Cache_givaro method), 51

element_from_data() (sage.rings.finite_rings.element_givaro.Cache_givaro method), 103

exists_conway_polynomial() (in module sage.rings.finite_rings.conway_polynomials), 87

c class in

FiniteField_element (class in sage.rings.finite_rings.element_givaro.Cache_givaro method), 103
FiniteField_element (class in sage.rings.finite_rings.element_givaro.Cache_givaro method), 105
FiniteField_element (class in sage.rings.finite_rings.element_givaro.Cache_givaro method), 105
FiniteField_element (class in sage.rings.finite_rings.element_givaro.Cache_givaro method), 105
FiniteField_element (class in sage.rings.finite_rings.element_givaro.Cache_givaro method), 105
FiniteFieldHomomorphism_prime (class in sage.rings.finite_rings.hom_prime_finite_field), 85
FiniteFieldHomomorphism_generic (class in sage.rings.finite_rings.homset), 74
FiniteFieldHomomorphism_givaro (class in sage.rings.finite_rings.homset_givaro), 108
FiniteFieldHomomorphism_prime (class in sage.rings.finite_rings.hom_prime_finite_field), 85
FiniteFieldHomset (class in sage.rings.finite_rings.homset), 74
FinitePolyExtElement (class in sage.rings.finite_rings.element_base), 65
FiniteRingElement (class in sage.rings.finite_rings.element_base), 73
fixed_field() (sage.rings.finite_rings.hom_field.FrobeniusEndomorphism_finite_field method), 79
fixed_field() (sage.rings.finite_rings.hom_field_givaro.FrobeniusEndomorphism_givaro method), 108
fixed_field() (sage.rings.finite_rings.hom_prime_finite_field.FrobeniusEndomorphism_prime method), 85
free_module() (sage.rings.finite_rings.hom_field_base.FrobeniusEndomorphism_finite_field method), 54
frobenius() (sage.rings.finite_rings.element_base.FinitePolyExtElement pari_ffelt method), 66
frobenius() (sage.rings.finite_rings.element_pari_ffelt.FiniteField_element_pari_ffelt method), 90
frobenius_endomorphism() (sage.rings.finite_rings.hom_field_base.FiniteField method), 55
frobenius_endomorphism() (sage.rings.finite_rings.hom_field_givaro.FiniteField method), 97
FrobeniusEndomorphism_finite_field (class in sage.rings.finite_rings.hom_field_finite_field), 79
FrobeniusEndomorphism_givaro (class in sage.rings.finite_rings.hom_field_givaro), 108
FrobeniusEndomorphism_prime (class in sage.rings.finite_rings.hom_prime_finite_field), 85
from_integer() (sage.rings.finite_rings.hom_field_base.FiniteField method), 56
from_integer() (sage.rings.finite_rings.hom_field_givaro.FiniteField method), 98
from_integer() (sage.rings.finite_rings.hom_field_nlt_gf2e.FiniteField_nlt_gf2e method), 112

galois_group() (sage.rings.finite_rings.hom_field_base.FiniteFieldFiniteField method), 57
gcd() (sage.rings.finite_rings.integer_mod.IntegerMod_gmp method), 29
gcd() (sage.rings.finite_rings.integer_mod.IntegerMod_int method), 30
gcd() (sage.rings.finite_rings.integer_mod.IntegerMod_int64 method), 34
gen() (sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField method), 136
gen() (sage.rings.finite_rings.element_givaro.Cache_givaro method), 104
gen() (sage.rings.finite_rings.hom_field_base.FiniteField method), 57
gen() (sage.rings.finite_rings.hom_field_base.FiniteField_givaro method), 98
gen() (sage.rings.finite_rings.hom_field_nlt_gf2e.FiniteField_nlt_gf2e method), 112
gen() (sage.rings.finite_rings.hom_field_pari_ffelt.FiniteField_pari_ffelt method), 88
gen() (sage.rings.finite_rings.hom_field_prime_modn.FiniteField_prime_modn method), 84
generating_log() (sage.rings.finite_rings.hom_field_givaro.FiniteField_givaro method), 18
gen() (sage.rings.finite_rings.hom_field_base.FiniteField method), 54
get_object() (sage.rings.finite_rings.integer_mod_ring.IntegerModFactory method), 4
id() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 79
ideal() (sage.rings.finite_rings.residue_field.ResidueField_generic method), 127
import_data() (sage.rings.finite_rings.element_nlt_gf2e.Cache_nlt_gf2e method), 114
inclusion() (sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField method), 137
Index() (sage.rings.finite_rings.homset.FiniteFieldHomset method), 75
Int_to_IntegerMod (class in sage.rings.finite_rings.element_givaro.Cache_givaro method), 16
int_to_log() (sage.rings.finite_rings.element_givaro.Cache_givaro method), 104
int_to_log() (sage.rings.finite_rings.hom_field_givaro.FiniteField_givaro method), 99
integer_representation() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 67
Integer_to_IntegerMod (class in sage.rings.finite_rings.element_givaro.Cache_givaro method), 35
IntegerMod() (in module sage.rings.finite_rings.integ_mod), 16
IntegerMod_abstract (class in sage.rings.finite_rings.element_givaro.Cache_givaro method), 16
Finite Rings, Release 10.2

IntegerMod_gmp
in \texttt{IntegerMod}\_gmp
\texttt{sage.rings.finite_rings.integer_mod}, 29

IntegerMod_hom
in \texttt{IntegerMod}\_hom
\texttt{sage.rings.finite_rings.integer_mod}, 30

IntegerMod_int
in \texttt{IntegerMod}\_int
\texttt{sage.rings.finite_rings.integer_mod}, 30

IntegerMod\_int64
in \texttt{IntegerMod}\_int64
\texttt{sage.rings.finite_rings.integer_mod}, 33

IntegerMod\_to\_Integer
in \texttt{IntegerMod}\_to\_Integer
\texttt{sage.rings.finite_rings.integer_mod}, 35

IntegerMod\_to\_IntegerMod
in \texttt{IntegerMod}\_to\_IntegerMod
\texttt{sage.rings.finite_rings.integer_mod}, 35

IntegerModFactory
in \texttt{IntegerModFactory}
\texttt{sage.rings.finite_rings.integer_mod\_ring}, 2

IntegerModRing\_generic
in \texttt{IntegerModRing\_generic}
\texttt{sage.rings.finite_rings.integer_mod\_ring}, 4

inverse()
\texttt{inverse()}
\texttt{sage.rings.finite_rings.hom\_finite\_field.FrobeniusEndomorphism_finite_field}, 80

inverses
\texttt{inverses}
\texttt{sage.rings.finite_rings.integer_mod.NativeIntStruct}, 78
\texttt{sage.rings.finite_rings.homset.FiniteFieldHomset}, 87
\texttt{sage.rings.finite_rings.element_base.FinitePolyExtElement}, 107
\texttt{sage.rings.finite_rings.finite_field_base.FiniteField}, 134
\texttt{sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt}, 150
\texttt{sage.rings.finite_rings.element_givaro.FiniteField_givaroElement}, 159
\texttt{sage.rings.finite_rings.element_ntl_gf2e.FiniteField_ntl_gf2eElement}, 162

is\_aut()
\texttt{is\_aut()}
\texttt{sage.rings.finite_rings.homset.FiniteFieldHomset}, 75
\texttt{sage.rings.finite_rings.element_givaro.FiniteField_givaroElement}, 105
\texttt{sage.rings.finite_rings.element_ntl_gf2e.FiniteField_ntl_gf2eElement}, 115

is\_conway()
\texttt{is\_conway()}
\texttt{sage.rings.finite_rings.hom\_finite\_field.FrobeniusEndomorphism_finite_field}, 80

is\_field()
\texttt{is\_field()}
\texttt{sage.rings.finite_rings.hom\_finite\_field.FrobeniusEndomorphism_finite_field}, 80
\texttt{sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt}, 105
\texttt{sage.rings.finite_rings.element_givaro.FiniteField_givaroElement}, 115
\texttt{sage.rings.finite_rings.element_ntl_gf2e.FiniteField_ntl_gf2eElement}, 125

is\_field()
\texttt{is\_field()}
\texttt{sage.rings.finite_rings.hom\_finite\_field.FrobeniusEndomorphism_finite_field}, 80
\texttt{sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt}, 105
\texttt{sage.rings.finite_rings.element_givaro.FiniteField_givaroElement}, 115
\texttt{sage.rings.finite_rings.element_ntl_gf2e.FiniteField_ntl_gf2eElement}, 125

is\_FiniteField()
\texttt{is\_FiniteField()}
\texttt{sage.rings.finite_rings.finite_field_base.FiniteField}, 64
\texttt{sage.rings.finite_rings.finite_field_prime_modn.FiniteField_prime_modn}, 84

is\_FiniteFieldElement()
\texttt{is\_FiniteFieldElement()}
\texttt{sage.rings.finite_rings.finite_field_base.FiniteField}, 73

is\_identity()
\texttt{is\_identity()}
\texttt{sage.rings.finite_rings.hom\_finite\_field.FrobeniusEndomorphism_finite_field}, 80

is\_injective()
\texttt{is\_injective()}
\texttt{sage.rings.finite_rings.hom\_finite\_field.FrobeniusEndomorphism_finite_field}, 80
\texttt{sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt}, 105
\texttt{sage.rings.finite_rings.element_givaro.FiniteField_givaroElement}, 115
\texttt{sage.rings.finite_rings.element_ntl_gf2e.FiniteField_ntl_gf2eElement}, 125

is\_injective()
\texttt{is\_injective()}
\texttt{sage.rings.finite_rings.hom\_fiber\_field.FrobeniusEndomorphism_finite_field}, 80

is\_injective()
\texttt{is\_injective()}
\texttt{sage.rings.finite_rings.hom\_finite\_field.FrobeniusEndomorphism_finite_field}, 80
\texttt{sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt}, 115
\texttt{sage.rings.finite_rings.element_givaro.FiniteField_givaroElement}, 125
\texttt{sage.rings.finite_rings.element_ntl_gf2e.FiniteField_ntl_gf2eElement}, 135

is\_IntegerMod()
\texttt{is\_IntegerMod()}
\texttt{sage.rings.finite_rings.integer_mod\_ring.IntegerModRing\_generic}, 37

is\_integral\_domain()
\texttt{is\_integral\_domain()}
\texttt{sage.rings.finite_rings.integer_mod\_ring.IntegerModRing\_generic}, 37

is\_nilpotent()
\texttt{is\_nilpotent()}
\texttt{sage.rings.finite_rings.integer_mod\_ring.IntegerModRing\_generic}, 37

is\_noetherian()
\texttt{is\_noetherian()}
\texttt{sage.rings.finite_rings.integer_mod\_ring.IntegerModRing\_generic}, 37

is\_one()
\texttt{is\_one()}
\texttt{sage.rings.finite_rings.element_givaro.FiniteField_givaroElement}, 105

is\_one()
\texttt{is\_one()}
\texttt{sage.rings.finite_rings.element_ntl_gf2e.FiniteField_ntl_gf2eElement}, 115

is\_one()
\texttt{is\_one()}
\texttt{sage.rings.finite_rings.integer_mod\_ring.IntegerMod\_abstract}, 19

is\_primitive\_root()
\texttt{is\_primitive\_root()}
\texttt{sage.rings.finite_rings.integer_mod\_ring.IntegerMod\_abstract}, 19

is\_prime\_field()
\texttt{is\_prime\_field()}
\texttt{sage.rings.finite_rings.integer_mod\_ring.IntegerMod\_abstract}, 19

is\_square()
\texttt{is\_square()}
\texttt{sage.rings.finite_rings.element_base.FinitePolyExtElement}, 67

is\_square()
\texttt{is\_square()}
\texttt{sage.rings.finite_rings.element_givaro.FiniteField_givaroElement}, 105

is\_square()
\texttt{is\_square()}
\texttt{sage.rings.finite_rings.element_ntl_gf2e.FiniteField_ntl_gf2eElement}, 115

is\_surjective()
\texttt{is\_surjective()}
\texttt{sage.rings.finite_rings.hom\_finite\_field.FrobeniusEndomorphism_finite_field}, 80

is\_surjective()
\texttt{is\_surjective()}
\texttt{sage.rings.finite_rings.hom\_finite\_field.FrobeniusEndomorphism_finite_field}, 80
\texttt{sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt}, 105
\texttt{sage.rings.finite_rings.element_givaro.FiniteField_givaroElement}, 115
\texttt{sage.rings.finite_rings.element_ntl_gf2e.FiniteField_ntl_gf2eElement}, 125

is\_unit()
\texttt{is\_unit()}
\texttt{sage.rings.finite_rings.element_givaro.FiniteField_givaroElement}, 105

is\_unit()
\texttt{is\_unit()}
\texttt{sage.rings.finite_rings.element_ntl_gf2e.FiniteField_ntl_gf2eElement}, 115

Index
Finite Rings, Release 10.2

sage.rings.finite_rings.finite_field_constructor,  method, 11
sage.rings.finite_rings.finite_field_givaro,  NativeIntStruct (class in
sage.rings.finite_rings.integer_mod), 37
sage.rings.finite_rings.finite_field_ntl_gf2e, 95
sage.rings.finite_rings.finite_field_pari_ffelt, 87
sage.rings.finite_rings.finite_field_prime_modn, 83
sage.rings.finite_rings.homFinite_field, 76
sage.rings.finite_rings.homFinite_field_givaro, 108
sage.rings.finite_rings.hom_primeFinite_field, 85
sage.rings.finite_rings.homset, 74
sage.rings.finite_rings.integer_mod, 15
sage.rings.finite_rings.integer_mod_ring, 1
sage.rings.finite_rings.residue_field, 119
modulus() (sage.rings.finite_rings.finite_field_base.FiniteField
method), 58
modulus() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract
method), 22
modulus() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic
method), 10
multiplicative_generator()
(sage.rings.finite_rings.finite_field_base.FiniteField
method), 59
multiplicative_generator()
(sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic
method), 10
multiplicative_group_is_cyclic()
(sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic
method), 10
multiplicative_order()
(sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField
method), 134
multiplicative_order()
(sage.rings.finite_rings.element_base.FinitePolyExtElement
method), 69
multiplicative_order()
(sage.rings.finite_rings.element_givaro.FiniteField_givaro.method), 99
multiplicative_order()
(sage.rings.finite_rings.element_ntl_gf2e.FiniteField_prime_modn
method), 84
multiplicative_order()
(sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteFieldElement
method), 11
nth_root()
(sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteFieldElement
method), 134
nth_root()
(sage.rings.finite_rings.element_base.FinitePolyExtElement
method), 70
nth_root()
(sage.rings.finite_rings.integer_mod.IntegerMod_abstract
method), 22
O
order()
(sage.rings.finite_rings.element_givaro.Cache_givaro
method), 104
order()
(sage.rings.finite_rings.element_ntl_gf2e.Cache_ntl_gf2e
method), 114
order()
(sage.rings.finite_rings.finite_field_base.FiniteField
method), 59
order()
(sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro
method), 99
order()
(sage.rings.finite_rings.finite_field_prime_modn.FiniteField_prime_modn
method), 84
order()
(sage.rings.finite_rings.homFinite_field.FrobeniusEndomorphism
method), 81
order()
(sage.rings.finite_rings.homset.FiniteFieldHomset
method), 76
order()
(sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic
method), 11
order_c()
(sage.rings.finite_rings.element_givaro.Cache_givaro
method), 105
P
polynomial()
(sage.rings.finite_rings.conway_polynomials.PseudoConwayPolynomial
method), 139
polynomial()
(sage.rings.finite_rings.element_givaro.FiniteField_givaroElement
method), 106
polynomial()
(sage.rings.finite_rings.element_mod.ModElement
method), 92
polynomial()
(sage.rings.finite_rings.element_mod_ring.IntegerMod_ring
method), 22
polynomial()
(sage.rings.finite_rings.element_givaro.FiniteField_givaroElement
method), 106
polynomial()
(sage.rings.finite_rings.element_ntl_gf2e.Cache_ntl_gf2e
method), 114
polynomial()
(sage.rings.finite_rings.element_ntl_gf2e.FiniteField_prime_modn
method), 117
polynomial()
(sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement
method), 92
polynomial() (sage.rings.finite_rings.finite_field_base.FiniteField method), 60
polynomial() (sage.rings.finite_rings.finite_field_prime_modn.FiniteField_prime_modn method), 85
polynomial() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 24
polynomial_ring() (sage.rings.finite_rings.finite_field_base.FiniteField method), 60
power() (sage.rings.finite_rings.hom_finite_field.FrobeniusEndomorphism_finite_field method), 81
precompute_table() (sage.rings.finite_rings.integer_mod.NativeIntStruct method), 37
prime_subfield() (sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro method), 100
prime_subfield() (sage.rings.finite_rings.finite_field_ntl_gf2e.FiniteField_ntl_gf2e method), 113
primitive_element() (sage.rings.finite_rings.finite_field_base.FiniteField method), 60
PseudoConwayLattice (class in sage.rings.finite_rings.conway_polynomials), 138
pth_power() (sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteFieldElement method), 135
pth_power() (sage.rings.algebraic_closure_finite_field.Element method), 71
pth_power() (sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteFieldElement method), 93
pth_root() (sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteFieldElement method), 135
pth_root() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 71
quadratic_nonresidue() (sage.rings.finite_rings.integer_mod.IntegerModRing_generic method), 11
random_element() (sage.rings.finite_rings.element_givaro.Cache_givaro method), 105
random_element() (sage.rings.finite_rings.element_givaro.Cache_givaro attribute), 105
random_element() (sage.rings.finite_rings.element_givaro.Cache_givaro module), 105
random_element() (sage.rings.finite_rings.element_givaro.FiniteFieldElement_givaro module), 108
random_element() (sage.rings.finite_rings.hom_finite_field.FiniteFieldHomomorphism_global module), 124
random_element() (sage.rings.finite_rings.hom_finite_field.FiniteFieldHomomorphism_prime_modn module), 130
random_element() (sage.rings.finite_rings.hom_finite_field.FiniteFieldHomomorphism_local module), 126
random_element() (sage.rings.finite_rings.hom_finite_field.FiniteFieldHomomorphism_local module), 126
random_element() (sage.rings.finite_rings.hom_finite_field.FiniteFieldHomomorphism_local module), 126
reduction_map() (sage.rings.finite_rings.residue_field.ResidueField_generic method), 129
ReductionMap (class in sage.rings.finite_rings.residue_field), 120
repr (sage.rings.finite_rings.element_givaro.Cache_givaro attribute), 105
ResidueField_generic (class in sage.rings.finite_rings.residue_field), 126
ResidueFieldFactory (class in sage.rings.finite_rings.residue_field), 122
ResidueFieldHomomorphism_global (class in sage.rings.finite_rings.hom_finite_field), 124
ResidueFieldHomomorphism_prime_modn (class in sage.rings.finite_rings.hom_finite_field), 130
ResidueFieldHomomorphism_local (class in sage.rings.finite_rings.hom_finite_field), 126
R
random_element() (sage.rings.finite_rings.element_givaro.Cache_givaro method), 105
random_element() (sage.rings.finite_rings.element_givaro.Cache_givaro module), 105
random_element() (sage.rings.finite_rings.finite_field_base.FiniteField method), 61
random_element() (sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro method), 100
random_element() (sage.rings.finite_rings.integer_mod.IntegerMod_ring.Generic method), 12
rational_reconstruction() (sage.rings.finite_rings.integer_mod.IntegerMod_ring.Generic method), 24
reduction_map() (sage.rings.finite_rings.residue_field.ResidueField_generic method), 129
ReductionMap (class in sage.rings.finite_rings.residue_field), 120
S
section() (sage.rings.finite_rings.integer_mod.Integer_to_IntegerMod method), 36
QuadraticNonResidue (module), 11
QuadraticNonResidue (module), 11
Index 153
section() (sage.rings.finite_rings.residue_field.ReductionMap method), 121
section() (sage.rings.finite_rings.residue_field.ResidueFieldElement method), 125
SectionFinitefieldHomomorphism_generic (class in sage.rings.finite_rings.hom_finite_field), 81
SectionFinitefieldHomomorphism_givaro (class in sage.rings.finite_rings.hom_finite_field_givaro), 109
SectionFinitefieldHomomorphism_prime (class in sage.rings.finite_rings.hom_prime_finite_field), 86
some_elements() (sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField_generic method), 137
some_elements() (sage.rings.finite_rings.finite_field_base.FiniteField method), 61
sqrt() (sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteFieldElement method), 135
sqrt() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 72
sqrt() (sage.rings.finite_rings.element_ntl_gf2e.FiniteField_ntl_gf2eElement method), 117
sqrt() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 28
sqrt() (sage.rings.finite_rings.integer_mod.IntegerMod_int method), 32
square_root() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 72
square_root() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 27
square_root_mod_prime() (in module sage.rings.finite_rings.integer_mod), 39
square_root_mod_prime_power() (in module sage.rings.finite_rings.integer_mod), 39
square_roots_of_one() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 12
subfield() (sage.rings.algebraic_closure_finite_field.AlgebraicClosureFiniteField_generic method), 137
subfield() (sage.rings.finite_rings.finite_field_base.FiniteField method), 61
subfields() (sage.rings.finite_rings.finite_field_base.FiniteField method), 62
table (sage.rings.finite_rings.integer_mod.NativeIntStruct attribute), 37
to_integer() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 72
trace() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 73
trace() (sage.rings.finite_rings.integer_mod.IntegerMod_abs method), 28
unit_gens() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 12
unit_group() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 13
unit_group_exponent() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 15
unit_group_order() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 15
unpickle_Cache_givaro() (in module sage.rings.finite_rings.element_givaro), 108
unpickle_FiniteField_ext() (in module sage.rings.finite_rings.finite_field_base), 65
unpickle_FiniteField_givaroElement() (in module sage.rings.finite_rings.element_givaro), 108
unpickle_FiniteField_prm() (in module sage.rings.finite_rings.finite_field_base), 65
unpickle_FiniteFieldElement_pari_ffelt() (in module sage.rings.finite_rings.element_pari_ffelt), 94
unpickleFiniteField_ntl_gf2eElement() (in module sage.rings.finite_rings.integer_mod), 118
valuation() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_abstract method), 29
weight() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_abstract method), 118
zeta() (sage.rings.finite_rings.finite_field_base.FiniteField method), 63
zeta_order() (sage.rings.finite_rings.finite_field_base.FiniteField method), 64