Sage Reference Manual: Standard Commutative Rings

Release 6.8

The Sage Development Team

July 29, 2015

1 Ring Z of Integers 1

2 Elements of the ring Z of integers 13

3 Ring Z/nZ of integers modulo n 57

4 Elements of Z/nZ 71

5 Field Q of Rational Numbers 95

6 Rational Numbers 107

7 Finite Fields 131

8 Base class for finite field elements 139

9 Indices and Tables 147

Bibliography 149

CHAPTER

ONE

RING $\mathbb{Z}$ OF INTEGERS

The `IntegerRing_class` represents the ring $\mathbf{Z}$ of (arbitrary precision) integers. Each integer is an instance of `Integer`, which is defined in a Pyrex extension module that wraps GMP integers (the `mpz_t` type in GMP).

```
sage: Z = IntegerRing(); Z
Integer Ring
sage: Z.characteristic()
0
sage: Z.is_field()
False
```

There is a unique instance of the `integer ring`. To create an `Integer`, coerce either a Python int, long, or a string. Various other types will also coerce to the integers, when it makes sense.

```
sage: a = Z(1234); b = Z(5678); print a, b
1234 5678
sage: type(a)
<type 'sage.rings.integer.Integer'>
sage: a + b
6912
sage: Z('94803849083985934859834583945394')
94803849083985934859834583945394
```

sage.rings.integer_ring.**IntegerRing**()
Return the integer ring.

EXAMPLES:

```
sage: IntegerRing()
Integer Ring
sage: ZZ==IntegerRing()
True
```

class sage.rings.integer_ring.**IntegerRing_class**
Bases: sage.rings.ring.PrincipalIdealDomain

The ring of integers.

In order to introduce the ring $\mathbf{Z}$ of integers, we illustrate creation, calling a few functions, and working with its elements.

```
sage: Z = IntegerRing(); Z
Integer Ring
sage: Z.characteristic()
0
sage: Z.is_field()
False
```

```
sage: Z.category()
Join of Category of euclidean domains
    and Category of infinite enumerated sets
sage: Z(2^(2^5) + 1)
4294967297
```

One can give strings to create integers. Strings starting with `0x` are interpreted as hexadecimal, and strings starting with `0o` are interpreted as octal:

```
sage: parent('37')
<type 'str'>
sage: parent(Z('37'))
Integer Ring
sage: Z('0x10')
16
sage: Z('0x1a')
26
sage: Z('0o20')
16
```

As an inverse to `digits()`, lists of digits are accepted, provided that you give a base. The lists are interpreted in little-endian order, so that entry `i` of the list is the coefficient of `base^i`:

```
sage: Z([4,1,7],base=100)
70104
sage: Z([4,1,7],base=10)
714
sage: Z([3, 7], 10)
73
sage: Z([3, 7], 9)
66
sage: Z([], 10)
0
```

Alphanumeric strings can be used for bases 2..36; letters `a` to `z` represent numbers 10 to 36. Letter case does not matter.

```
sage: Z("sage",base=32)
928270
sage: Z("SAGE",base=32)
928270
sage: Z("Sage",base=32)
928270
sage: Z([14, 16, 10, 28],base=32)
928270
sage: 14 + 16*32 + 10*32^2 + 28*32^3
928270
```

We next illustrate basic arithmetic in $\mathbf{Z}$:

```
sage: a = Z(1234); b = Z(5678); print a, b
1234 5678
sage: type(a)
<type 'sage.rings.integer.Integer'>
sage: a + b
6912
sage: b + a
6912
sage: a * b
7006652
```

```
sage: b * a
7006652
sage: a - b
-4444
sage: b - a
4444
```

When we divide two integers using /, the result is automatically coerced to the field of rational numbers, even if the result is an integer.

```
sage: a / b
617/2839
sage: type(a/b)
<type 'sage.rings.rational.Rational'>
sage: a/a
1
sage: type(a/a)
<type 'sage.rings.rational.Rational'>
```

For floor division, use the // operator instead:

```
sage: a // b
0
sage: type(a//b)
<type 'sage.rings.integer.Integer'>
```

Next we illustrate arithmetic with automatic coercion. The types that coerce are: str, int, long, Integer.

```
sage: a + 17
1251
sage: a * 374
461516
sage: 374 * a
461516
sage: a/19
1234/19
sage: 0 + Z(-64)
-64
```

Integers can be coerced:

```
sage: a = Z(-64)
sage: int(a)
-64
```

We can create integers from several types of objects:

```
sage: Z(17/1)
17
sage: Z(Mod(19,23))
19
sage: Z(2 + 3*5 + O(5^3))
17
```

Arbitrary numeric bases are supported; strings or list of integers are used to provide the digits (more details in IntegerRing_class):

```
sage: Z("sage",base=32)
928270
sage: Z([14, 16, 10, 28],base=32)
928270
```

The `digits` method allows you to get the list of digits of an integer in a different basis (note that the digits are returned in little-endian order):

```
sage: b = Z([4,1,7],base=100)
sage: b
70104
sage: b.digits(base=71)
[27, 64, 13]

sage: Z(15).digits(2)
[1, 1, 1, 1]
sage: Z(15).digits(3)
[0, 2, 1]
```

The `str` method returns a string of the digits, using letters `a` to `z` to represent digits 10..36:

```
sage: Z(928270).str(base=32)
'sage'
```

Note that `str` only works with bases 2 through 36.

TESTS:

```
sage: TestSuite(ZZ).run()
sage: list(ZZ)
Traceback (most recent call last):
...
NotImplementedError: len() of an infinite set
```

absolute_degree()

Return the absolute degree of the integers, which is 1.

Here, absolute degree refers to the rank of the ring as a module over the integers.

EXAMPLES:

```
sage: ZZ.absolute_degree()
1
```

characteristic()

Return the characteristic of the integers, which is 0.

EXAMPLES:

```
sage: ZZ.characteristic()
0
```

completion(*p*, *prec*, *extras={}*)

Return the completion of the integers at the prime p.

INPUT:

- `p` – a prime (or `infinity`)
- `prec` – the desired precision
- `extras` – any further parameters to pass to the method used to create the completion.

OUTPUT:

- The completion of $\mathbf{Z}$ at p.

EXAMPLES:

```
sage: ZZ.completion(infinity, 53)
Real Field with 53 bits of precision
sage: ZZ.completion(5, 15, {'print_mode': 'bars'})
5-adic Ring with capped relative precision 15
```

degree()

Return the degree of the integers, which is 1.

Here, degree refers to the rank of the ring as a module over the integers.

EXAMPLES:

```
sage: ZZ.degree()
1
```

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

Return the order generated by the specified list of polynomials.

INPUT:

- `poly` – a list of one or more polynomials
- `names` – a parameter which will be passed to `EquationOrder()`.
- `embedding` – a parameter which will be passed to `EquationOrder()`.

OUTPUT:

- Given a single polynomial as input, return the order generated by a root of the polynomial in the field generated by a root of the polynomial.

  Given a list of polynomials as input, return the relative order generated by a root of the first polynomial in the list, over the order generated by the roots of the subsequent polynomials.

EXAMPLES:

```
sage: ZZ.extension(x^2-5, 'a')
Order in Number Field in a with defining polynomial x^2 - 5
sage: ZZ.extension([x^2 + 1, x^2 + 2], 'a,b')
Relative Order in Number Field in a with defining polynomial
x^2 + 1 over its base field
```

fraction_field()

Return the field of rational numbers - the fraction field of the integers.

EXAMPLES:

```
sage: ZZ.fraction_field()
Rational Field
sage: ZZ.fraction_field() == QQ
True
```

gen(*n=0*)

Return the additive generator of the integers, which is 1.

INPUT:

- `n` (default: 0) – In a ring with more than one generator, the optional parameter n indicates which generator to return; since there is only one generator in this case, the only valid value for n is 0.

EXAMPLES:

```
sage: ZZ.gen()
1
sage: type(ZZ.gen())
<type 'sage.rings.integer.Integer'>
```

gens()

Return the tuple `(1,)` containing a single element, the additive generator of the integers, which is 1.

EXAMPLES:

```
sage: ZZ.gens(); ZZ.gens()[0]
(1,)
1
sage: type(ZZ.gens()[0])
<type 'sage.rings.integer.Integer'>
```

is_field(*proof=True*)

Return `False` since the integers are not a field.

EXAMPLES:

```
sage: ZZ.is_field()
False
```

is_finite()

Return `False` since the integers are an infinite ring.

EXAMPLES:

```
sage: ZZ.is_finite()
False
```

is_integrally_closed()

Return that the integer ring is, in fact, integrally closed.

EXAMPLES:

```
sage: ZZ.is_integrally_closed()
True
```

is_noetherian()

Return `True` since the integers are a Noetherian ring.

EXAMPLES:

```
sage: ZZ.is_noetherian()
True
```

is_subring(*other*)

Return `True` if $\mathbf{Z}$ is a subring of other in a natural way.

Every ring of characteristic 0 contains $\mathbf{Z}$ as a subring.

EXAMPLES:

```
sage: ZZ.is_subring(QQ)
True
```

krull_dimension()

Return the Krull dimension of the integers, which is 1.

EXAMPLES:

```
sage: ZZ.krull_dimension()
1
```

ngens()
Return the number of additive generators of the ring, which is 1.

EXAMPLES:

```
sage: ZZ.ngens()
1
sage: len(ZZ.gens())
1
```

order()
Return the order (cardinality) of the integers, which is `+Infinity`.

EXAMPLES:

```
sage: ZZ.order()
+Infinity
```

parameter()
Return an integer of degree 1 for the Euclidean property of **Z**, namely 1.

EXAMPLES:

```
sage: ZZ.parameter()
1
```

quotient(*I*, *names=None*)
Return the quotient of **Z** by the ideal or integer `I`.

EXAMPLES:

```
sage: ZZ.quo(6*ZZ)
Ring of integers modulo 6
sage: ZZ.quo(0*ZZ)
Integer Ring
sage: ZZ.quo(3)
Ring of integers modulo 3
sage: ZZ.quo(3*QQ)
Traceback (most recent call last):
...
TypeError: I must be an ideal of ZZ
```

random_element(*x=None*, *y=None*, *distribution=None*)
Return a random integer.

INPUT:

- `x`, `y` integers – bounds for the result.
- **distribution– a string:**
  - `'uniform'`
  - `'mpz_rrandomb'`
  - `'1/n'`
  - `'gaussian'`

OUTPUT:

- With no input, return a random integer.

  If only one integer x is given, return an integer between 0 and $x - 1$.

  If two integers are given, return an integer between x and $y - 1$ inclusive.

  If at least one bound is given, the default distribution is the uniform distribution; otherwise, it is the distribution described below.

  If the distribution `'1/n'` is specified, the bounds are ignored.

  If the distribution `'mpz_rrandomb'` is specified, the output is in the range from 0 to $2^x - 1$.

  If the distribution `'gaussian'` is specified, the output is sampled from a discrete Gaussian distribution with parameter $\sigma = x$ and centered at zero. That is, the integer v is returned with probability proportional to $\exp(-v^2/(2\sigma^2))$. See `sage.stats.distributions.discrete_gaussian_integer` for details. Note that if many samples from the same discrete Gaussian distribution are needed, it is faster to construct a `sage.stats.distributions.discrete_gaussian_integer.DiscreteGaussianDistributionI` object which is then repeatedly queried.

The default distribution for `ZZ.random_element()` is based on $X = \text{trunc}(4/(5R))$, where R is a random variable uniformly distributed between -1 and 1. This gives $\Pr(X = 0) = 1/5$, and $\Pr(X = n) = 2/(5|n|(|n| + 1))$ for $n \neq 0$. Most of the samples will be small; -1, 0, and 1 occur with probability $1/5$ each. But we also have a small but non-negligible proportion of "outliers"; $\Pr(|X| \geq n) = 4/(5n)$, so for instance, we expect that $|X| \geq 1000$ on one in 1250 samples.

We actually use an easy-to-compute truncation of the above distribution; the probabilities given above hold fairly well up to about $|n| = 10000$, but around $|n| = 30000$ some values will never be returned at all, and we will never return anything greater than 2^{30}.

EXAMPLES:

```
sage: [ZZ.random_element() for _ in range(10)]
[-8, 2, 0, 0, 1, -1, 2, 1, -95, -1]
```

The default uniform distribution is integers in $[-2, 2]$:

```
sage: [ZZ.random_element(distribution="uniform") for _ in range(10)]
[2, -2, 2, -2, -1, 1, -1, 2, 1, 0]
```

Here we use the distribution `'1/n'`:

```
sage: [ZZ.random_element(distribution="1/n") for _ in range(10)]
[-6, 1, -1, 1, 1, -1, 1, -1, -3, 1]
```

If a range is given, the default distribution is uniform in that range:

```
sage: ZZ.random_element(-10,10)
-2
sage: ZZ.random_element(10)
2
sage: ZZ.random_element(10^50)
95316047862915367272947233286221109019733658989 88
sage: [ZZ.random_element(5) for _ in range(10)]
[3, 1, 2, 3, 0, 0, 3, 4, 0, 3]
```

Notice that the right endpoint is not included:

```
sage: [ZZ.random_element(-2,2) for _ in range(10)]
[1, -2, -2, -1, -2, -1, -1, -2, 0, -2]
```

We compute a histogram over 1000 samples of the default distribution:

```
sage: from collections import defaultdict
sage: d = defaultdict(lambda: 0)
sage: for _ in range(1000):
...       samp = ZZ.random_element()
...       d[samp] = d[samp] + 1

sage: sorted(d.items())
[(-1955, 1), (-1026, 1), (-357, 1), (-248, 1), (-145, 1), (-81, 1), (-80, 1), (-79, 1), (-75
```

We return a sample from a discrete Gaussian distribution:

```
sage: ZZ.random_element(11.0, distribution="gaussian")
5
```

range(*start*, *end=None*, *step=None*)

Optimized range function for Sage integers.

AUTHORS:

- Robert Bradshaw (2007-09-20)

EXAMPLES:

```
sage: ZZ.range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: ZZ.range(-5,5)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
sage: ZZ.range(0,50,5)
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
sage: ZZ.range(0,50,-5)
[]
sage: ZZ.range(50,0,-5)
[50, 45, 40, 35, 30, 25, 20, 15, 10, 5]
sage: ZZ.range(50,0,5)
[]
sage: ZZ.range(50,-1,-5)
[50, 45, 40, 35, 30, 25, 20, 15, 10, 5, 0]
```

It uses different code if the step doesn't fit in a long:

```
sage: ZZ.range(0,2^83,2^80)
[0, 1208925819614629174706176, 2417851639229258349412352, 3626777458843887524118528, 4835703
```

Make sure trac ticket #8818 is fixed:

```
sage: ZZ.range(1r, 10r)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

residue_field(*prime*, *check=True*)

Return the residue field of the integers modulo the given prime, i.e. $\mathbf{Z}/p\mathbf{Z}$.

INPUT:

- `prime` - a prime number
- `check` - (boolean, default `True`) whether or not to check the primality of prime.

OUTPUT: The residue field at this prime.

EXAMPLES:

```
sage: F = ZZ.residue_field(61); F
Residue field of Integers modulo 61
sage: pi = F.reduction_map(); pi
Partially defined reduction map:
  From: Rational Field
  To:   Residue field of Integers modulo 61
sage: pi(123/234)
6
sage: pi(1/61)
Traceback (most recent call last):
...
ZeroDivisionError: Cannot reduce rational 1/61 modulo 61:
it has negative valuation
sage: lift = F.lift_map(); lift
Lifting map:
  From: Residue field of Integers modulo 61
  To:   Integer Ring
sage: lift(F(12345/67890))
33
sage: (12345/67890) % 61
33
```

Construction can be from a prime ideal instead of a prime:

```
sage: ZZ.residue_field(ZZ.ideal(97))
Residue field of Integers modulo 97
```

TESTS:

```
sage: ZZ.residue_field(ZZ.ideal(96))
Traceback (most recent call last):
...
TypeError: Principal ideal (96) of Integer Ring is not prime
sage: ZZ.residue_field(96)
Traceback (most recent call last):
...
TypeError: 96 is not prime
```

zeta(*n=2*)

Return a primitive n-th root of unity in the integers, or raise an error if none exists.

INPUT:

- n – (default 2) a positive integer

OUTPUT:

- an n-th root of unity in **Z**.

EXAMPLE:

```
sage: ZZ.zeta()
-1
sage: ZZ.zeta(1)
1
sage: ZZ.zeta(3)
Traceback (most recent call last):
...
ValueError: no nth root of unity in integer ring
sage: ZZ.zeta(0)
Traceback (most recent call last):
```

```
...
ValueError: n must be positive in zeta()
```

sage.rings.integer_ring.**crt_basis**(*X*, *xgcd=None*)

Compute and return a Chinese Remainder Theorem basis for the list `X` of coprime integers.

INPUT:

- `X` – a list of Integers that are coprime in pairs.
- `xgcd` – an optional parameter which is ignored.

OUTPUT:

- `E` - a list of Integers such that `E[i] = 1` (mod `X[i]`) and `E[i] = 0` (mod `X[j]`) for all $j \neq i$.

For this explanation, let `E[i]` be denoted by E_i.

The E_i have the property that if A is a list of objects, e.g., integers, vectors, matrices, etc., where A_i is understood modulo X_i, then a CRT lift of A is simply

$$\sum_i E_i A_i.$$

ALGORITHM: To compute E_i, compute integers s and t such that

$$sX_i + t\prod_{i \neq j} X_j = 1.()$$

Then

$$E_i = t\prod_{i \neq j} X[j].$$

Notice that equation (*) implies that E_i is congruent to 1 modulo X_i and to 0 modulo the other X_j for $j \neq i$.

COMPLEXITY: We compute `len(X)` extended GCD's.

EXAMPLES:

```
sage: X = [11,20,31,51]
sage: E = crt_basis([11,20,31,51])
sage: E[0]%X[0], E[1]%X[0], E[2]%X[0], E[3]%X[0]
(1, 0, 0, 0)
sage: E[0]%X[1], E[1]%X[1], E[2]%X[1], E[3]%X[1]
(0, 1, 0, 0)
sage: E[0]%X[2], E[1]%X[2], E[2]%X[2], E[3]%X[2]
(0, 0, 1, 0)
sage: E[0]%X[3], E[1]%X[3], E[2]%X[3], E[3]%X[3]
(0, 0, 0, 1)
```

sage.rings.integer_ring.**is_IntegerRing**(*x*)

Internal function: return `True` iff `x` is the ring **Z** of integers.

TESTS:

```
sage: from sage.rings.integer_ring import is_IntegerRing
sage: is_IntegerRing(ZZ)
True
sage: is_IntegerRing(QQ)
False
sage: is_IntegerRing(parent(3))
True
```

```
sage: is_IntegerRing(parent(1/3))
False
```

CHAPTER

TWO

ELEMENTS OF THE RING $\mathbb{Z}$ OF INTEGERS

AUTHORS:

- William Stein (2005): initial version
- Gonzalo Tornaria (2006-03-02): vastly improved python/GMP conversion; hashing
- Didier Deshommes (2006-03-06): numerous examples and docstrings
- William Stein (2006-03-31): changes to reflect GMP bug fixes
- William Stein (2006-04-14): added GMP factorial method (since it's now very fast).
- David Harvey (2006-09-15): added nth_root, exact_log
- David Harvey (2006-09-16): attempt to optimise Integer constructor
- Rishikesh (2007-02-25): changed quo_rem so that the rem is positive
- David Harvey, Martin Albrecht, Robert Bradshaw (2007-03-01): optimized Integer constructor and pool
- Pablo De Napoli (2007-04-01): multiplicative_order should return +infinity for non zero numbers
- Robert Bradshaw (2007-04-12): is_perfect_power, Jacobi symbol (with Kronecker extension). Convert some methods to use GMP directly rather than PARI, Integer(), PY_NEW(Integer)
- David Roe (2007-03-21): sped up valuation and is_square, added val_unit, is_power, is_power_of and divide_knowing_divisible_by
- Robert Bradshaw (2008-03-26): gamma function, multifactorials
- Robert Bradshaw (2008-10-02): bounded squarefree part
- David Loeffler (2011-01-15): fixed bug #10625 (inverse_mod should accept an ideal as argument)
- Vincent Delecroix (2010-12-28): added unicode in Integer.__init__
- David Roe (2012-03): deprecate `is_power()` in favour of `is_perfect_power()` (see trac ticket #12116)

EXAMPLES:

Add 2 integers:

```
sage: a = Integer(3) ; b = Integer(4)
sage: a + b == 7
True
```

Add an integer and a real number:

```
sage: a + 4.0
7.00000000000000
```

Add an integer and a rational number:

```
sage: a + Rational(2)/5
17/5
```

Add an integer and a complex number:

```
sage: b = ComplexField().0 + 1.5
sage: loads((a+b).dumps()) == a+b
True

sage: z = 32
sage: -z
-32
sage: z = 0; -z
0
sage: z = -0; -z
0
sage: z = -1; -z
1
```

Multiplication:

```
sage: a = Integer(3) ; b = Integer(4)
sage: a * b == 12
True
sage: loads((a * 4.0).dumps()) == a*b
True
sage: a * Rational(2)/5
6/5

sage: list([2,3]) * 4
[2, 3, 2, 3, 2, 3, 2, 3]

sage: 'sage'*Integer(3)
'sagesagesage'
```

COERCIONS:

Returns version of this integer in the multi-precision floating real field R:

```
sage: n = 9390823
sage: RR = RealField(200)
sage: RR(n)
9.3908230000000000000000000000000000000000000000000000000000e6
```

sage.rings.integer.**GCD_list**(*v*)

Return the greatest common divisor of a list of integers.

INPUT:

- v – list or tuple

Elements of v are converted to Sage integers. An empty list has GCD zero.

This function is used, for example, by `rings/arith.py`.

EXAMPLES:

```
sage: from sage.rings.integer import GCD_list
sage: w = GCD_list([3,9,30]); w
```

```
3
sage: type(w)
<type 'sage.rings.integer.Integer'>
```

Check that the bug reported in trac ticket #3118 has been fixed:

```
sage: sage.rings.integer.GCD_list([2,2,3])
1
```

The inputs are converted to Sage integers.

```
sage: w = GCD_list([int(3), int(9), '30']); w
3
sage: type(w)
<type 'sage.rings.integer.Integer'>
```

Check that the GCD of the empty list is zero (trac ticket #17257):

```
sage: GCD_list([])
0
```

class sage.rings.integer.**Integer**

Bases: sage.structure.element.EuclideanDomainElement

The Integer class represents arbitrary precision integers. It derives from the Element class, so integers can be used as ring elements anywhere in Sage.

Integer() interprets strings that begin with 0o as octal numbers, strings that begin with 0x as hexadecimal numbers and strings that begin with 0b as binary numbers.

The class Integer is implemented in Cython, as a wrapper of the GMP mpz_t integer type.

EXAMPLES:

```
sage: Integer(123)
123
sage: Integer("123")
123
```

Sage Integers support **PEP 3127** literals:

```
sage: Integer('0x12')
18
sage: Integer('-0o12')
-10
sage: Integer('+0b101010')
42
```

Conversion from PARI:

```
sage: Integer(pari('-10380104371593008048799446356441519384'))
-10380104371593008048799446356441519384
sage: Integer(pari('Pol([-3])'))
-3
```

additive_order()

Return the additive order of self.

EXAMPLES:

```
sage: ZZ(0).additive_order()
1
```

```
sage: ZZ(1).additive_order()
+Infinity
```

binary()

Return the binary digits of self as a string.

EXAMPLES:

```
sage: print Integer(15).binary()
1111
sage: print Integer(16).binary()
10000
sage: print Integer(169384023840928430928430983243).binary()
1101101011101100011110001110010010100111010001101010001111111000101000000000101111000010000
```

binomial(*m*, *algorithm='mpir'*)

Return the binomial coefficient "self choose m".

INPUT:

- m – an integer
- algorithm – 'mpir' (default) or 'pari'; 'mpir' is faster for small m, and 'pari' tends to be faster for large m

OUTPUT:

- integer

EXAMPLES:

```
sage: 10.binomial(2)
45
sage: 10.binomial(2, algorithm='pari')
45
sage: 10.binomial(-2)
0
sage: (-2).binomial(3)
-4
sage: (-3).binomial(0)
1
```

The argument m or (self-m) must fit into unsigned long:

```
sage: (2**256).binomial(2**256)
1
sage: (2**256).binomial(2**256-1)
115792089237316195423570985008687907853269984665640564039457584007913129639936
sage: (2**256).binomial(2**128)
Traceback (most recent call last):
...
OverflowError: m must fit in an unsigned long
```

TESTS:

```
sage: 0.binomial(0)
1
sage: 0.binomial(1)
0
sage: 0.binomial(-1)
0
```

```
sage: 13.binomial(2r)
78
```

Check that it can be interrupted:

```
sage: alarm(0.5); (2^100).binomial(2^22, algorithm='mpir')
Traceback (most recent call last):
...
AlarmInterrupt
sage: alarm(0.5); (2^100).binomial(2^22, algorithm='pari')
Traceback (most recent call last):
...
AlarmInterrupt
```

bits()

Return the bits in self as a list, least significant first. The result satisfies the identity

```
x == sum(b*2^e for e, b in enumerate(x.bits()))
```

Negative numbers will have negative "bits". (So, strictly speaking, the entries of the returned list are not really members of $\mathbf{Z}/2\mathbf{Z}$.)

This method just calls `digits()` with `base=2`.

SEE ALSO:

`nbits()` (number of bits; a faster way to compute `len(x.bits())`; and `binary()`, which returns a string in more-familiar notation.

EXAMPLES:

```
sage: 500.bits()
[0, 0, 1, 0, 1, 1, 1, 1, 1]
sage: 11.bits()
[1, 1, 0, 1]
sage: (-99).bits()
[-1, -1, 0, 0, 0, -1, -1]
```

ceil()

Return the ceiling of self, which is self since self is an integer.

EXAMPLES:

```
sage: n = 6
sage: n.ceil()
6
```

class_number(*proof=True*)

Returns the class number of the quadratic order with this discriminant.

INPUT:

- `self` – an integer congruent to 0 or 1 $\mod 4$ which is not a square
- `proof` (boolean, default `True`) – if `False` then for negative disscriminants a faster algorithm is used by the PARI library which is known to give incorrect results when the class group has many cyclic factors.

OUTPUT:

(integer) the class number of the quadratic order with this discriminant.

Note: This is not always equal to the number of classes of primitive binary quadratic forms of discriminant D, which is equal to the narrow class number. The two notions are the same when $D < 0$, or $D > 0$ and the fundamental unit of the order has negative norm; otherwise the number of classes of forms is twice this class number.

EXAMPLES:

```
sage: (-163).class_number()
1
sage: (-104).class_number()
6
sage: [((4*n+1),(4*n+1).class_number()) for n in [21..29]]
[(85, 2),
(89, 1),
(93, 1),
(97, 1),
(101, 1),
(105, 2),
(109, 1),
(113, 1),
(117, 1)]
```

TESTS:

The integer must not be a square or an error is raised:

```
sage: 100.class_number()
Traceback (most recent call last):
...
ValueError: class_number not defined for square integers
```

The integer must be 0 or 1 mod 4 or an error is raised:

```
sage: 10.class_number()
Traceback (most recent call last):
...
ValueError: class_number only defined for integers congruent to 0 or 1 modulo 4
sage: 3.class_number()
Traceback (most recent call last):
...
ValueError: class_number only defined for integers congruent to 0 or 1 modulo 4
```

conjugate()

Return the complex conjugate of this integer, which is the integer itself.

EXAMPLES: sage: n = 205 sage: n.conjugate() 205

coprime_integers(m)

Return the positive integers $< m$ that are coprime to self.

EXAMPLES:

```
sage: n = 8
sage: n.coprime_integers(8)
[1, 3, 5, 7]
sage: n.coprime_integers(11)
[1, 3, 5, 7, 9]
sage: n = 5; n.coprime_integers(10)
[1, 2, 3, 4, 6, 7, 8, 9]
sage: n.coprime_integers(5)
[1, 2, 3, 4]
```

```
sage: n = 99; n.coprime_integers(99)
[1, 2, 4, 5, 7, 8, 10, 13, 14, 16, 17, 19, 20, 23, 25, 26, 28, 29, 31, 32, 34, 35, 37, 38, 4
```

AUTHORS:

- Naqi Jaffery (2006-01-24): examples

ALGORITHM: Naive - compute lots of GCD's. If this isn't good enough for you, please code something better and submit a patch.

crt(*y*, *m*, *n*)

Return the unique integer between 0 and mn that is congruent to the integer modulo m and to y modulo n. We assume that m and n are coprime.

EXAMPLES:

```
sage: n = 17
sage: m = n.crt(5, 23, 11); m
247
sage: m%23
17
sage: m%11
5
```

denominator()

Return the denominator of this integer, which of course is always 1.

EXAMPLES:

```
sage: x = 5
sage: x.denominator()
1
sage: x = 0
sage: x.denominator()
1
```

digits(*base=10*, *digits=None*, *padto=0*)

Return a list of digits for `self` in the given base in little endian order.

The returned value is unspecified if self is a negative number and the digits are given.

INPUT:

- `base` - integer (default: 10)
- `digits` - optional indexable object as source for the digits
- `padto` - the minimal length of the returned list, sufficient number of zeros are added to make the list minimum that length (default: 0)

As a shorthand for `digits(2)`, you can use `bits()`.

Also see `ndigits()`.

EXAMPLES:

```
sage: 17.digits()
[7, 1]
sage: 5.digits(base=2, digits=["zero","one"])
['one', 'zero', 'one']
sage: 5.digits(3)
[2, 1]
sage: 0.digits(base=10)  # 0 has 0 digits
[]
```

```
sage: 0.digits(base=2)  # 0 has 0 digits
[]
sage: 10.digits(16,'0123456789abcdef')
['a']
sage: 0.digits(16,'0123456789abcdef')
[]
sage: 0.digits(16,'0123456789abcdef',padto=1)
['0']
sage: 123.digits(base=10,padto=5)
[3, 2, 1, 0, 0]
sage: 123.digits(base=2,padto=3)       # padto is the minimal length
[1, 1, 0, 1, 1, 1, 1]
sage: 123.digits(base=2,padto=10,digits=(1,-1))
[-1, -1, 1, -1, -1, -1, -1, 1, 1, 1]
sage: a=9939082340; a.digits(10)
[0, 4, 3, 2, 8, 0, 9, 3, 9, 9]
sage: a.digits(512)
[100, 302, 26, 74]
sage: (-12).digits(10)
[-2, -1]
sage: (-12).digits(2)
[0, 0, -1, -1]
```

We support large bases.

```
sage: n=2^6000
sage: n.digits(2^3000)
[0, 0, 1]

sage: base=3; n=25
sage: l=n.digits(base)
sage: # the next relationship should hold for all n,base
sage: sum(base^i*l[i] for i in range(len(l)))==n
True
sage: base=3; n=-30; l=n.digits(base); sum(base^i*l[i] for i in range(len(l)))==n
True
```

The inverse of this method – constructing an integer from a list of digits and a base – can be done using the above method or by simply using `ZZ()` with a base:

```
sage: x = 123; ZZ(x.digits(), 10)
123
sage: x == ZZ(x.digits(6), 6)
True
sage: x == ZZ(x.digits(25), 25)
True
```

Using `sum()` and `enumerate()` to do the same thing is slightly faster in many cases (and `balanced_sum()` may be faster yet). Of course it gives the same result:

```
sage: base = 4
sage: sum(digit * base^i for i, digit in enumerate(x.digits(base))) == ZZ(x.digits(base), ba
True
```

Note: In some cases it is faster to give a digits collection. This would be particularly true for computing the digits of a series of small numbers. In these cases, the code is careful to allocate as few python objects as reasonably possible.

```
sage: digits = range(15)
sage: l=[ZZ(i).digits(15,digits) for i in range(100)]
sage: l[16]
[1, 1]
```

This function is comparable to `str` for speed.

```
sage: n=3^100000
sage: n.digits(base=10)[-1]  # slightly slower than str
1
sage: n=10^10000
sage: n.digits(base=10)[-1]  # slightly faster than str
1
```

AUTHORS:

- Joel B. Mohler (2008-03-02): significantly rewrote this entire function

divide_knowing_divisible_by(*right*)

Returns the integer self / right when self is divisible by right.

If self is not divisible by right, the return value is undefined, and may not even be close to self/right for multi-word integers.

EXAMPLES:

```
sage: a = 8; b = 4
sage: a.divide_knowing_divisible_by(b)
2
sage: (100000).divide_knowing_divisible_by(25)
4000
sage: (100000).divide_knowing_divisible_by(26) # close (random)
3846
```

However, often it's way off.

```
sage: a = 2^70; a
1180591620717411303424
sage: a // 11  # floor divide
107326510974310118493
sage: a.divide_knowing_divisible_by(11) # way off and possibly random
432153614787434223889704550 40
```

divides(*n*)

Return True if self divides n.

EXAMPLES:

```
sage: Z = IntegerRing()
sage: Z(5).divides(Z(10))
True
sage: Z(0).divides(Z(5))
False
sage: Z(10).divides(Z(5))
False
```

divisors()

Returns a list of all positive integer divisors of the integer self.

EXAMPLES:

```
sage: a = -3; a.divisors()
[1, 3]
sage: a = 6; a.divisors()
[1, 2, 3, 6]
sage: a = 28; a.divisors()
[1, 2, 4, 7, 14, 28]
sage: a = 2^5; a.divisors()
[1, 2, 4, 8, 16, 32]
sage: a = 100; a.divisors()
[1, 2, 4, 5, 10, 20, 25, 50, 100]
sage: a = 1; a.divisors()
[1]
sage: a = 0; a.divisors()
Traceback (most recent call last):
...
ValueError: n must be nonzero
sage: a = 2^3 * 3^2 * 17; a.divisors()
[1, 2, 3, 4, 6, 8, 9, 12, 17, 18, 24, 34, 36, 51, 68, 72, 102, 136, 153, 204, 306, 408, 612,
sage: a = odd_part(factorial(31))
sage: v = a.divisors(); len(v)
172800
sage: prod(e+1 for p,e in factor(a))
172800
sage: all([t.divides(a) for t in v])
True

sage: n = 2^551 - 1
sage: L = n.divisors()
sage: len(L)
256
sage: L[-1] == n
True
```

TESTS:

```
sage: prod(primes_first_n(64)).divisors()
Traceback (most recent call last):
...
OverflowError: value too large
sage: prod(primes_first_n(58)).divisors()
Traceback (most recent call last):
...
OverflowError: value too large                                  # 32-bit
MemoryError: failed to allocate 288230376151711744 * 24 bytes  # 64-bit
```

Check for memory leaks and ability to interrupt (the `divisors` call below allocates about 800 MB every time, so a memory leak will not go unnoticed):

```
sage: n = prod(primes_first_n(25))
sage: for i in range(20):  # long time
....:     try:
....:         alarm(RDF.random_element(1e-3, 0.5))
....:         _ = n.divisors()
....:         cancel_alarm()  # we never get here
....:     except AlarmInterrupt:
....:         pass
```

Note: If one first computes all the divisors and then sorts it, the sorting step can easily dominate the

runtime. Note, however, that (non-negative) multiplication on the left preserves relative order. One can leverage this fact to keep the list in order as one computes it using a process similar to that of the merge sort algorithm.

---

euclidean_degree()

Return the degree of this element as an element of a euclidean domain.

If this is an element in the ring of integers, this is simply its absolute value.

EXAMPLES:

```
sage: ZZ(1).euclidean_degree()
1
```

exact_log(m)

Returns the largest integer k such that $m^k \leq$ self, i.e., the floor of $\log_m(\text{self})$.

This is guaranteed to return the correct answer even when the usual log function doesn't have sufficient precision.

INPUT:

- m - integer >= 2

AUTHORS:

- David Harvey (2006-09-15)
- **Joel B. Mohler (2009-04-08) – rewrote this to handle small cases** and/or easy cases up to 100x faster..

EXAMPLES:

```
sage: Integer(125).exact_log(5)
3
sage: Integer(124).exact_log(5)
2
sage: Integer(126).exact_log(5)
3
sage: Integer(3).exact_log(5)
0
sage: Integer(1).exact_log(5)
0
sage: Integer(178^1700).exact_log(178)
1700
sage: Integer(178^1700-1).exact_log(178)
1699
sage: Integer(178^1700+1).exact_log(178)
1700
sage: # we need to exercise the large base code path too
sage: Integer(1780^1700-1).exact_log(1780)
1699

sage: # The following are very very fast.
sage: # Note that for base m a perfect power of 2, we get the exact log by counting bits.
sage: n=29835798237501857013751098 35; m=32
sage: n.exact_log(m)
18
sage: # The next is a favorite of mine.  The log2 approximate is exact and immediately prova
sage: n=90153710570912709517902579010793251709257901270941709247901209742124;m=213509721309
sage: n.exact_log(m)
4
```

```
sage: x = 3^100000
sage: RR(log(RR(x), 3))
100000.000000000
sage: RR(log(RR(x + 100000), 3))
100000.000000000

sage: x.exact_log(3)
100000
sage: (x+1).exact_log(3)
100000
sage: (x-1).exact_log(3)
99999

sage: x.exact_log(2.5)
Traceback (most recent call last):
...
TypeError: Attempt to coerce non-integral RealNumber to Integer
```

exp(*prec=None*)

Returns the exponential function of self as a real number.

This function is provided only so that Sage integers may be treated in the same manner as real numbers when convenient.

INPUT:

- `prec` - integer (default: None): if None, returns symbolic, else to given bits of precision as in RealField

EXAMPLES:

```
sage: Integer(8).exp()
e^8
sage: Integer(8).exp(prec=100)
2980.9579870417282747435920995
sage: exp(Integer(8))
e^8
```

For even fairly large numbers, this may not be useful.

```
sage: y=Integer(145^145)
sage: y.exp()
e^25024207011349079210459585279553675697932183658421565260323592409432707306554163224876110
sage: y.exp(prec=53) # default RealField precision
+infinity
```

factor(*algorithm='pari'*, *proof=None*, *limit=None*, *int_=False*, *verbose=0*)

Return the prime factorization of this integer as a formal Factorization object.

INPUT:

- `algorithm` - string
  - `'pari'` - (default) use the PARI library
  - `'kash'` - use the KASH computer algebra system (requires the optional kash package)
  - `'magma'` - use the MAGMA computer algebra system (requires an installation of MAGMA)
  - `'qsieve'` - use Bill Hart's quadratic sieve code; WARNING: this may not work as expected, see qsieve? for more information

– 'ecm' - use ECM-GMP, an implementation of Hendrik Lenstra's elliptic curve method.

- **proof - bool (default: True) whether or not to prove** primality of each factor (only applicable for 'pari' and 'ecm').
- limit - int or None (default: None) if limit is given it must fit in a signed int, and the factorization is done using trial division and primes up to limit.

OUTPUT:

- a Factorization object containing the prime factors and their multiplicities

EXAMPLES:

```
sage: n = 2^100 - 1; n.factor()
3 * 5^3 * 11 * 31 * 41 * 101 * 251 * 601 * 1801 * 4051 * 8101 * 268501
```

This factorization can be converted into a list of pairs (p, e), where p is prime and e is a positive integer. Each pair can also be accessed directly by its index (ordered by increasing size of the prime):

```
sage: f = 60.factor()
sage: list(f)
[(2, 2), (3, 1), (5, 1)]
sage: f[2]
(5, 1)
```

Similarly, the factorization can be converted to a dictionary so the exponent can be extracted for each prime:

```
sage: f = (3^6).factor()
sage: dict(f)
{3: 6}
sage: dict(f)[3]
6
```

We use proof=False, which doesn't prove correctness of the primes that appear in the factorization:

```
sage: n = 92038409284239042384829034820394809238408234908 2
sage: n.factor(proof=False)
2 * 11 * 1531 * 4402903 * 10023679 * 619162955472170540533894518173
sage: n.factor(proof=True)
2 * 11 * 1531 * 4402903 * 10023679 * 619162955472170540533894518173
```

We factor using trial division only:

```
sage: n.factor(limit=1000)
2 * 11 * 41835640583745019265831379463815822381094652231
```

We factor using a quadratic sieve algorithm:

```
sage: p = next_prime(10^20)
sage: q = next_prime(10^21)
sage: n = p*q
sage: n.factor(algorithm='qsieve')
doctest:... RuntimeWarning: the factorization returned
by qsieve may be incomplete (the factors may not be prime)
or even wrong; see qsieve? for details
1000000000000000039 * 10000000000000000117
```

We factor using the elliptic curve method:

```
sage: p = next_prime(10^15)
sage: q = next_prime(10^21)
```

```
sage: n = p*q
sage: n.factor(algorithm='ecm')
1000000000000037 * 10000000000000000117
```

TESTS:

```
sage: n.factor(algorithm='foobar')
Traceback (most recent call last):
...
ValueError: Algorithm is not known
```

factorial()

Return the factorial $n! = 1 \cdot 2 \cdot 3 \cdots n$.

If the input does not fit in an `unsigned long int` a symbolic expression is returned.

EXAMPLES:

```
sage: for n in srange(7):
...       print n, n.factorial()
0 1
1 1
2 2
3 6
4 24
5 120
6 720
sage: 234234209384023842034.factorial()
factorial(234234209384023842034)
```

floor()

Return the floor of self, which is just self since self is an integer.

EXAMPLES:

```
sage: n = 6
sage: n.floor()
6
```

gamma()

The gamma function on integers is the factorial function (shifted by one) on positive integers, and $\pm\infty$ on non-positive integers.

EXAMPLES:

```
sage: gamma(5)
24
sage: gamma(0)
Infinity
sage: gamma(-1)
Infinity
sage: gamma(-2^150)
Infinity
```

gcd(n)

Return the greatest common divisor of self and n.

EXAMPLE:

```
sage: gcd(-1,1)
1
```

```
sage: gcd(0,1)
1
sage: gcd(0,0)
0
sage: gcd(2,2^6)
2
sage: gcd(21,2^6)
1
```

global_height(*prec=None*)

Returns the absolute logarithmic height of this rational integer.

INPUT:

- `prec` (int) – desired floating point precision (default: default RealField precision).

OUTPUT:

(real) The absolute logarithmic height of this rational integer.

ALGORITHM:

The height of the integer n is $\log |n|$.

EXAMPLES:

```
sage: ZZ(5).global_height()
1.60943791243410
sage: ZZ(-2).global_height(prec=100)
0.69314718055994530941723212146
sage: exp(_)
2.0000000000000000000000000000
```

imag()

Returns the imaginary part of self, which is zero.

EXAMPLES:

```
sage: Integer(9).imag()
0
```

inverse_mod(*n*)

Returns the inverse of self modulo n, if this inverse exists. Otherwise, raises a `ZeroDivisionError` exception.

INPUT:

- `self` - Integer
- `n` - Integer, or ideal of integer ring

OUTPUT:

- `x` - Integer such that x*self = 1 (mod m), or raises ZeroDivisionError.

IMPLEMENTATION:

Call the mpz_invert GMP library function.

EXAMPLES:

```
sage: a = Integer(189)
sage: a.inverse_mod(10000)
4709
sage: a.inverse_mod(-10000)
```

```
4709
sage: a.inverse_mod(1890)
Traceback (most recent call last):
...
ZeroDivisionError: Inverse does not exist.
sage: a = Integer(19)**100000
sage: b = a*a
sage: c = a.inverse_mod(b)
Traceback (most recent call last):
...
ZeroDivisionError: Inverse does not exist.
```

We check that #10625 is fixed:

```
sage: ZZ(2).inverse_mod(ZZ.ideal(3))
2
```

We check that #9955 is fixed:

```
sage: Rational(3)%Rational(-1)
0
```

inverse_of_unit()

Return inverse of self if self is a unit in the integers, i.e., self is -1 or 1. Otherwise, raise a ZeroDivisionError.

EXAMPLES:

```
sage: (1).inverse_of_unit()
1
sage: (-1).inverse_of_unit()
-1
sage: 5.inverse_of_unit()
Traceback (most recent call last):
...
ZeroDivisionError: Inverse does not exist.
sage: 0.inverse_of_unit()
Traceback (most recent call last):
...
ZeroDivisionError: Inverse does not exist.
```

is_integer()

Returns `True` as they are integers

EXAMPLES:

```
sage: sqrt(4).is_integer()
True
```

is_integral()

Return `True` since integers are integral, i.e., satisfy a monic polynomial with integer coefficients.

EXAMPLES:

```
sage: Integer(3).is_integral()
True
```

is_irreducible()

Returns `True` if self is irreducible, i.e. +/- prime

EXAMPLES:

```
sage: z = 2^31 - 1
sage: z.is_irreducible()
True
sage: z = 2^31
sage: z.is_irreducible()
False
sage: z = 7
sage: z.is_irreducible()
True
sage: z = -7
sage: z.is_irreducible()
True
```

is_norm(*K*, *element=False*, *proof=True*)

See `QQ(self).is_norm()`.

EXAMPLES:

```
sage: K = NumberField(x^2 - 2, 'beta')
sage: n = 4
sage: n.is_norm(K)
True
sage: 5.is_norm(K)
False
sage: 7.is_norm(QQ)
True
sage: n.is_norm(K, element=True)
(True, -4*beta + 6)
sage: n.is_norm(K, element=True)[1].norm()
4
sage: n = 5
sage: n.is_norm(K, element=True)
(False, None)
sage: n = 7
sage: n.is_norm(QQ, element=True)
(True, 7)
```

is_one()

Returns `True` if the integer is 1, otherwise `False`.

EXAMPLES:

```
sage: Integer(1).is_one()
True
sage: Integer(0).is_one()
False
```

is_perfect_power()

Returns `True` if `self` is a perfect power, ie if there exist integers a and b, $b > 1$ with `self` $= a^b$.

See also:

- `perfect_power()`: Finds the minimal base for which this integer is a perfect power.
- `is_power_of()`: If you know the base already this method is the fastest option.
- `is_prime_power()`: Checks whether the base is prime.

EXAMPLES:

```
sage: Integer(-27).is_perfect_power()
True
sage: Integer(12).is_perfect_power()
False

sage: z = 8
sage: z.is_perfect_power()
True
sage: 144.is_perfect_power()
True
sage: 10.is_perfect_power()
False
sage: (-8).is_perfect_power()
True
sage: (-4).is_perfect_power()
False
```

TESTS:

This is a test to make sure we work around a bug in GMP, see trac ticket #4612.

```
sage: [ -a for a in srange(100) if not (-a^3).is_perfect_power() ]
[]
```

is_power_of(*n*)

Returns `True` if there is an integer b with `self` $= n^b$.

See also:

- `perfect_power()`: Finds the minimal base for which this integer is a perfect power.
- `is_perfect_power()`: If you don't know the base but just want to know if this integer is a perfect power, use this function.
- `is_prime_power()`: Checks whether the base is prime.

EXAMPLES:

```
sage: Integer(64).is_power_of(4)
True
sage: Integer(64).is_power_of(16)
False
```

TESTS:

```
sage: Integer(-64).is_power_of(-4)
True
sage: Integer(-32).is_power_of(-2)
True
sage: Integer(1).is_power_of(1)
True
sage: Integer(-1).is_power_of(-1)
True
sage: Integer(0).is_power_of(1)
False
sage: Integer(0).is_power_of(0)
True
sage: Integer(1).is_power_of(0)
True
sage: Integer(1).is_power_of(8)
```

```
True
sage: Integer(-8).is_power_of(2)
False
sage: Integer(-81).is_power_of(-3)
False
```

Note: For large integers self, is_power_of() is faster than is_perfect_power(). The following examples gives some indication of how much faster.

```
sage: b = lcm(range(1,10000))
sage: b.exact_log(2)
14446
sage: t=cputime()
sage: for a in range(2, 1000): k = b.is_perfect_power()
sage: cputime(t)      # random
0.53203299999999976
sage: t=cputime()
sage: for a in range(2, 1000): k = b.is_power_of(2)
sage: cputime(t)      # random
0.0
sage: t=cputime()
sage: for a in range(2, 1000): k = b.is_power_of(3)
sage: cputime(t)      # random
0.032002000000000308

sage: b = lcm(range(1, 1000))
sage: b.exact_log(2)
1437
sage: t=cputime()
sage: for a in range(2, 10000): k = b.is_perfect_power() # note that we change the range fro
sage: cputime(t)      # random
0.17201100000000036
sage: t=cputime(); TWO=int(2)
sage: for a in range(2, 10000): k = b.is_power_of(TWO)
sage: cputime(t)      # random
0.0040000000000000036
sage: t=cputime()
sage: for a in range(2, 10000): k = b.is_power_of(3)
sage: cputime(t)      # random
0.040003000000000011
sage: t=cputime()
sage: for a in range(2, 10000): k = b.is_power_of(a)
sage: cputime(t)      # random
0.02800199999999986
```

is_prime (*proof=None*)

Test whether `self` is prime.

INPUT:

- `proof` – Boolean or `None` (default). If False, use a strong pseudo-primality test (see `is_pseudoprime()`). If True, use a provable primality test. If unset, use the `default arithmetic proof flag`.

Note: Integer primes are by definition *positive*! This is different than Magma, but the same as in PARI. See also the `is_irreducible()` method.

EXAMPLES:

```
sage: z = 2^31 - 1
sage: z.is_prime()
True
sage: z = 2^31
sage: z.is_prime()
False
sage: z = 7
sage: z.is_prime()
True
sage: z = -7
sage: z.is_prime()
False
sage: z.is_irreducible()
True

sage: z = 10^80 + 129
sage: z.is_prime(proof=False)
True
sage: z.is_prime(proof=True)
True
```

When starting Sage the arithmetic proof flag is True. We can change it to False as follows:

```
sage: proof.arithmetic()
True
sage: n = 10^100 + 267
sage: timeit("n.is_prime()")  # not tested
5 loops, best of 3: 163 ms per loop
sage: proof.arithmetic(False)
sage: proof.arithmetic()
False
sage: timeit("n.is_prime()")  # not tested
1000 loops, best of 3: 573 us per loop
```

ALGORITHM:

Calls the PARI `isprime` function.

is_prime_power (*flag=None*, *proof=None*, *get_data=False*)

Return `True` if this integer is a prime power, and `False` otherwise.

A prime power is a prime number raised to a positive power. Hence 1 is not a prime power.

For a method that uses a pseudoprimality test instead see `is_pseudoprime_power()`.

INPUT:

- `proof` – Boolean or `None` (default). If `False`, use a strong pseudo-primality test (see `is_pseudoprime()`). If `True`, use a provable primality test. If unset, use the default arithmetic proof flag.
- `get_data` – (default `False`), if `True` return a pair `(p,k)` such that this integer equals `p^k` with `p` a prime and `k` a positive integer or the pair `(self,0)` otherwise.

See also:

- `perfect_power()`: Finds the minimal base for which integer is a perfect power.
- `is_perfect_power()`: Doesn't test whether the base is prime.
- `is_power_of()`: If you know the base already this method is the fastest option.

- is_pseudoprime_power(): If the entry is very large.

EXAMPLES:

```
sage: 17.is_prime_power()
True
sage: 10.is_prime_power()
False
sage: 64.is_prime_power()
True
sage: (3^10000).is_prime_power()
True
sage: (10000).is_prime_power()
False
sage: (-3).is_prime_power()
False
sage: 0.is_prime_power()
False
sage: 1.is_prime_power()
False
sage: p = next_prime(10^20); p
100000000000000000039
sage: p.is_prime_power()
True
sage: (p^97).is_prime_power()
True
sage: (p+1).is_prime_power()
False
```

With the get_data keyword set to True:

```
sage: (3^100).is_prime_power(get_data=True)
(3, 100)
sage: 12.is_prime_power(get_data=True)
(12, 0)
sage: (p^97).is_prime_power(get_data=True)
(100000000000000000039, 97)
sage: q = p.next_prime(); q
100000000000000000129
sage: (p*q).is_prime_power(get_data=True)
(10000000000000000016800000000000000005031, 0)
```

The method works for large entries when $proof = False$:

```
sage: proof.arithmetic(False)
sage: ((10^500 + 961)^4).is_prime_power()
True
sage: proof.arithmetic(True)
```

We check that trac ticket #4777 is fixed:

```
sage: n = 150607571^14
sage: n.is_prime_power()
True
```

is_pseudoprime()

Test whether self is a pseudoprime

This uses PARI's Baillie-PSW probabilistic primality test. Currently, there are no known pseudoprimes for Baille-PSW that are not actually prime. However it is conjectured that there are infinitely many.

EXAMPLES:

```
sage: z = 2^31 - 1
sage: z.is_pseudoprime()
True
sage: z = 2^31
sage: z.is_pseudoprime()
False
```

is_pseudoprime_power(*get_data=False*)

Test if this number is a power of a pseudoprime number.

For large numbers, this method might be faster than `is_prime_power()`.

INPUT:

- `get_data` – (default `False`) if `True` return a pair (p, k) such that this number equals p^k with p a pseudoprime and k a positive integer or the pair `(self,0)` otherwise.

EXAMPLES:

```
sage: x = 10^200 + 357
sage: x.is_pseudoprime()
True
sage: (x^12).is_pseudoprime_power()
True
sage: (x^12).is_pseudoprime_power(get_data=True)
(1000...000357, 12)
sage: (997^100).is_pseudoprime_power()
True
sage: (998^100).is_pseudoprime_power()
False
sage: ((10^1000 + 453)^2).is_pseudoprime_power()
True
```

TESTS:

```
sage: 0.is_pseudoprime_power()
False
sage: (-1).is_pseudoprime_power()
False
sage: 1.is_pseudoprime_power()
False
```

is_square()

Returns `True` if self is a perfect square.

EXAMPLES:

```
sage: Integer(4).is_square()
True
sage: Integer(41).is_square()
False
```

is_squarefree()

Returns True if this integer is not divisible by the square of any prime and False otherwise.

EXAMPLES:

```
sage: 100.is_squarefree()
False
sage: 102.is_squarefree()
True
```

```
sage: 0.is_squarefree()
False
```

is_unit()

Returns `true` if this integer is a unit, i.e., 1 or -1.

EXAMPLES:

```
sage: for n in srange(-2,3):
...    print n, n.is_unit()
-2 False
-1 True
0 False
1 True
2 False
```

isqrt()

Returns the integer floor of the square root of self, or raises an `ValueError` if self is negative.

EXAMPLE:

```
sage: a = Integer(5)
sage: a.isqrt()
2

sage: Integer(-102).isqrt()
Traceback (most recent call last):
...
ValueError: square root of negative integer not defined.
```

jacobi(b)

Calculate the Jacobi symbol $\left(\frac{self}{b}\right)$.

EXAMPLES:

```
sage: z = -1
sage: z.jacobi(17)
1
sage: z.jacobi(19)
-1
sage: z.jacobi(17*19)
-1
sage: (2).jacobi(17)
1
sage: (3).jacobi(19)
-1
sage: (6).jacobi(17*19)
-1
sage: (6).jacobi(33)
0
sage: a = 3; b = 7
sage: a.jacobi(b) == -b.jacobi(a)
True
```

kronecker(b)

Calculate the Kronecker symbol $\left(\frac{self}{b}\right)$ with the Kronecker extension $(self/2) = (2/self)$ when $self$ is odd, or $(self/2) = 0$ when $self$ is even.

EXAMPLES:

```
sage: z = 5
sage: z.kronecker(41)
1
sage: z.kronecker(43)
-1
sage: z.kronecker(8)
-1
sage: z.kronecker(15)
0
sage: a = 2; b = 5
sage: a.kronecker(b) == b.kronecker(a)
True
```

list()

Return a list with this integer in it, to be compatible with the method for number fields.

EXAMPLES:

```
sage: m = 5
sage: m.list()
[5]
```

log(*m=None*, *prec=None*)

Returns symbolic log by default, unless the logarithm is exact (for an integer base). When precision is given, the RealField approximation to that bit precision is used.

This function is provided primarily so that Sage integers may be treated in the same manner as real numbers when convenient. Direct use of exact_log is probably best for arithmetic log computation.

INPUT:

- `m` - default: natural log base e
- `prec` - integer (default: None): if None, returns symbolic, else to given bits of precision as in RealField

EXAMPLES:

```
sage: Integer(124).log(5)
log(124)/log(5)
sage: Integer(124).log(5,100)
2.9950093311241087454822446806
sage: Integer(125).log(5)
3
sage: Integer(125).log(5,prec=53)
3.00000000000000
sage: log(Integer(125))
log(125)
```

For extremely large numbers, this works:

```
sage: x = 3^100000
sage: log(x,3)
100000
```

With the new Pynac symbolic backend, log(x) also works in a reasonable amount of time for this x:

```
sage: x = 3^100000
sage: log(x)
log(1334971414230...5522000001)
```

But approximations are probably more useful in this case, and work to as high a precision as we desire:

```
sage: x.log(3,53) # default precision for RealField
100000.000000000
sage: (x+1).log(3,53)
100000.000000000
sage: (x+1).log(3,1000)
100000.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

We can use non-integer bases, with default e:

```
sage: x.log(2.5,prec=53)
119897.784671579
```

We also get logarithms of negative integers, via the symbolic ring, using the branch from $-pi$ to pi:

```
sage: log(-1)
I*pi
```

The logarithm of zero is done likewise:

```
sage: log(0)
-Infinity
```

multifactorial(k)

Computes the k-th factorial $n!^{(k)}$ of self. For k=1 this is the standard factorial, and for k greater than one it is the product of every k-th terms down from self to k. The recursive definition is used to extend this function to the negative integers.

EXAMPLES:

```
sage: 5.multifactorial(1)
120
sage: 5.multifactorial(2)
15
sage: 23.multifactorial(2)
316234143225
sage: prod([1..23, step=2])
316234143225
sage: (-29).multifactorial(7)
1/2640
```

multiplicative_order()

Return the multiplicative order of self.

EXAMPLES:

```
sage: ZZ(1).multiplicative_order()
1
sage: ZZ(-1).multiplicative_order()
2
sage: ZZ(0).multiplicative_order()
+Infinity
sage: ZZ(2).multiplicative_order()
+Infinity
```

nbits()

Return the number of bits in self.

EXAMPLES:

```
sage: 500.nbits()
9
sage: 5.nbits()
3
sage: 0.nbits() == len(0.bits()) == 0.ndigits(base=2)
True
sage: 12345.nbits() == len(12345.binary())
True
```

ndigits (*base=10*)

Return the number of digits of self expressed in the given base.

INPUT:

- `base` - integer (default: 10)

EXAMPLES:

```
sage: n = 52
sage: n.ndigits()
2
sage: n = -10003
sage: n.ndigits()
5
sage: n = 15
sage: n.ndigits(2)
4
sage: n = 1000**1000000+1
sage: n.ndigits()
3000001
sage: n = 1000**1000000-1
sage: n.ndigits()
3000000
sage: n = 10**10000000-10**9999990
sage: n.ndigits()
10000000
```

next_prime (*proof=None*)

Return the next prime after self.

This method calls the PARI `nextprime` function.

INPUT:

- `proof` - bool or None (default: None, see proof.arithmetic or sage.structure.proof) Note that the global Sage default is proof=True

EXAMPLES:

```
sage: 100.next_prime()
101
sage: (10^50).next_prime()
100000000000000000000000000000000000000000000000151
```

Use `proof=False`, which is way faster since it does not need a primality proof:

```
sage: b = (2^1024).next_prime(proof=False)
sage: b - 2^1024
643
```

```
sage: Integer(0).next_prime()
2
```

```
sage: Integer(1001).next_prime()
1009
```

next_prime_power(*proof=None*)

Return the next prime power after self.

INPUT:

- `proof` - if `True` ensure that the returned value is the next prime power and if set to `False` uses probabilistic methods (i.e. the result is not guaranteed). By default it uses global configuration variables to determine which alternative to use (see `proof.arithmetic` or `sage.structure.proof`).

ALGORITHM:

The algorithm is naive. It computes the next power of 2 and go through the odd numbers calling `is_prime_power()`.

See also:

- `previous_prime_power()`
- `is_prime_power()`
- `next_prime()`
- `previous_prime()`

EXAMPLES:

```
sage: (-1).next_prime_power()
2
sage: 2.next_prime_power()
3
sage: 103.next_prime_power()
107
sage: 107.next_prime_power()
109
sage: 2044.next_prime_power()
2048
```

TESTS:

```
sage: [(2**k-1).next_prime_power() for k in range(1,10)]
[2, 4, 8, 16, 32, 64, 128, 256, 512]
sage: [(2**k).next_prime_power() for k in range(10)]
[2, 3, 5, 9, 17, 37, 67, 131, 257, 521]

sage: for _ in range(10):
....:     n = ZZ.random_element(2**256).next_prime_power()
....:     m = n.next_prime_power().previous_prime_power()
....:     assert m == n, "problem with n = {}".format(n)
```

next_probable_prime()

Returns the next probable prime after self, as determined by PARI.

EXAMPLES:

```
sage: (-37).next_probable_prime()
2
sage: (100).next_probable_prime()
101
sage: (2^512).next_probable_prime()
```

```
1340780792994259709957402499820584612747936582059239337772356144372176403007354697680187429
sage: 0.next_probable_prime()
2
sage: 126.next_probable_prime()
127
sage: 144168.next_probable_prime()
144169
```

nth_root (*n*, *truncate_mode=0*)

Returns the (possibly truncated) n'th root of self.

INPUT:

- n - integer >= 1 (must fit in C int type).
- truncate_mode - boolean, whether to allow truncation if self is not an n'th power.

OUTPUT:

If truncate_mode is 0 (default), then returns the exact n'th root if self is an n'th power, or raises a ValueError if it is not.

If truncate_mode is 1, then if either n is odd or self is positive, returns a pair (root, exact_flag) where root is the truncated nth root (rounded towards zero) and exact_flag is a boolean indicating whether the root extraction was exact; otherwise raises a ValueError.

AUTHORS:

- David Harvey (2006-09-15)
- Interface changed by John Cremona (2009-04-04)

EXAMPLES:

```
sage: Integer(125).nth_root(3)
5
sage: Integer(124).nth_root(3)
Traceback (most recent call last):
...
ValueError: 124 is not a 3rd power
sage: Integer(124).nth_root(3, truncate_mode=1)
(4, False)
sage: Integer(125).nth_root(3, truncate_mode=1)
(5, True)
sage: Integer(126).nth_root(3, truncate_mode=1)
(5, False)

sage: Integer(-125).nth_root(3)
-5
sage: Integer(-125).nth_root(3,truncate_mode=1)
(-5, True)
sage: Integer(-124).nth_root(3,truncate_mode=1)
(-4, False)
sage: Integer(-126).nth_root(3,truncate_mode=1)
(-5, False)

sage: Integer(125).nth_root(2, True)
(11, False)
sage: Integer(125).nth_root(3, True)
(5, True)
```

```
sage: Integer(125).nth_root(-5)
Traceback (most recent call last):
...
ValueError: n (=-5) must be positive

sage: Integer(-25).nth_root(2)
Traceback (most recent call last):
...
ValueError: cannot take even root of negative number

sage: a=9
sage: a.nth_root(3)
Traceback (most recent call last):
...
ValueError: 9 is not a 3rd power

sage: a.nth_root(22)
Traceback (most recent call last):
...
ValueError: 9 is not a 22nd power

sage: ZZ(2^20).nth_root(21)
Traceback (most recent call last):
...
ValueError: 1048576 is not a 21st power

sage: ZZ(2^20).nth_root(21, truncate_mode=1)
(1, False)
```

numerator()

Return the numerator of this integer.

EXAMPLES:

```
sage: x = 5
sage: x.numerator()
5

sage: x = 0
sage: x.numerator()
0
```

odd_part()

The odd part of the integer n. This is $n/2^v$, where $v = \text{valuation}(n, 2)$.

IMPLEMENTATION:

Currently returns 0 when self is 0. This behaviour is fairly arbitrary, and in Sage 4.6 this special case was not handled at all, eventually propagating a TypeError. The caller should not rely on the behaviour in case self is 0.

EXAMPLES:

```
sage: odd_part(5)
5
sage: odd_part(4)
1
sage: odd_part(factorial(31))
122529844256906551386796875
```

ord(*p*)
Return the p-adic valuation of self.

INPUT:

- p - an integer at least 2.

EXAMPLE:

```
sage: n = 60
sage: n.valuation(2)
2
sage: n.valuation(3)
1
sage: n.valuation(7)
0
sage: n.valuation(1)
Traceback (most recent call last):
...
ValueError: You can only compute the valuation with respect to a integer larger than 1.
```

We do not require that p is a prime:

```
sage: (2^11).valuation(4)
5
```

ordinal_str()
Returns a string representation of the ordinal associated to self.

EXAMPLES:

```
sage: [ZZ(n).ordinal_str() for n in range(25)]
['0th',
'1st',
'2nd',
'3rd',
'4th',
...
'10th',
'11th',
'12th',
'13th',
'14th',
...
'20th',
'21st',
'22nd',
'23rd',
'24th']

sage: ZZ(1001).ordinal_str()
'1001st'

sage: ZZ(113).ordinal_str()
'113th'
sage: ZZ(112).ordinal_str()
'112th'
sage: ZZ(111).ordinal_str()
'111th'
```

perfect_power()

Returns `(a, b)`, where this integer is a^b and b is maximal.

If called on -1, 0 or 1, b will be 1, since there is no maximal value of b.

See also:

- `is_perfect_power()`: testing whether an integer is a perfect power is usually faster than finding a and b.
- `is_prime_power()`: checks whether the base is prime.
- `is_power_of()`: if you know the base already, this method is the fastest option.

EXAMPLES:

```
sage: 144.perfect_power()
(12, 2)
sage: 1.perfect_power()
(1, 1)
sage: 0.perfect_power()
(0, 1)
sage: (-1).perfect_power()
(-1, 1)
sage: (-8).perfect_power()
(-2, 3)
sage: (-4).perfect_power()
(-4, 1)
sage: (101^29).perfect_power()
(101, 29)
sage: (-243).perfect_power()
(-3, 5)
sage: (-64).perfect_power()
(-4, 3)
```

popcount()

Return the number of 1 bits in the binary representation. If self<0, we return Infinity.

EXAMPLES:

```
sage: n = 123
sage: n.str(2)
'1111011'
sage: n.popcount()
6

sage: n = -17
sage: n.popcount()
+Infinity
```

powermod(*exp*, *mod*)

Compute self**exp modulo mod.

EXAMPLES:

```
sage: z = 2
sage: z.powermod(31,31)
2
sage: z.powermod(0,31)
1
sage: z.powermod(-31,31) == 2^-31 % 31
True
```

As expected, the following is invalid:

```
sage: z.powermod(31,0)
Traceback (most recent call last):
...
ZeroDivisionError: cannot raise to a power modulo 0
```

powermodm_ui(**args*, ***kwds*)

Deprecated: Use `powermod()` instead. See trac ticket #17852 for details.

previous_prime(*proof=None*)

Returns the previous prime before self.

This method calls the PARI `precprime` function.

INPUT:

- `proof` - if `True` ensure that the returned value is the next prime power and if set to `False` uses probabilistic methods (i.e. the result is not guaranteed). By default it uses global configuration variables to determine which alternative to use (see `proof.arithmetic` or `sage.structure.proof`).

EXAMPLES:

```
sage: 10.previous_prime()
7
sage: 7.previous_prime()
5
sage: 14376485.previous_prime()
14376463

sage: 2.previous_prime()
Traceback (most recent call last):
...
ValueError: no prime less than 2
```

An example using `proof=False`, which is way faster since it does not need a primality proof:

```
sage: b = (2^1024).previous_prime(proof=False)
sage: 2^1024 - b
105
```

previous_prime_power(*proof=None*)

Return the previous prime power before self.

INPUT:

- `proof` - if `True` ensure that the returned value is the next prime power and if set to `False` uses probabilistic methods (i.e. the result is not guaranteed). By default it uses global configuration variables to determine which alternative to use (see `proof.arithmetic` or `sage.structure.proof`).

ALGORITHM:

The algorithm is naive. It computes the previous power of 2 and go through the odd numbers calling the method `is_prime_power()`.

See also:

- `next_prime_power()`
- `is_prime_power()`
- `previous_prime()`
- `next_prime()`

EXAMPLES:

```
sage: 3.previous_prime_power()
2
sage: 103.previous_prime_power()
101
sage: 107.previous_prime_power()
103
sage: 2044.previous_prime_power()
2039

sage: 2.previous_prime_power()
Traceback (most recent call last):
...
ValueError: no prime power less than 2
```

TESTS:

```
sage: [(2**k+1).previous_prime_power() for k in range(1,10)]
[2, 4, 8, 16, 32, 64, 128, 256, 512]
sage: [(2**k).previous_prime_power() for k in range(2, 10)]
[3, 7, 13, 31, 61, 127, 251, 509]

sage: for _ in range(10):
....:     n = ZZ.random_element(3,2**256).previous_prime_power()
....:     m = n.previous_prime_power().next_prime_power()
....:     assert m == n, "problem with n = {}".format(n)
```

prime_divisors()

The prime divisors of self, sorted in increasing order. If n is negative, we do *not* include -1 among the prime divisors, since -1 is not a prime number.

EXAMPLES:

```
sage: a = 1; a.prime_divisors()
[]
sage: a = 100; a.prime_divisors()
[2, 5]
sage: a = -100; a.prime_divisors()
[2, 5]
sage: a = 2004; a.prime_divisors()
[2, 3, 167]
```

prime_factors()

The prime divisors of self, sorted in increasing order. If n is negative, we do *not* include -1 among the prime divisors, since -1 is not a prime number.

EXAMPLES:

```
sage: a = 1; a.prime_divisors()
[]
sage: a = 100; a.prime_divisors()
[2, 5]
sage: a = -100; a.prime_divisors()
[2, 5]
sage: a = 2004; a.prime_divisors()
[2, 3, 167]
```

prime_to_m_part(*m*)

Returns the prime-to-m part of self, i.e., the largest divisor of `self` that is coprime to `m`.

INPUT:

- m - Integer

OUTPUT: Integer

EXAMPLES:

```
sage: 43434.prime_to_m_part(20)
21717
sage: 2048.prime_to_m_part(2)
1
sage: 2048.prime_to_m_part(3)
2048

sage: 0.prime_to_m_part(2)
Traceback (most recent call last):
...
ArithmeticError: self must be nonzero
```

quo_rem(*other*)

Returns the quotient and the remainder of self divided by other. Note that the remainder returned is always either zero or of the same sign as other.

INPUT:

- other - the divisor

OUTPUT:

- q - the quotient of self/other
- r - the remainder of self/other

EXAMPLES:

```
sage: z = Integer(231)
sage: z.quo_rem(2)
(115, 1)
sage: z.quo_rem(-2)
(-116, -1)
sage: z.quo_rem(0)
Traceback (most recent call last):
...
ZeroDivisionError: Integer division by zero

sage: a = ZZ.random_element(10**50)
sage: b = ZZ.random_element(10**15)
sage: q, r = a.quo_rem(b)
sage: q*b + r == a
True

sage: 3.quo_rem(ZZ['x'].0)
(0, 3)
```

TESTS:

The divisor can be rational as well, although the remainder will always be zero (trac ticket #7965):

```
sage: 5.quo_rem(QQ(2))
(5/2, 0)
sage: 5.quo_rem(2/3)
(15/2, 0)
```

radical(**args*, ***kwds*)

Return the product of the prime divisors of self. Computing the radical of zero gives an error.

EXAMPLES:

```
sage: Integer(10).radical()
10
sage: Integer(20).radical()
10
sage: Integer(-100).radical()
10
sage: Integer(0).radical()
Traceback (most recent call last):
...
ArithmeticError: Radical of 0 not defined.
```

rational_reconstruction(*m*)

Return the rational reconstruction of this integer modulo m, i.e., the unique (if it exists) rational number that reduces to self modulo m and whose numerator and denominator is bounded by sqrt(m/2).

INPUT:

- `self` – Integer
- `m` – Integer

OUTPUT:

- a `Rational`

EXAMPLES:

```
sage: (3/7)%100
29
sage: (29).rational_reconstruction(100)
3/7
```

TESTS:

Check that trac:9345 is fixed:

```
sage: 0.rational_reconstruction(0)
Traceback (most recent call last):
...
ZeroDivisionError: rational reconstruction with zero modulus
sage: ZZ.random_element(-10^6, 10^6).rational_reconstruction(0)
Traceback (most recent call last):
...
ZeroDivisionError: rational reconstruction with zero modulus
```

real()

Returns the real part of self, which is self.

EXAMPLES:

```
sage: Integer(-4).real()
-4
```

sign()

Returns the sign of this integer, which is -1, 0, or 1 depending on whether this number is negative, zero, or positive respectively.

OUTPUT: Integer

EXAMPLES:

```
sage: 500.sign()
1
sage: 0.sign()
0
sage: (-10^43).sign()
-1
```

sqrt (*prec=None*, *extend=True*, *all=False*)

The square root function.

INPUT:

- `prec` - integer (default: None): if None, returns an exact square root; otherwise returns a numerical square root if necessary, to the given bits of precision.
- `extend` - bool (default: True); if True, return a square root in an extension ring, if necessary. Otherwise, raise a ValueError if the square is not in the base ring. Ignored if prec is not None.
- `all` - bool (default: False); if True, return all square roots of self, instead of just one.

EXAMPLES:

```
sage: Integer(144).sqrt()
12
sage: sqrt(Integer(144))
12
sage: Integer(102).sqrt()
sqrt(102)

sage: n = 2
sage: n.sqrt(all=True)
[sqrt(2), -sqrt(2)]
sage: n.sqrt(prec=10)
1.4
sage: n.sqrt(prec=100)
1.4142135623730950488016887242
sage: n.sqrt(prec=100,all=True)
[1.4142135623730950488016887242, -1.4142135623730950488016887242]
sage: n.sqrt(extend=False)
Traceback (most recent call last):
...
ValueError: square root of 2 not an integer
sage: Integer(144).sqrt(all=True)
[12, -12]
sage: Integer(0).sqrt(all=True)
[0]
sage: type(Integer(5).sqrt())
<type 'sage.symbolic.expression.Expression'>
sage: type(Integer(5).sqrt(prec=53))
<type 'sage.rings.real_mpfr.RealNumber'>
sage: type(Integer(-5).sqrt(prec=53))
<type 'sage.rings.complex_number.ComplexNumber'>
```

TESTS:

Check that trac ticket #9466 is fixed:

```
sage: 3.sqrt(extend=False, all=True)
[]
```

sqrtrem()

Return (s, r) where s is the integer square root of self and r is the remainder such that self $= s^2 + r$. Raises `ValueError` if self is negative.

EXAMPLES:

```
sage: 25.sqrtrem()
(5, 0)
sage: 27.sqrtrem()
(5, 2)
sage: 0.sqrtrem()
(0, 0)

sage: Integer(-102).sqrtrem()
Traceback (most recent call last):
...
ValueError: square root of negative integer not defined.
```

squarefree_part(*bound=-1*)

Return the square free part of x (=self), i.e., the unique integer z that $x = zy^2$, with y^2 a perfect square and z square-free.

Use `self.radical()` for the product of the primes that divide self.

If self is 0, just returns 0.

EXAMPLES:

```
sage: squarefree_part(100)
1
sage: squarefree_part(12)
3
sage: squarefree_part(17*37*37)
17
sage: squarefree_part(-17*32)
-34
sage: squarefree_part(1)
1
sage: squarefree_part(-1)
-1
sage: squarefree_part(-2)
-2
sage: squarefree_part(-4)
-1

sage: a = 8 * 5^6 * 101^2
sage: a.squarefree_part(bound=2).factor()
2 * 5^6 * 101^2
sage: a.squarefree_part(bound=5).factor()
2 * 101^2
sage: a.squarefree_part(bound=1000)
2
sage: a.squarefree_part(bound=2**14)
2
sage: a = 7^3 * next_prime(2^100)^2 * next_prime(2^200)
sage: a / a.squarefree_part(bound=1000)
49
```

str(*base=10*)

Return the string representation of `self` in the given base.

EXAMPLES:

```
sage: Integer(2^10).str(2)
'10000000000'
sage: Integer(2^10).str(17)
'394'

sage: two=Integer(2)
sage: two.str(1)
Traceback (most recent call last):
...
ValueError: base (=1) must be between 2 and 36

sage: two.str(37)
Traceback (most recent call last):
...
ValueError: base (=37) must be between 2 and 36

sage: big = 10^5000000
sage: s = big.str()        # long time (2s on sage.math, 2014)
sage: len(s)               # long time (depends on above defn of s)
5000001
sage: s[:10]               # long time (depends on above defn of s)
'1000000000'
```

support()

Return a sorted list of the primes dividing this integer.

OUTPUT: The sorted list of primes appearing in the factorization of this rational with positive exponent.

EXAMPLES:

```
sage: factorial(10).support()
[2, 3, 5, 7]
sage: (-999).support()
[3, 37]
```

Trying to find the support of 0 gives an arithmetic error:

```
sage: 0.support()
Traceback (most recent call last):
...
ArithmeticError: Support of 0 not defined.
```

test_bit(*index*)

Return the bit at `index`.

If the index is negative, returns 0.

Although internally a sign-magnitude representation is used for integers, this method pretends to use a two's complement representation. This is illustrated with a negative integer below.

EXAMPLES:

```
sage: w = 6
sage: w.str(2)
'110'
sage: w.test_bit(2)
1
sage: w.test_bit(-1)
0
sage: x = -20
```

```
sage: x.str(2)
'-10100'
sage: x.test_bit(4)
0
sage: x.test_bit(5)
1
sage: x.test_bit(6)
1
```

trailing_zero_bits()

Return the number of trailing zero bits in self, i.e. the exponent of the largest power of 2 dividing self.

EXAMPLES:

```
sage: 11.trailing_zero_bits()
0
sage: (-11).trailing_zero_bits()
0
sage: (11<<5).trailing_zero_bits()
5
sage: (-11<<5).trailing_zero_bits()
5
sage: 0.trailing_zero_bits()
0
```

trial_division(*bound='LONG_MAX'*, *start=2*)

Return smallest prime divisor of self up to bound, beginning checking at start, or abs(self) if no such divisor is found.

INPUT:

- bound – a positive integer that fits in a C signed long
- start – a positive integer that fits in a C signed long

OUTPUT:

- a positive integer

EXAMPLES:

```
sage: n = next_prime(10^6)*next_prime(10^7); n.trial_division()
1000003
sage: (-n).trial_division()
1000003
sage: n.trial_division(bound=100)
10000049000057
sage: n.trial_division(bound=-10)
Traceback (most recent call last):
...
ValueError: bound must be positive
sage: n.trial_division(bound=0)
Traceback (most recent call last):
...
ValueError: bound must be positive
sage: ZZ(0).trial_division()
Traceback (most recent call last):
...
ValueError: self must be nonzero

sage: n = next_prime(10^5) * next_prime(10^40); n.trial_division()
```

```
100003
sage: n.trial_division(bound=10^4)
10000300000000000000000000000000000000012100363
sage: (-n).trial_division(bound=10^4)
10000300000000000000000000000000000000012100363
sage: (-n).trial_division()
100003
sage: n = 2 * next_prime(10^40); n.trial_division()
2
sage: n = 3 * next_prime(10^40); n.trial_division()
3
sage: n = 5 * next_prime(10^40); n.trial_division()
5
sage: n = 2 * next_prime(10^4); n.trial_division()
2
sage: n = 3 * next_prime(10^4); n.trial_division()
3
sage: n = 5 * next_prime(10^4); n.trial_division()
5
```

You can specify a starting point:

```
sage: n = 3*5*101*103
sage: n.trial_division(start=50)
101
```

val_unit(*p*)

Returns a pair: the p-adic valuation of self, and the p-adic unit of self.

INPUT:

- `p` - an integer at least 2.

OUTPUT:

- `v_p(self)` - the p-adic valuation of `self`
- `u_p(self)` - `self` / $p^{v_p(\text{self})}$

EXAMPLE:

```
sage: n = 60
sage: n.val_unit(2)
(2, 15)
sage: n.val_unit(3)
(1, 20)
sage: n.val_unit(7)
(0, 60)
sage: (2^11).val_unit(4)
(5, 2)
sage: 0.val_unit(2)
(+Infinity, 1)
```

valuation(*p*)

Return the p-adic valuation of self.

INPUT:

- `p` - an integer at least 2.

EXAMPLE:

```
sage: n = 60
sage: n.valuation(2)
2
sage: n.valuation(3)
1
sage: n.valuation(7)
0
sage: n.valuation(1)
Traceback (most recent call last):
...
ValueError: You can only compute the valuation with respect to a integer larger than 1.
```

We do not require that p is a prime:

```
sage: (2^11).valuation(4)
5
```

xgcd(*n*)

Return the extended gcd of this element and `n`.

INPUT:

- `n` – an integer

OUTPUT:

A triple `(g, s, t)` such that `g` is the non-negative gcd of `self` and `n`, and `s` and `t` are cofactors satisfying the Bezout identity

$$g = s \cdot \text{self} + t \cdot n.$$

Note: There is no guarantee that the cofactors will be minimal. If you need the cofactors to be minimal use `_xgcd()`. Also, using `_xgcd()` directly might be faster in some cases, see trac ticket #13628.

EXAMPLES:

```
sage: 6.xgcd(4)
(2, 1, -1)
```

class sage.rings.integer.**IntegerWrapper**

Bases: sage.rings.integer.Integer

Rationale for the `IntegerWrapper` class:

With `Integers`, the allocation/deallocation function slots are hijacked with custom functions that stick already allocated `Integers` (with initialized `parent` and `mpz_t` fields) into a pool on "deallocation" and then pull them out whenever a new one is needed. Because `Integers` are so common, this is actually a significant savings. However , this does cause issues with subclassing a Python class directly from `Integer` (but that's ok for a Cython class).

As a workaround, one can instead derive a class from the intermediate class `IntegerWrapper`, which sets statically its alloc/dealloc methods to the *original* `Integer` alloc/dealloc methods, before they are swapped manually for the custom ones.

The constructor of `IntegerWrapper` further allows for specifying an alternative parent to `IntegerRing()`.

sage.rings.integer.**LCM_list**(*v*)

Return the LCM of a list v of integers. Elements of v are converted to Sage integers if they aren't already.

This function is used, e.g., by rings/arith.py

INPUT:

- v - list or tuple

OUTPUT: integer

EXAMPLES:

```
sage: from sage.rings.integer import LCM_list
sage: w = LCM_list([3,9,30]); w
90
sage: type(w)
<type 'sage.rings.integer.Integer'>
```

The inputs are converted to Sage integers.

```
sage: w = LCM_list([int(3), int(9), '30']); w
90
sage: type(w)
<type 'sage.rings.integer.Integer'>
```

sage.rings.integer.**free_integer_pool**()

class sage.rings.integer.**int_to_Z**

Bases: sage.categories.morphism.Morphism

Morphism from Python ints to Sage integers.

EXAMPLES:

```
sage: f = ZZ.coerce_map_from(int); type(f)
<type 'sage.rings.integer.int_to_Z'>
sage: f(5r)
5
sage: type(f(5r))
<type 'sage.rings.integer.Integer'>
sage: 1 + 2r
3
sage: type(1 + 2r)
<type 'sage.rings.integer.Integer'>
```

This is intented for internal use by the coercion system, to facilitate fast expressions mixing ints and more complex Python types. Note that (as with all morphisms) the input is forcably coerced to the domain int if it is not already of the correct type which may have undesirable results:

```
sage: f.domain()
Set of Python objects of type 'int'
sage: f(1/3)
0
sage: f(1.7)
1
sage: f("10")
10
```

A pool is used for small integers:

```
sage: f(10) is f(10)
True
sage: f(-2) is f(-2)
True
```

sage.rings.integer.**is_Integer**(*x*)

Return true if x is of the Sage integer type.

EXAMPLES:

```
sage: from sage.rings.integer import is_Integer
sage: is_Integer(2)
True
sage: is_Integer(2/1)
False
sage: is_Integer(int(2))
False
sage: is_Integer(long(2))
False
sage: is_Integer('5')
False
```

class sage.rings.integer.**long_to_Z**

Bases: sage.categories.morphism.Morphism

EXAMPLES:

```
sage: f = ZZ.coerce_map_from(long); f
Native morphism:
  From: Set of Python objects of type 'long'
  To:   Integer Ring
sage: f(1rL)
1
sage: f(-100000000000000000000001r)
-100000000000000000000001
```

sage.rings.integer.**make_integer**(*s*)

Create a Sage integer from the base-32 Python *string* s. This is used in unpickling integers.

EXAMPLES:

```
sage: from sage.rings.integer import make_integer
sage: make_integer('-29')
-73
sage: make_integer(29)
Traceback (most recent call last):
...
TypeError: expected string or Unicode object, sage.rings.integer.Integer found
```

CHAPTER

THREE

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

EXAMPLES:

```
sage: R = Integers(97)
sage: a = R(5)
sage: a**10000000000000000000000000000000000000000000000000000000000000000000000000000
61
```

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

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

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

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

AUTHORS:

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

class sage.rings.finite_rings.integer_mod_ring.**IntegerModFactory**
Bases: sage.structure.factory.UniqueFactory

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

INPUT:

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

---

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

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

EXAMPLES:

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

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

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

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

EXAMPLES:

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

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

EXAMPLES:

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

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

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

However, the mistaken assignment is not automatically corrected:

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

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

An integer mod ring is specified uniquely by its order.

EXAMPLES:

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

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

EXAMPLES:

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

get_object(*version*, *key*, *extra_args*)

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

Bases: sage.rings.quotient_ring.QuotientRing_generic

The ring of integers modulo N, with N composite.

INPUT:

- `order` – an integer

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

OUTPUT:

The ring of integers modulo N.

EXAMPLES:

First we compute with integers modulo 29.

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

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

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

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

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

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

Next we compute with the integers modulo 16.

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

Saving and loading:

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

Testing ideals and quotients:

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

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

cardinality()
Return the cardinality of this ring.

EXAMPLES:

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

characteristic()
EXAMPLES:

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

degree()
Return 1.

EXAMPLE:

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

extension(*poly*, *name=None*, *names=None*, *embedding=None*)
Return an algebraic extension of `self`. See `sage.rings.ring.CommutativeRing.extension()` for more information.

EXAMPLES:

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

factored_order()
EXAMPLES:

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

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

EXAMPLES:

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

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

EXAMPLES:

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

is_field(*proof=None*)

Return True precisely if the order is prime.

INPUT:

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

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

EXAMPLES:

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

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

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

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

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

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

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

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

is_integral_domain(*proof=None*)
Return `True` if and only if the order of `self` is prime.

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

TESTS:

Check that trac ticket #17453 is fixed:
```
sage: R = Zmod(5)
sage: R in IntegralDomains()
True
```

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

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

is_prime_field()
Return `True` if the order is prime.

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

is_unique_factorization_domain(*proof=None*)
Return `True` if and only if the order of `self` is prime.

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

krull_dimension()
Return the Krull dimension of `self`.

EXAMPLES:

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

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

EXAMPLES:

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

modulus()
Return the polynomial $x - 1$ over this ring.

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

EXAMPLES:

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

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

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

EXAMPLES:

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

multiplicative_group_is_cyclic()
Return `True` if the multiplicative group of this field is cyclic. This is the case exactly when the order is

less than 8, a power of an odd prime, or twice a power of an odd prime.

EXAMPLES:

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

We test that trac ticket #5250 is fixed:

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

multiplicative_subgroups()

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

EXAMPLES:

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

TESTS:

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

order()

Return the order of this ring.

EXAMPLES:

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

quadratic_nonresidue()

Return a quadratic non-residue in `self`.

EXAMPLES:

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

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

random_element(*bound=None*)

Return a random element of this ring.

INPUT:

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

EXAMPLES:

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

We test bound-option:

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

square_roots_of_one()

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

OUTPUT:

The square roots of 1 in self as a tuple.

EXAMPLES:

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

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

unit_gens(***kwds*)

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

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

OUTPUT:

A tuple containing the units of self.

EXAMPLES:

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

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

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

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

TESTS:

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

unit_group(*algorithm='sage'*)

Return the unit group of `self`.

INPUT:

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

OUTPUT:

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

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

EXAMPLES:

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

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

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

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

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

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

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

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

sage: A = Zmod(30.factorial())
sage: A.unit_group()
Multiplicative Abelian group isomorphic to C2 x C16777216 x C3188646 x C62500 x C2058 x C110
sage: A.unit_group(algorithm='pari')
Multiplicative Abelian group isomorphic to C204996473853050880000000 x C55440 x C12 x C12 x C
```

TESTS:

We test the cases where the unit group is trivial:

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

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

unit_group_exponent()

EXAMPLES:

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

```
6
```

unit_group_order()
Return the order of the unit group of this residue class ring.

EXAMPLES:

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

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

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

EXAMPLES:

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

sage.rings.finite_rings.integer_mod_ring.**is_IntegerModRing**(*x*)
Return `True` if `x` is an integer modulo ring.

EXAMPLES:

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

CHAPTER

FOUR

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 $2^{31}-1$.
- `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

TESTS:

```
sage: R = Integers(101^3)
sage: a = R(824362); b = R(205942)
sage: a * b
851127

sage: type(IntegerModRing(2^31-1).an_element())
<type 'sage.rings.finite_rings.integer_mod.IntegerMod_int64'>
sage: type(IntegerModRing(2^31).an_element())
<type 'sage.rings.finite_rings.integer_mod.IntegerMod_gmp'>
```

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

Bases: sage.rings.finite_rings.integer_mod.IntegerMod_hom

EXAMPLES:

We make sure it works for every type.

```
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]
[<type 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>, <type 'sage.rings.finite_rings.int
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.

class sage.rings.finite_rings.integer_mod.**IntegerMod_abstract**

Bases: sage.rings.finite_rings.element_base.FiniteRingElement

EXAMPLES:

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

additive_order()

Returns the additive order of self.

This is the same as `self.order()`.

EXAMPLES:

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

centerlift()

Lift `self` to an integer i such that $n/2 < i <= n/2$ (where n denotes the modulus).

EXAMPLES:

```
sage: Mod(0,5).centerlift()
0
sage: Mod(1,5).centerlift()
1
sage: Mod(2,5).centerlift()
2
sage: Mod(3,5).centerlift()
-2
sage: Mod(4,5).centerlift()
-1
sage: Mod(50,100).centerlift()
50
sage: Mod(51,100).centerlift()
-49
sage: Mod(-1,3^100).centerlift()
-1
```

charpoly(*var='x'*)

Returns the characteristic polynomial of this element.

EXAMPLES:

```
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:

```
sage: a = mod(3,5)
sage: b = mod(2,7)
sage: a.crt(b)
23

sage: a = mod(37,10^8)
sage: b = mod(9,3^8)
sage: a.crt(b)
125900000037

sage: b = mod(0,1)
sage: a.crt(b) == a
True
sage: a.crt(b).modulus()
100000000
```

TESTS:

```
sage: mod(0,1).crt(mod(4,2^127))
4
sage: mod(4,2^127).crt(mod(0,1))
4
sage: mod(4,2^30).crt(mod(0,1))
4
sage: mod(0,1).crt(mod(4,2^30))
4
sage: mod(0,1).crt(mod(4,2^15))
4
sage: mod(4,2^15).crt(mod(0,1))
4
```

AUTHORS:

- Robert Bradshaw

generalised_log()

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

..math:

```
\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 `self.parent().unit_gens()`.

EXAMPLES:

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

See also:

The method `log()`.

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

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())
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 $x \in \mathbf{Z}/n\mathbf{Z}$ 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:

```
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()
True
sage: mod(11,1009^2).is_primitive_root()
True
```

TESTS:

```
sage: for p in prime_range(3,12):
...       for k in range(1,4):
...           for even in [1,2]:
...               n = even*p^k
...               phin = euler_phi(n)
...               for _ in range(6):
...                   a = Zmod(n).random_element()
...                   if not a.is_unit(): continue
...                   if a.is_primitive_root().__xor__(a.multiplicative_order()==phin):
...                       print "mod(%s,%s) incorrect"%(a,n)
```

is_square()

EXAMPLES:

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

TESTS:

```
sage: Mod(1/25, 2^8).is_square()
True
sage: Mod(1/25, 2^40).is_square()
True

sage: for p,q,r in cartesian_product_iterator([[3,5],[11,13],[17,19]]): # long time
....:     for ep,eq,er in cartesian_product_iterator([[0,1,2,3],[0,1,2,3],[0,1,2,3]]):
....:         for e2 in [0, 1, 2, 3, 4]:
....:             n = p^ep * q^eq * r^er * 2^e2
....:             for _ in range(2):
....:                 a = Zmod(n).random_element()
....:                 if a.is_square().__xor__(a._pari_().issquare()):
....:                     print a, n
```

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

The case $p = 2$ is handled separately.

AUTHORS:

- Robert Bradshaw

is_unit()

log(*b=None*)

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: If the modulus is prime and b is a generator, this calls Pari's znlog function, which is rather fast. If not, it falls back on the generic discrete log implementation in sage.groups.generic.discrete_log().

EXAMPLES:

```
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:

```
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
```

Things that can go wrong. E.g., if the base is not a generator for the multiplicative group, or not even a unit.

```
sage: Mod(3, 7).log(Mod(2, 7))
Traceback (most recent call last):
...
ValueError: No discrete log of 3 found to base 2
sage: a = Mod(16, 100); b = Mod(4,100)
sage: a.log(b)
Traceback (most recent call last):
...
ZeroDivisionError: Inverse does not exist.
```

We check that #9205 is fixed:

```
sage: Mod(5,9).log(Mod(2, 9))
5
```

We test against a bug (side effect on PARI) fixed in #9438:

```
sage: R.<a, b> = QQ[]
sage: pari(b)
b
sage: GF(7)(5).log()
5
sage: pari(b)
b
```

AUTHORS:

- David Joyner and William Stein (2005-11)

•William Stein (2007-01-27): update to use PARI as requested by David Kohel.

•Simon King (2010-07-07): fix a side effect on PARI

minimal_polynomial(*var='x'*)
Returns the minimal polynomial of this element.

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

minpoly(*var='x'*)
Returns the minimal polynomial of this element.

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

modulus()
EXAMPLES:

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

multiplicative_order()
Returns the multiplicative order of self.

EXAMPLES:

```
sage: Mod(-1,5).multiplicative_order()
2
sage: Mod(1,5).multiplicative_order()
1
sage: Mod(0,5).multiplicative_order()
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:

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

AUTHORS:

•Craig Citro

nth_root(*n*, *extend=False*, *all=False*, *algorithm=None*, *cunningham=False*)
Returns an nth root of `self`.

INPUT:

•`n` - integer ≥ 1

•`extend` - bool (default: True); if True, return an nth root in an extension ring, if necessary. Otherwise, raise a ValueError if the root is not in the base ring. Warning: this option is not implemented!

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

•`algorithm` - string (default: None); The algorithm for the prime modulus case. CRT and p-adic log techniques are used to reduce to this case. 'Johnston' is the only currently supported option.

- `cunningham` - bool (default: `False`); In some cases, factorization of `n` is computed. If cunningham is set to `True`, the factorization of `n` is computed using trial division for all primes in the so called Cunningham table. Refer to sage.rings.factorint.factor_cunningham for more information. You need to install an optional package to use this method, this can be done with the following command line `sage -i cunningham_tables`

OUTPUT:

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

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

NOTES:

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

EXAMPLES:

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

```
ValueError
sage: mod(1,6).nth_root(0,all=True)
[1, 2, 3, 4, 5]
```

TESTS:

```
sage: for p in [1009,2003,10007,100003]:
...       K = GF(p)
...       for r in (p-1).divisors():
...           if r == 1: continue
...           x = K.random_element()
...           y = x^r
...           if y.nth_root(r)**r != y: raise RuntimeError
...           if (y^41).nth_root(41*r)**(41*r) != y^41: raise RuntimeError
...           if (y^307).nth_root(307*r)**(307*r) != y^307: raise RuntimeError

sage: for t in xrange(200):
...       n = randint(1,2^63)
...       K = Integers(n)
...       b = K.random_element()
...       e = randint(-2^62, 2^63)
...       try:
...           a = b.nth_root(e)
...           if a^e != b:
...               print n, b, e, a
...               raise NotImplementedError
...       except ValueError:
...           pass
```

We check that #13172 is resolved:

```
sage: mod(-1, 4489).nth_root(2, all=True)
[]
```

Check that the code path cunningham might be used:

```
sage: a = Mod(9,11)
sage: a.nth_root(2, False, True, 'Johnston', cunningham = True) # optional - cunningham
[3, 8]
```

ALGORITHMS:

- The default for prime modulus is currently an algorithm described in the following paper:

Johnston, Anna M. A generalized qth root algorithm. Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms. Baltimore, 1999: pp 929-930.

AUTHORS:

- David Roe (2010-2-13)

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())
<type 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>
```

rational_reconstruction()

Use rational reconstruction to try to find a lift of this element to the rational numbers.

EXAMPLES:

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

This method is also inherited by prime finite fields elements:

```
sage: k = GF(97)
sage: a = k(RationalField()('2/3'))
sage: a
33
sage: a.rational_reconstruction()
2/3
```

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

Returns 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` (in this module) for more algorithmic details.

EXAMPLES:

```
sage: mod(-1, 17).sqrt()
4
sage: mod(5, 389).sqrt()
86
sage: mod(7, 18).sqrt()
5
sage: a = mod(14, 5^60).sqrt()
sage: a*a
14
sage: mod(15, 389).sqrt(extend=False)
Traceback (most recent call last):
...
ValueError: self must be a square
sage: Mod(1/9, next_prime(2^40)).sqrt()^(-2)
9
sage: Mod(1/25, next_prime(2^90)).sqrt()^(-2)
25
```

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

We compute all square roots in several cases:

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

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

Modulo a power of 2:

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

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

Returns 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` (in this module) for more algorithmic details.

EXAMPLES:

```
sage: mod(-1, 17).sqrt()
4
sage: mod(5, 389).sqrt()
86
sage: mod(7, 18).sqrt()
5
sage: a = mod(14, 5^60).sqrt()
sage: a*a
14
sage: mod(15, 389).sqrt(extend=False)
Traceback (most recent call last):
...
ValueError: self must be a square
sage: Mod(1/9, next_prime(2^40)).sqrt()^(-2)
9
sage: Mod(1/25, next_prime(2^90)).sqrt()^(-2)
25

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

We compute all square roots in several cases:

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

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

Modulo a power of 2:

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

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

AUTHORS:

- Craig Citro

valuation(*p*)

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

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

EXAMPLES:

This example shows that the (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.
```

TESTS:

```
sage: R=ZZ.quo(12)
sage: a=R(2)
sage: b=R(4)
sage: a.valuation(2)
1
sage: b.valuation(2)
+Infinity
sage: ZZ.quo(1024)(16).valuation(4)
2
```

class sage.rings.finite_rings.integer_mod.**IntegerMod_gmp**

Bases: sage.rings.finite_rings.integer_mod.IntegerMod_abstract

Elements of $\mathbf{Z}/n\mathbf{Z}$ 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 $\mathbf{Z}/N\mathbf{Z}$ 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*17^8).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:

```
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:

```
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:

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

class sage.rings.finite_rings.integer_mod.**IntegerMod_hom**
Bases: sage.categories.morphism.Morphism

class sage.rings.finite_rings.integer_mod.**IntegerMod_int**
Bases: sage.rings.finite_rings.integer_mod.IntegerMod_abstract

Elements of $\mathbf{Z}/n\mathbf{Z}$ for n small enough to be operated on in 32 bits

AUTHORS:

- Robert Bradshaw (2006-08-24)

gcd(*other*)
Greatest common divisor

Returns the "smallest" generator in $\mathbf{Z}/N\mathbf{Z}$ of the ideal generated by `self` and `other`.

INPUT:

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

EXAMPLES:

```
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
```

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

EXAMPLES:

```
sage: mod(6,5).is_one()
True
```

```
sage: mod(0,5).is_one()
False
```

is_unit()

Return True iff this element is a unit

EXAMPLES:

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

lift()

Lift an integer modulo n to the integers.

EXAMPLES:

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

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

Returns 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` (in this module) for more algorithmic details.

EXAMPLES:

```
sage: mod(-1, 17).sqrt()
4
sage: mod(5, 389).sqrt()
86
sage: mod(7, 18).sqrt()
5
sage: a = mod(14, 5^60).sqrt()
sage: a*a
14
sage: mod(15, 389).sqrt(extend=False)
Traceback (most recent call last):
...
ValueError: self must be a square
sage: Mod(1/9, next_prime(2^40)).sqrt()^(-2)
9
sage: Mod(1/25, next_prime(2^90)).sqrt()^(-2)
25
```

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

We compute all square roots in several cases:

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

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

Modulo a power of 2:

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

class sage.rings.finite_rings.integer_mod.**IntegerMod_int64**

Bases: sage.rings.finite_rings.integer_mod.IntegerMod_abstract

Elements of $\mathbf{Z}/n\mathbf{Z}$ for n small enough to be operated on in 64 bits

AUTHORS:

•Robert Bradshaw (2006-09-14)

gcd(*other*)

Greatest common divisor

Returns the "smallest" generator in $\mathbf{Z}/N\mathbf{Z}$ 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)
<type 'sage.rings.finite_rings.integer_mod.IntegerMod_int64'>
sage: lift(a)
8943
sage: a.lift()
8943
```

class sage.rings.finite_rings.integer_mod.**IntegerMod_to_Integer**

Bases: sage.categories.map.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: sage.rings.finite_rings.integer_mod.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]
[<type 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>, <type 'sage.rings.finite_rings.int
sage: fs = [IntegerMod_to_IntegerMod(S, R) for R in Rs for S in Rs if S is not R and S.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]
```

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

Bases: sage.rings.finite_rings.integer_mod.IntegerMod_hom

Fast $\mathbf{Z} \to \mathbf{Z}/n\mathbf{Z}$ morphism.

EXAMPLES:

We make sure it works for every type.

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

section()

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

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

EXAMPLES:

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

You can also use the lowercase version:

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

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

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

class sage.rings.finite_rings.integer_mod.**NativeIntStruct**

Bases: object

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

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

precompute_table(*parent*, *inverses=True*)

Function to compute and cache all elements of this class.

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

EXAMPLES:

This is used by the `sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic` constructor:

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

Check that the inverse of 0 modulo 1 works, see trac ticket #13639:

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

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:

AUTHORS:

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

TESTS:

```
sage: from sage.rings.finite_rings.integer_mod import lucas
sage: p = randint(0,100000)
sage: q = randint(0,100000)
sage: n = randint(0,100)
sage: all([lucas(k,p,q,n)[0] == Mod(lucas_number2(k,p,q),n)
...        for k in Integers(20)])
True
sage: from sage.rings.finite_rings.integer_mod import lucas
sage: p = randint(0,100000)
sage: q = randint(0,100000)
sage: n = randint(0,100)
sage: k = randint(0,100)
sage: lucas(k,p,q,n) == [Mod(lucas_number2(k,p,q),n),Mod(q^(int(k/2)),n)]
True
```

EXAMPLES:

```
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:

AUTHORS:

- Robert Bradshaw

TESTS:

```
sage: from sage.rings.finite_rings.integer_mod import lucas_q1
sage: all([lucas_q1(k, a) == BinaryRecurrenceSequence(a, -1, 2, a)(k)
....:      for a in Integers(23)
....:      for k in range(13)])
True
```

sage.rings.finite_rings.integer_mod.**makeNativeIntStruct**(*z*)

Function to convert a Sage Integer into class NativeIntStruct.

Note: This function is only used for the unpickle override below.

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

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

EXAMPLES:

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

You can also use the lowercase version:

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

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

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

sage.rings.finite_rings.integer_mod.**slow_lucas**(*k*, *P*, *Q=1*)

Lucas function defined using the standard definition, for consistency testing. This is deprecated in trac ticket #11802. Use `BinaryRecurrenceSequence(P, -Q, 2, P)(k)` instead.

See also:

`BinaryRecurrenceSequence`

REFERENCES:

- Wikipedia article Lucas_sequence

TESTS:

```
sage: from sage.rings.finite_rings.integer_mod import slow_lucas
sage: [slow_lucas(k, 1, -1) for k in range(10)]
doctest:...: DeprecationWarning: slow_lucas() is deprecated. Use BinaryRecurrenceSequence instea
See http://trac.sagemath.org/11802 for details.
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
```

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 $p \bmod 16$.

- $p \bmod 2 = 0$: $p = 2$ so $\sqrt{a} = a$.
- $p \bmod 4 = 3$: $\sqrt{a} = a^{(p+1)/4}$.
- $p \bmod 8 = 5$: $\sqrt{a} = \zeta i a$ where $\zeta = (2a)^{(p-5)/8}$, $i = \sqrt{-1}$.
- $p \bmod 16 = 9$: Similar, work in a bi-quadratic extension of $\mathbf{F}_p$ for small p, Tonelli and Shanks for large p.
- $p \bmod 16 = 1$: Tonelli and Shanks.

REFERENCES:

- Siguna Muller. 'On the Computation of Square Roots in Finite Fields' Designs, Codes and Cryptography, Volume 31, Issue 3 (March 2004)
- A. Oliver L. Atkin. 'Probabilistic primality testing' (Chapter 30, Section 4) In Ph. Flajolet and P. Zimmermann, editors, Algorithms Seminar, 1991-1992. INRIA Research Report 1779, 1992, http://www.inria.fr/rrrt/rr-1779.html. Summary by F. Morain. http://citeseer.ist.psu.edu/atkin92probabilistic.html
- H. Postl. 'Fast evaluation of Dickson Polynomials' Contrib. to General Algebra, Vol. 6 (1988) pp. 223-225

AUTHORS:

- Robert Bradshaw

TESTS: Every case appears in the first hundred primes.

```
sage: from sage.rings.finite_rings.integer_mod import square_root_mod_prime    # sqrt() uses brut
sage: all([square_root_mod_prime(a*a)^2 == a*a
...        for p in prime_range(100)
...        for a in Integers(p)])
True
```

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: Perform p-adically by stripping off even powers of p to get a unit and lifting $\sqrt{unit} \bmod p$ via Newton's method.

AUTHORS:

- Robert Bradshaw

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)
sage: b^2 == a
True
```

CHAPTER

FIVE

FIELD Q OF RATIONAL NUMBERS

The class `RationalField` represents the field **Q** of (arbitrary precision) rational numbers. Each rational number is an instance of the class `Rational`.

Interactively, an instance of `RationalField` is available as `QQ`:

```
sage: QQ
Rational Field
```

Values of various types can be converted to rational numbers by using the `__call__` method of `RationalField` (that is, by treating `QQ` as a function).

```
sage: RealField(9).pi()
3.1
sage: QQ(RealField(9).pi())
22/7
sage: QQ(RealField().pi())
245850922/78256779
sage: QQ(35)
35
sage: QQ('12/347')
12/347
sage: QQ(exp(pi*I))
-1
sage: x = polygen(ZZ)
sage: QQ((3*x)/(4*x))
3/4
```

TEST:

```
sage: Q = RationalField()
sage: Q == loads(dumps(Q))
True
sage: RationalField() is RationalField()
True
```

AUTHORS:

- Niles Johnson (2010-08): trac ticket #3893: `random_element()` should pass on `*args` and `**kwds`.
- Travis Scrimshaw (2012-10-18): Added additional docstrings for full coverage. Removed duplicates of `discriminant()` and `signature()`.

class `sage.rings.rational_field.`**`RationalField`**

Bases: `sage.misc.fast_methods.Singleton`, `sage.rings.number_field.number_field_base.NumberF`

The class `RationalField` represents the field **Q** of rational numbers.

EXAMPLES:

```
sage: a = long(90182430982109382109381209381092830918309183209 1)
sage: b = QQ(a); b
90182430982109382109381209381092830918309183209 1
sage: QQ(b)
90182430982109382109381209381092830918309183209 1
sage: QQ(int(93820984323))
93820984323
sage: QQ(ZZ(90182430982109382109381209381092830918309183209 1))
90182430982109382109381209381092830918309183209 1
sage: QQ('-930482/9320842317')
-930482/9320842317
sage: QQ((-930482, 9320842317))
-930482/9320842317
sage: QQ([9320842317])
9320842317
sage: QQ(pari(39029384023840928309482842098430284398243982394))
39029384023840928309482842098430284398243982394
sage: QQ('sage')
Traceback (most recent call last):
...
TypeError: unable to convert sage to a rational
```

Conversion from the reals to the rationals is done by default using continued fractions.

```
sage: QQ(RR(3929329/32))
3929329/32
sage: QQ(-RR(3929329/32))
-3929329/32
sage: QQ(RR(1/7)) - 1/7
0
```

If you specify an optional second base argument, then the string representation of the float is used.

```
sage: QQ(23.2, 2)
6530219459687219/281474976710656
sage: 6530219459687219.0/281474976710656
23.200000000000000
sage: a = 23.2; a
23.200000000000000
sage: QQ(a, 10)
116/5
```

Here's a nice example involving elliptic curves:

```
sage: E = EllipticCurve('11a')
sage: L = E.lseries().at1(300)[0]; L
0.25384186085591068433775892 33...
sage: O = E.period_lattice().omega(); O
1.26920930427955
sage: t = L/O; t
0.200000000000000
sage: QQ(RealField(45)(t))
1/5
```

absolute_degree()

Return the absolute degree of $\mathbf{Q}$ which is 1.

EXAMPLES:

```
sage: QQ.absolute_degree()
1
```

absolute_discriminant()
Return the absolute discriminant, which is 1.

EXAMPLES:

```
sage: QQ.absolute_discriminant()
1
```

algebraic_closure()
Return the algebraic closure of self (which is $\overline{\mathbf{Q}}$).

EXAMPLES:

```
sage: QQ.algebraic_closure()
Algebraic Field
```

characteristic()
Return 0 since the rational field has characteristic 0.

EXAMPLES:

```
sage: c = QQ.characteristic(); c
0
sage: parent(c)
Integer Ring
```

class_number()
Return the class number of the field of rational numbers, which is 1.

EXAMPLES:

```
sage: QQ.class_number()
1
```

completion(*p*, *prec*, *extras={}*)
Return the completion of $\mathbf{Q}$ at p.

EXAMPLES:

```
sage: QQ.completion(infinity, 53)
Real Field with 53 bits of precision
sage: QQ.completion(5, 15, {'print_mode': 'bars'})
5-adic Field with capped relative precision 15
```

complex_embedding(*prec=53*)
Return embedding of the rational numbers into the complex numbers.

EXAMPLES:

```
sage: QQ.complex_embedding()
Ring morphism:
  From: Rational Field
  To:   Complex Field with 53 bits of precision
  Defn: 1 |--> 1.00000000000000
sage: QQ.complex_embedding(20)
Ring morphism:
  From: Rational Field
  To:   Complex Field with 20 bits of precision
  Defn: 1 |--> 1.0000
```

construction()
Returns a pair `(functor, parent)` such that `functor(parent)` returns `self`.

This is the construction of $\mathbf{Q}$ as the fraction field of $\mathbf{Z}$.

EXAMPLES:

```
sage: QQ.construction()
(FractionField, Integer Ring)
```

degree()
Return the degree of $\mathbf{Q}$ which is 1.

EXAMPLES:

```
sage: QQ.degree()
1
```

discriminant()
Return the discriminant of the field of rational numbers, which is 1.

EXAMPLES:

```
sage: QQ.discriminant()
1
```

embeddings(K)
Return list of the one embedding of $\mathbf{Q}$ into K, if it exists.

EXAMPLES:

```
sage: QQ.embeddings(QQ)
[Ring Coercion endomorphism of Rational Field]
sage: QQ.embeddings(CyclotomicField(5))
[Ring Coercion morphism:
  From: Rational Field
  To:   Cyclotomic Field of order 5 and degree 4]
```

K must have characteristic 0:

```
sage: QQ.embeddings(GF(3))
Traceback (most recent call last):
...
ValueError: no embeddings of the rational field into K.
```

extension(*poly*, *names*, *check=True*, *embedding=None*)
Create a field extension of $\mathbf{Q}$.

EXAMPLES:

We make a single absolute extension:

```
sage: K.<a> = QQ.extension(x^3 + 5); K
Number Field in a with defining polynomial x^3 + 5
```

We make an extension generated by roots of two polynomials:

```
sage: K.<a,b> = QQ.extension([x^3 + 5, x^2 + 3]); K
Number Field in a with defining polynomial x^3 + 5 over its base field
sage: b^2
-3
```

```
sage: a^3
-5
```

gen(*n=0*)
Return the n-th generator of **Q**.

There is only the 0-th generator which is 1.

EXAMPLES:

```
sage: QQ.gen()
1
```

gens()
Return a tuple of generators of **Q** which is only `(1,)`.

EXAMPLES:

```
sage: QQ.gens()
(1,)
```

is_absolute()
Q is an absolute extension of **Q**.

EXAMPLES:

```
sage: QQ.is_absolute()
True
```

is_field(*proof=True*)
Return `True`, since the rational field is a field.

EXAMPLES:

```
sage: QQ.is_field()
True
```

is_finite()
Return `False`, since the rational field is not finite.

EXAMPLES:

```
sage: QQ.is_finite()
False
```

is_prime_field()
Return `True` since **Q** is a prime field.

EXAMPLES:

```
sage: QQ.is_prime_field()
True
```

is_subring(*K*)
Return `True` if **Q** is a subring of K.

We are only able to determine this in some cases, e.g., when K is a field or of positive characteristic.

EXAMPLES:

```
sage: QQ.is_subring(QQ)
True
sage: QQ.is_subring(QQ['x'])
True
```

```
sage: QQ.is_subring(GF(7))
False
sage: QQ.is_subring(CyclotomicField(7))
True
sage: QQ.is_subring(ZZ)
False
sage: QQ.is_subring(Frac(ZZ))
True
```

maximal_order()

Return the maximal order of the rational numbers, i.e., the ring $\mathbf{Z}$ of integers.

EXAMPLES:

```
sage: QQ.maximal_order()
Integer Ring
sage: QQ.ring_of_integers ()
Integer Ring
```

ngens()

Return the number of generators of $\mathbf{Q}$ which is 1.

EXAMPLES:

```
sage: QQ.ngens()
1
```

number_field()

Return the number field associated to $\mathbf{Q}$. Since $\mathbf{Q}$ is a number field, this just returns $\mathbf{Q}$ again.

EXAMPLES:

```
sage: QQ.number_field() is QQ
True
```

order()

Return the order of $\mathbf{Q}$ which is ∞.

EXAMPLES:

```
sage: QQ.order()
+Infinity
```

places(*all_complex=False*, *prec=None*)

Return the collection of all infinite places of self, which in this case is just the embedding of self into $\mathbf{R}$.

By default, this returns homomorphisms into `RR`. If `prec` is not None, we simply return homomorphisms into `RealField(prec)` (or `RDF` if `prec=53`).

There is an optional flag `all_complex`, which defaults to False. If `all_complex` is True, then the real embeddings are returned as embeddings into the corresponding complex field.

For consistency with non-trivial number fields.

EXAMPLES:

```
sage: QQ.places()
[Ring morphism:
  From: Rational Field
  To:   Real Field with 53 bits of precision
  Defn: 1 |--> 1.00000000000000]
sage: QQ.places(prec=53)
```

```
[Ring morphism:
  From: Rational Field
  To:   Real Double Field
  Defn: 1 |--> 1.0]
sage: QQ.places(prec=200, all_complex=True)
[Ring morphism:
  From: Rational Field
  To:   Complex Field with 200 bits of precision
  Defn: 1 |--> 1.000000000000000000000000000000000000000000000000000000000000]
```

power_basis()

Return a power basis for this number field over its base field.

The power basis is always `[1]` for the rational field. This method is defined to make the rational field behave more like a number field.

EXAMPLES:

```
sage: QQ.power_basis()
[1]
```

primes_of_bounded_norm_iter(B)

Iterator yielding all primes less than or equal to B.

INPUT:

- `B` – a positive integer; upper bound on the primes generated.

OUTPUT:

An iterator over all integer primes less than or equal to B.

Note: This function exists for compatibility with the related number field method, though it returns prime integers, not ideals.

EXAMPLES:

```
sage: it = QQ.primes_of_bounded_norm_iter(10)
sage: list(it)
[2, 3, 5, 7]
sage: list(QQ.primes_of_bounded_norm_iter(1))
[]
```

random_element(*num_bound=None*, *den_bound=None*, **args*, ***kwds*)

Return an random element of **Q**.

Elements are constructed by randomly choosing integers for the numerator and denominator, not necessarily coprime.

INPUT:

- `num_bound` – a positive integer, specifying a bound on the absolute value of the numerator. If absent, no bound is enforced.
- `den_bound` – a positive integer, specifying a bound on the value of the denominator. If absent, the bound for the numerator will be reused.

Any extra positional or keyword arguments are passed through to `sage.rings.integer_ring.IntegerRing_class.random_element()`.

EXAMPLES:

```
sage: QQ.random_element()
-4
sage: QQ.random_element()
0
sage: QQ.random_element()
-1/2
```

In the following example, the resulting numbers range from -5/1 to 5/1 (both inclusive), while the smallest possible positive value is 1/10:

```
sage: QQ.random_element(5, 10)
-2/7
```

Extra positional or keyword arguments are passed through:

```
sage: QQ.random_element(distribution='1/n')
0
sage: QQ.random_element(distribution='1/n')
-1
```

range_by_height(*start*, *end=None*)

Range function for rational numbers, ordered by height.

Returns a Python generator for the list of rational numbers with heights in `range(start, end)`. Follows the same convention as Python range, see `range?` for details.

See also `__iter__()`.

EXAMPLES:

All rational numbers with height strictly less than 4:

```
sage: list(QQ.range_by_height(4))
[0, 1, -1, 1/2, -1/2, 2, -2, 1/3, -1/3, 3, -3, 2/3, -2/3, 3/2, -3/2]
sage: [a.height() for a in QQ.range_by_height(4)]
[1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3]
```

All rational numbers with height 2:

```
sage: list(QQ.range_by_height(2, 3))
[1/2, -1/2, 2, -2]
```

Nonsensical integer arguments will return an empty generator:

```
sage: list(QQ.range_by_height(3, 3))
[]
sage: list(QQ.range_by_height(10, 1))
[]
```

There are no rational numbers with height ≤ 0:

```
sage: list(QQ.range_by_height(-10, 1))
[]
```

relative_discriminant()

Return the relative discriminant, which is 1.

EXAMPLES:

```
sage: QQ.relative_discriminant()
1
```

residue_field(*p*, *check=True*)

Return the residue field of $\mathbf{Q}$ at the prime p, for consistency with other number fields.

INPUT:

- `p` - a prime integer.
- `check` (default True) - if True check the primality of p, else do not.

OUTPUT: The residue field at this prime.

EXAMPLES:

```
sage: QQ.residue_field(5)
Residue field of Integers modulo 5
sage: QQ.residue_field(next_prime(10^9))
Residue field of Integers modulo 1000000007
```

selmer_group(*S*, *m*, *proof=True*, *orders=False*)

Compute the group $\mathbf{Q}(S, m)$.

INPUT:

- `S` – a set of primes
- `m` – a positive integer
- `proof` – ignored
- `orders` (default False) – if True, output two lists, the generators and their orders

OUTPUT:

A list of generators of $\mathbf{Q}(S, m)$ (and, optionally, their orders in $\mathbf{Q}^\times/(\mathbf{Q}^\times)^m$). This is the subgroup of $\mathbf{Q}^\times/(\mathbf{Q}^\times)^m$ consisting of elements a such that the valuation of a is divisible by m at all primes not in S. It is equal to the group of S-units modulo m-th powers. The group $\mathbf{Q}(S, m)$ contains the subgroup of those a such that $\mathbf{Q}(\sqrt[m]{a})/\mathbf{Q}$ is unramified at all primes of $\mathbf{Q}$ outside of S, but may contain it properly when not all primes dividing m are in S.

EXAMPLES:

```
sage: QQ.selmer_group((), 2)
[-1]
sage: QQ.selmer_group((3,), 2)
[-1, 3]
sage: QQ.selmer_group((5,), 2)
[-1, 5]
```

The previous examples show that the group generated by the output may be strictly larger than the 'true' Selmer group of elements giving extensions unramified outside S.

When m is even, -1 is a generator of order 2:

```
sage: QQ.selmer_group((2,3,5,7,), 2, orders=True)
([-1, 2, 3, 5, 7], [2, 2, 2, 2, 2])
sage: QQ.selmer_group((2,3,5,7,), 3, orders=True)
([2, 3, 5, 7], [3, 3, 3, 3])
```

selmer_group_iterator(*S*, *m*, *proof=True*)

Return an iterator through elements of the finite group $\mathbf{Q}(S, m)$.

INPUT:

- `S` – a set of primes

- m – a positive integer
- proof – ignored

OUTPUT:

An iterator yielding the distinct elements of $\mathbf{Q}(S, m)$. See the docstring for selmer_group() for more information.

EXAMPLES:

```
sage: list(QQ.selmer_group_iterator((), 2))
[1, -1]
sage: list(QQ.selmer_group_iterator((2,), 2))
[1, 2, -1, -2]
sage: list(QQ.selmer_group_iterator((2,3), 2))
[1, 3, 2, 6, -1, -3, -2, -6]
sage: list(QQ.selmer_group_iterator((5,), 2))
[1, 5, -1, -5]
```

signature()

Return the signature of the rational field, which is $(1, 0)$, since there are 1 real and no complex embeddings.

EXAMPLES:

```
sage: QQ.signature()
(1, 0)
```

zeta(*n=2*)

Return a root of unity in self.

INPUT:

- n – integer (default: 2) order of the root of unity

EXAMPLES:

```
sage: QQ.zeta()
-1
sage: QQ.zeta(2)
-1
sage: QQ.zeta(1)
1
sage: QQ.zeta(3)
Traceback (most recent call last):
...
ValueError: no n-th root of unity in rational field
```

sage.rings.rational_field.**frac**(*n*, *d*)

Return the fraction n/d.

EXAMPLES:

```
sage: from sage.rings.rational_field import frac
sage: frac(1,2)
1/2
```

sage.rings.rational_field.**is_RationalField**(*x*)

Check to see if x is the rational field.

EXAMPLES:

```
sage: from sage.rings.rational_field import is_RationalField as is_RF
sage: is_RF(QQ)
```

```
True
sage: is_RF(ZZ)
False
```

CHAPTER

SIX

RATIONAL NUMBERS

AUTHORS:

- William Stein (2005): first version
- William Stein (2006-02-22): floor and ceil (pure fast GMP versions).
- Gonzalo Tornaria and William Stein (2006-03-02): greatly improved python/GMP conversion; hashing
- William Stein and Naqi Jaffery (2006-03-06): height, sqrt examples, and improve behavior of sqrt.
- David Harvey (2006-09-15): added nth_root
- Pablo De Napoli (2007-04-01): corrected the implementations of multiplicative_order, is_one; optimized __nonzero__ ; documented: lcm,gcd
- John Cremona (2009-05-15): added support for local and global logarithmic heights.
- Travis Scrimshaw (2012-10-18): Added doctests for full coverage.
- Vincent Delecroix (2013): continued fraction

TESTS:

```
sage: a = -2/3
sage: a == loads(dumps(a))
True
```

class sage.rings.rational.**Q_to_Z**

Bases: sage.categories.map.Map

A morphism from **Q** to **Z**.

TESTS:

```
sage: type(ZZ.convert_map_from(QQ))
<type 'sage.rings.rational.Q_to_Z'>
```

section()

Return a section of this morphism.

EXAMPLES:

```
sage: sage.rings.rational.Q_to_Z(QQ, ZZ).section()
Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

class sage.rings.rational.**Rational**

Bases: sage.structure.element.FieldElement

A rational number.

Rational numbers are implemented using the GMP C library.

EXAMPLES:

```
sage: a = -2/3
sage: type(a)
<type 'sage.rings.rational.Rational'>
sage: parent(a)
Rational Field
sage: Rational('1/0')
Traceback (most recent call last):
...
TypeError: unable to convert 1/0 to a rational
sage: Rational(1.5)
3/2
sage: Rational('9/6')
3/2
sage: Rational((2^99,2^100))
1/2
sage: Rational(("2", "10"), 16)
1/8
sage: Rational(QQbar(125/8).nth_root(3))
5/2
sage: Rational(AA(209735/343 - 17910/49*golden_ratio).nth_root(3) + 3*AA(golden_ratio))
53/7
sage: QQ(float(1.5))
3/2
sage: QQ(RDF(1.2))
6/5
```

Conversion from PARI:

```
sage: Rational(pari('-939082/3992923'))
-939082/3992923
sage: Rational(pari('Pol([-1/2])'))  #9595
-1/2
```

Conversions from numpy:

```
sage: import numpy as np
sage: QQ(np.int8('-15'))
-15
sage: QQ(np.int16('-32'))
-32
sage: QQ(np.int32('-19'))
-19
sage: QQ(np.uint32('1412'))
1412

sage: QQ(np.float16('12'))
12
```

absolute_norm()

Returns the norm from Q to Q of x (which is just x). This was added for compatibility with NumberFields

EXAMPLES:

```
sage: (6/5).absolute_norm()
6/5
```

```
sage: QQ(7/5).absolute_norm()
7/5
```

additive_order()
Return the additive order of `self`.

OUTPUT: integer or infinity

EXAMPLES:

```
sage: QQ(0).additive_order()
1
sage: QQ(1).additive_order()
+Infinity
```

ceil()
Return the ceiling of this rational number.

OUTPUT: Integer

If this rational number is an integer, this returns this number, otherwise it returns the floor of this number +1.

EXAMPLES:

```
sage: n = 5/3; n.ceil()
2
sage: n = -17/19; n.ceil()
0
sage: n = -7/2; n.ceil()
-3
sage: n = 7/2; n.ceil()
4
sage: n = 10/2; n.ceil()
5
```

charpoly(*var*)
Return the characteristic polynomial of this rational number. This will always be just `var - self`; this is really here so that code written for number fields won't crash when applied to rational numbers.

INPUT:

- `var` - a string

OUTPUT: Polynomial

EXAMPLES:

```
sage: (1/3).charpoly('x')
 x - 1/3
```

AUTHORS:

- Craig Citro

conjugate()
Return the complex conjugate of this rational number, which is the number itself.

EXAMPLES:

```
sage: n = 23/11
sage: n.conjugate()
23/11
```

content(*other*)

Return the content of `self` and `other`, i.e. the unique positive rational number c such that `self/c` and `other/c` are coprime integers.

`other` can be a rational number or a list of rational numbers.

EXAMPLES:

```
sage: a = 2/3
sage: a.content(2/3)
2/3
sage: a.content(1/5)
1/15
sage: a.content([2/5, 4/9])
2/45
```

continued_fraction()

Return the continued fraction of that rational.

EXAMPLES:

```
sage: (641/472).continued_fraction()
[1; 2, 1, 3, 1, 4, 1, 5]

sage: a = (355/113).continued_fraction(); a
[3; 7, 16]
sage: a.n(digits=10)
3.141592920
sage: pi.n(digits=10)
3.141592654
```

It's almost pi!

continued_fraction_list(*type='std'*)

Return the list of partial quotients of this rational number.

INPUT:

- `type` - either "std" (the default) for the standard continued fractions or "hj" for the Hirzebruch-Jung ones.

EXAMPLES:

```
sage: (13/9).continued_fraction_list()
[1, 2, 4]
sage: 1 + 1/(2 + 1/4)
13/9

sage: (225/157).continued_fraction_list()
[1, 2, 3, 4,  5]
sage: 1 + 1/(2 + 1/(3 + 1/(4 + 1/5)))
225/157

sage: (fibonacci(20)/fibonacci(19)).continued_fraction_list()
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]

sage: (-1/3).continued_fraction_list()
[-1, 1, 2]
```

Check that the partial quotients of an integer `n` is simply `[n]`:

```
sage: QQ(1).continued_fraction_list()
[1]
sage: QQ(0).continued_fraction_list()
[0]
sage: QQ(-1).continued_fraction_list()
[-1]
```

Hirzebruch-Jung continued fractions:

```
sage: (11/19).continued_fraction_list("hj")
[1, 3, 2, 3, 2]
sage: 1 - 1/(3 - 1/(2 - 1/(3 - 1/2)))
11/19

sage: (225/137).continued_fraction_list("hj")
[2, 3, 5, 10]
sage: 2 - 1/(3 - 1/(5 - 1/10))
225/137

sage: (-23/19).continued_fraction_list("hj")
[-1, 5, 4]
sage: -1 - 1/(5 - 1/4)
-23/19
```

denom()

Returns the denominator of this rational number.

EXAMPLES:

```
sage: x = 5/13
sage: x.denom()
13
sage: x = -9/3
sage: x.denom()
1
```

denominator()

Returns the denominator of this rational number.

EXAMPLES:

```
sage: x = -5/11
sage: x.denominator()
11
sage: x = 9/3
sage: x.denominator()
1
```

factor()

Return the factorization of this rational number.

OUTPUT: Factorization

EXAMPLES:

```
sage: (-4/17).factor()
-1 * 2^2 * 17^-1
```

Trying to factor 0 gives an arithmetic error:

```
sage: (0/1).factor()
Traceback (most recent call last):
...
ArithmeticError: Prime factorization of 0 not defined.
```

floor()

Return the floor of this rational number as an integer.

OUTPUT: Integer

EXAMPLES:

```
sage: n = 5/3; n.floor()
1
sage: n = -17/19; n.floor()
-1
sage: n = -7/2; n.floor()
-4
sage: n = 7/2; n.floor()
3
sage: n = 10/2; n.floor()
5
```

gamma(*prec=None*)

Return the gamma function evaluated at `self`. This value is exact for integers and half-integers, and returns a symbolic value otherwise. For a numerical approximation, use keyword `prec`.

EXAMPLES:

```
sage: gamma(1/2)
sqrt(pi)
sage: gamma(7/2)
15/8*sqrt(pi)
sage: gamma(-3/2)
4/3*sqrt(pi)
sage: gamma(6/1)
120
sage: gamma(1/3)
gamma(1/3)
```

This function accepts an optional precision argument:

```
sage: (1/3).gamma(prec=100)
2.6789385347077476336556929410
sage: (1/2).gamma(prec=100)
1.7724538509055160272981674833
```

global_height(*prec=None*)

Returns the absolute logarithmic height of this rational number.

INPUT:

- `prec` (int) – desired floating point precision (default: default RealField precision).

OUTPUT:

(real) The absolute logarithmic height of this rational number.

ALGORITHM:

The height is the sum of the total archimedean and non-archimedean components, which is equal to $\max(\log(n), \log(d))$ where n, d are the numerator and denominator of the rational number.

EXAMPLES:

```
sage: a = QQ(6/25)
sage: a.global_height_arch() + a.global_height_non_arch()
3.21887582486820
sage: a.global_height()
3.21887582486820
sage: (1/a).global_height()
3.21887582486820
sage: QQ(0).global_height()
0.000000000000000
sage: QQ(1).global_height()
0.000000000000000
```

global_height_arch(*prec=None*)

Returns the total archimedean component of the height of this rational number.

INPUT:

- `prec` (int) – desired floating point precision (default: default RealField precision).

OUTPUT:

(real) The total archimedean component of the height of this rational number.

ALGORITHM:

Since $\mathbf{Q}$ has only one infinite place this is just the value of the local height at that place. This separate function is included for compatibility with number fields.

EXAMPLES:

```
sage: a = QQ(6/25)
sage: a.global_height_arch()
0.000000000000000
sage: (1/a).global_height_arch()
1.42711635564015
sage: (1/a).global_height_arch(100)
1.427116355640145748389041308 1
```

global_height_non_arch(*prec=None*)

Returns the total non-archimedean component of the height of this rational number.

INPUT:

- `prec` (int) – desired floating point precision (default: default RealField precision).

OUTPUT:

(real) The total non-archimedean component of the height of this rational number.

ALGORITHM:

This is the sum of the local heights at all primes p, which may be computed without factorization as the log of the denominator.

EXAMPLES:

```
sage: a = QQ(5/6)
sage: a.support()
[2, 3, 5]
sage: a.global_height_non_arch()
1.79175946922805
sage: [a.local_height(p) for p in a.support()]
[0.693147180559945, 1.09861228866811, 0.000000000000000]
```

```
sage: sum([a.local_height(p) for p in a.support()])
1.79175946922805
```

height()

The max absolute value of the numerator and denominator of `self`, as an `Integer`.

OUTPUT: Integer

EXAMPLES:

```
sage: a = 2/3
sage: a.height()
3
sage: a = 34/3
sage: a.height()
34
sage: a = -97/4
sage: a.height()
97
```

AUTHORS:

- Naqi Jaffery (2006-03-05): examples

Note: For the logarithmic height, use `global_height()`.

---

imag()

Returns the imaginary part of `self`, which is zero.

EXAMPLES:

```
sage: (1/239).imag()
0
```

is_S_integral(*S*=[])

Determine if the rational number is `S`-integral.

`x` is `S`-integral if `x.valuation(p)>=0` for all `p` not in `S`, i.e., the denominator of `x` is divisible only by the primes in `S`.

INPUT:

- `S` – list or tuple of primes.

OUTPUT: bool

Note: Primality of the entries in `S` is not checked.

EXAMPLES:

```
sage: QQ(1/2).is_S_integral()
False
sage: QQ(1/2).is_S_integral([2])
True
sage: [a for a in range(1,11) if QQ(101/a).is_S_integral([2,5])]
[1, 2, 4, 5, 8, 10]
```

is_S_unit(*S*=*None*)

Determine if the rational number is an `S`-unit.

x is an S-unit if x.valuation(p)==0 for all p not in S, i.e., the numerator and denominator of x are divisible only by the primes in S.

INPUT:

- S – list or tuple of primes.

OUTPUT: bool

Note: Primality of the entries in S is not checked.

EXAMPLES:

```
sage: QQ(1/2).is_S_unit()
False
sage: QQ(1/2).is_S_unit([2])
True
sage: [a for a in range(1,11) if QQ(10/a).is_S_unit([2,5])]
[1, 2, 4, 5, 8, 10]
```

is_integer()

Determine if a rational number is integral (i.e is in **Z**).

OUTPUT: bool

EXAMPLES:

```
sage: QQ(1/2).is_integral()
False
sage: QQ(4/4).is_integral()
True
```

is_integral()

Determine if a rational number is integral (i.e is in **Z**).

OUTPUT: bool

EXAMPLES:

```
sage: QQ(1/2).is_integral()
False
sage: QQ(4/4).is_integral()
True
```

is_norm(*L*, *element=False*, *proof=True*)

Determine whether self is the norm of an element of L.

INPUT:

- L – a number field
- element – (default: False) boolean whether to also output an element of which self is a norm
- proof – If True, then the output is correct unconditionally. If False, then the output assumes GRH.

OUTPUT:

If element is False, then the output is a boolean B, which is True if and only if self is the norm of an element of L. If element is False, then the output is a pair (B, x), where B is as above. If B is True, then x an element of L such that self == x.norm(). Otherwise, x is None.

ALGORITHM:

Uses PARI's bnfisnorm. See _bnfisnorm().

EXAMPLES:

```
sage: K = NumberField(x^2 - 2, 'beta')
sage: (1/7).is_norm(K)
True
sage: (1/10).is_norm(K)
False
sage: 0.is_norm(K)
True
sage: (1/7).is_norm(K, element=True)
(True, 1/7*beta + 3/7)
sage: (1/10).is_norm(K, element=True)
(False, None)
sage: (1/691).is_norm(QQ, element=True)
(True, 1/691)
```

The number field doesn't have to be defined by an integral polynomial:

```
sage: B, e = (1/5).is_norm(QuadraticField(5/4, 'a'), element=True)
sage: B
True
sage: e.norm()
1/5
```

A non-Galois number field:

```
sage: K.<a> = NumberField(x^3-2)
sage: B, e = (3/5).is_norm(K, element=True); B
True
sage: e.norm()
3/5

sage: 7.is_norm(K)
Traceback (most recent call last):
...
NotImplementedError: is_norm is not implemented unconditionally for norms from non-Galois nu
sage: 7.is_norm(K, proof=False)
False
```

AUTHORS:

- Craig Citro (2008-04-05)
- Marco Streng (2010-12-03)

is_nth_power(*n*)

Returns `True` if self is an n-th power, else `False`.

INPUT:

- `n` - integer (must fit in C int type)

Note: Use this function when you need to test if a rational number is an n-th power, but do not need to know the value of its n-th root. If the value is needed, use `nth_root()`.

AUTHORS:

- John Cremona (2009-04-04)

EXAMPLES:

```
sage: QQ(25/4).is_nth_power(2)
True
sage: QQ(125/8).is_nth_power(3)
True
sage: QQ(-125/8).is_nth_power(3)
True
sage: QQ(25/4).is_nth_power(-2)
True

sage: QQ(9/2).is_nth_power(2)
False
sage: QQ(-25).is_nth_power(2)
False
```

is_one()

Determine if a rational number is one.

OUTPUT: bool

EXAMPLES:

```
sage: QQ(1/2).is_one()
False
sage: QQ(4/4).is_one()
True
```

is_padic_square(*p*)

Determines whether this rational number is a square in $\mathbf{Q}_p$ (or in R when `p = infinity`).

INPUT:

- `p` - a prime number, or `infinity`

EXAMPLES:

```
sage: QQ(2).is_padic_square(7)
True
sage: QQ(98).is_padic_square(7)
True
sage: QQ(2).is_padic_square(5)
False
```

TESTS:

```
sage: QQ(5/7).is_padic_square(int(2))
False
```

is_perfect_power(*expected_value=False*)

Returns `True` if `self` is a perfect power.

INPUT:

- `expected_value` - (bool) whether or not this rational is expected be a perfect power. This does not affect the correctness of the output, only the runtime.

If `expected_value` is `False` (default) it will check the smallest of the numerator and denominator is a perfect power as a first step, which is often faster than checking if the quotient is a perfect power.

EXAMPLES:

```
sage: (4/9).is_perfect_power()
True
sage: (144/1).is_perfect_power()
```

```
True
sage: (4/3).is_perfect_power()
False
sage: (2/27).is_perfect_power()
False
sage: (4/27).is_perfect_power()
False
sage: (-1/25).is_perfect_power()
False
sage: (-1/27).is_perfect_power()
True
sage: (0/1).is_perfect_power()
True
```

The second parameter does not change the result, but may change the runtime.

```
sage: (-1/27).is_perfect_power(True)
True
sage: (-1/25).is_perfect_power(True)
False
sage: (2/27).is_perfect_power(True)
False
sage: (144/1).is_perfect_power(True)
True
```

This test makes sure we workaround a bug in GMP (see trac ticket #4612):

```
sage: [ -a for a in srange(100) if not QQ(-a^3).is_perfect_power() ]
[]
sage: [ -a for a in srange(100) if not QQ(-a^3).is_perfect_power(True) ]
[]
```

is_square()

Return whether or not this rational number is a square.

OUTPUT: bool

EXAMPLES:

```
sage: x = 9/4
sage: x.is_square()
True
sage: x = (7/53)^100
sage: x.is_square()
True
sage: x = 4/3
sage: x.is_square()
False
sage: x = -1/4
sage: x.is_square()
False
```

list()

Return a list with the rational element in it, to be compatible with the method for number fields.

OUTPUT:

- `list` - the list `[self]`

EXAMPLES:

```
sage: m = 5/3
sage: m.list()
[5/3]
```

local_height(*p*, *prec=None*)

Returns the local height of this rational number at the prime p.

INPUT:

- p – a prime number
- prec (int) – desired floating point precision (default: default RealField precision).

OUTPUT:

(real) The local height of this rational number at the prime p.

EXAMPLES:

```
sage: a = QQ(25/6)
sage: a.local_height(2)
0.693147180559945
sage: a.local_height(3)
1.09861228866811
sage: a.local_height(5)
0.000000000000000
```

local_height_arch(*prec=None*)

Returns the Archimedean local height of this rational number at the infinite place.

INPUT:

- prec (int) – desired floating point precision (default: default RealField precision).

OUTPUT:

(real) The local height of this rational number x at the unique infinite place of $\mathbf{Q}$, which is $\max(\log(|x|), 0)$.

EXAMPLES:

```
sage: a = QQ(6/25)
sage: a.local_height_arch()
0.000000000000000
sage: (1/a).local_height_arch()
1.42711635564015
sage: (1/a).local_height_arch(100)
1.4271163556401457483890413081
```

minpoly(*var='x'*)

Return the minimal polynomial of this rational number. This will always be just `x - self`; this is really here so that code written for number fields won't crash when applied to rational numbers.

INPUT:

- var - a string

OUTPUT: Polynomial

EXAMPLES:

```
sage: (1/3).minpoly()
x - 1/3
sage: (1/3).minpoly('y')
y - 1/3
```

AUTHORS:

- Craig Citro

mod_ui(*n*)

Return the remainder upon division of `self` by the unsigned long integer `n`.

INPUT:

- `n` - an unsigned long integer

OUTPUT: integer

EXAMPLES:

```
sage: (-4/17).mod_ui(3)
1
sage: (-4/17).mod_ui(17)
Traceback (most recent call last):
...
ArithmeticError: The inverse of 0 modulo 17 is not defined.
```

multiplicative_order()

Return the multiplicative order of `self`.

OUTPUT: Integer or `infinity`

EXAMPLES:

```
sage: QQ(1).multiplicative_order()
1
sage: QQ('1/-1').multiplicative_order()
2
sage: QQ(0).multiplicative_order()
+Infinity
sage: QQ('2/3').multiplicative_order()
+Infinity
sage: QQ('1/2').multiplicative_order()
+Infinity
```

norm()

Returns the norm from $\mathbf{Q}$ to $\mathbf{Q}$ of x (which is just x). This was added for compatibility with `NumberFields`.

OUTPUT:

- `Rational` - reference to `self`

EXAMPLES:

```
sage: (1/3).norm()
 1/3
```

AUTHORS:

- Craig Citro

nth_root(*n*)

Computes the n-th root of `self`, or raises a `ValueError` if `self` is not a perfect n-th power.

INPUT:

- `n` - integer (must fit in C int type)

AUTHORS:

- David Harvey (2006-09-15)

EXAMPLES:

```
sage: (25/4).nth_root(2)
5/2
sage: (125/8).nth_root(3)
5/2
sage: (-125/8).nth_root(3)
-5/2
sage: (25/4).nth_root(-2)
2/5

sage: (9/2).nth_root(2)
Traceback (most recent call last):
...
ValueError: not a perfect 2nd power

sage: (-25/4).nth_root(2)
Traceback (most recent call last):
...
ValueError: cannot take even root of negative number
```

numer()

Return the numerator of this rational number.

EXAMPLE:

```
sage: x = -5/11
sage: x.numer()
-5
```

numerator()

Return the numerator of this rational number.

EXAMPLE:

```
sage: x = 5/11
sage: x.numerator()
5

sage: x = 9/3
sage: x.numerator()
3
```

ord(p)

Return the power of `p` in the factorization of self.

INPUT:

- `p` - a prime number

OUTPUT:

(integer or infinity) `Infinity` if `self` is zero, otherwise the (positive or negative) integer e such that `self` = $m * p^e$ with m coprime to p.

Note: See also `val_unit()` which returns the pair (e, m). The function `ord()` is an alias for `valuation()`.

EXAMPLES:

```
sage: x = -5/9
sage: x.valuation(5)
1
sage: x.ord(5)
1
sage: x.valuation(3)
-2
sage: x.valuation(2)
0
```

Some edge cases:

```
sage: (0/1).valuation(4)
+Infinity
sage: (7/16).valuation(4)
-2
```

period()

Return the period of the repeating part of the decimal expansion of this rational number.

ALGORITHM:

When a rational number n/d with $(n, d) = 1$ is expanded, the period begins after s terms and has length t, where s and t are the smallest numbers satisfying $10^s = 10^{s+t} \mod d$. In general if $d = 2^a 3^b m$ where m is coprime to 10, then $s = \max(a, b)$ and t is the order of 10 modulo d.

EXAMPLES:

```
sage: (1/7).period()
6
sage: RR(1/7)
0.142857142857143
sage: (1/8).period()
1
sage: RR(1/8)
0.125000000000000
sage: RR(1/6)
0.166666666666667
sage: (1/6).period()
1
sage: x = 333/106
sage: x.period()
13
sage: RealField(200)(x)
3.14150943396226415094339622641509433962264150943396226415094339622641509
```

prime_to_S_part(*S*=[])

Returns `self` with all powers of all primes in `S` removed.

INPUT:

- `S` - list or tuple of primes.

OUTPUT: rational

Note: Primality of the entries in S is not checked.

EXAMPLES:

```
sage: QQ(3/4).prime_to_S_part()
3/4
```

```
sage: QQ(3/4).prime_to_S_part([2])
3
sage: QQ(-3/4).prime_to_S_part([3])
-1/4
sage: QQ(700/99).prime_to_S_part([2,3,5])
7/11
sage: QQ(-700/99).prime_to_S_part([2,3,5])
-7/11
sage: QQ(0).prime_to_S_part([2,3,5])
0
sage: QQ(-700/99).prime_to_S_part([])
-700/99
```

real()

Returns the real part of `self`, which is `self`.

EXAMPLES:

```
sage: (1/2).real()
1/2
```

relative_norm()

Returns the norm from Q to Q of x (which is just x). This was added for compatibility with NumberFields

EXAMPLES:

```
sage: (6/5).relative_norm()
6/5

sage: QQ(7/5).relative_norm()
7/5
```

round(*mode='away'*)

Returns the nearest integer to `self`, rounding away from 0 by default, for consistency with the builtin Python round.

INPUT:

- `self` - a rational number
- `mode` - a rounding mode for half integers:
  - 'toward' rounds toward zero
  - 'away' (default) rounds away from zero
  - 'up' rounds up
  - 'down' rounds down
  - 'even' rounds toward the even integer
  - 'odd' rounds toward the odd integer

OUTPUT: Integer

EXAMPLES:

```
sage: (9/2).round()
5
sage: n = 4/3; n.round()
1
sage: n = -17/4; n.round()
-4
```

```
sage: n = -5/2; n.round()
-3
sage: n.round("away")
-3
sage: n.round("up")
-2
sage: n.round("down")
-3
sage: n.round("even")
-2
sage: n.round("odd")
-3
```

sign()

Returns the sign of this rational number, which is -1, 0, or 1 depending on whether this number is negative, zero, or positive respectively.

OUTPUT: Integer

EXAMPLES:

```
sage: (2/3).sign()
1
sage: (0/3).sign()
0
sage: (-1/6).sign()
-1
```

sqrt(*prec=None*, *extend=True*, *all=False*)

The square root function.

INPUT:

- `prec` – integer (default: `None`): if `None`, returns an exact square root; otherwise returns a numerical square root if necessary, to the given bits of precision.
- `extend` – bool (default: `True`); if `True`, return a square root in an extension ring, if necessary. Otherwise, raise a `ValueError` if the square is not in the base ring.
- `all` – bool (default: `False`); if `True`, return all square roots of self, instead of just one.

EXAMPLES:

```
sage: x = 25/9
sage: x.sqrt()
5/3
sage: sqrt(x)
5/3
sage: x = 64/4
sage: x.sqrt()
4
sage: x = 100/1
sage: x.sqrt()
10
sage: x.sqrt(all=True)
[10, -10]
sage: x = 81/5
sage: x.sqrt()
9*sqrt(1/5)
sage: x = -81/3
```

```
sage: x.sqrt()
3*sqrt(-3)

sage: n = 2/3
sage: n.sqrt()
sqrt(2/3)
sage: n.sqrt(prec=10)
0.82
sage: n.sqrt(prec=100)
0.81649658092772603273242802490
sage: n.sqrt(prec=100)^2
0.66666666666666666666666666667
sage: n.sqrt(prec=53, all=True)
[0.816496580927726, -0.816496580927726]
sage: n.sqrt(extend=False, all=True)
Traceback (most recent call last):
...
ValueError: square root of 2/3 not a rational number
sage: sqrt(-2/3, all=True)
[sqrt(-2/3), -sqrt(-2/3)]
sage: sqrt(-2/3, prec=53)
0.816496580927726*I
sage: sqrt(-2/3, prec=53, all=True)
[0.816496580927726*I, -0.816496580927726*I]
```

AUTHORS:

- Naqi Jaffery (2006-03-05): some examples

squarefree_part()

Return the square free part of x, i.e., an integer z such that $x = zy^2$, for a perfect square y^2.

EXAMPLES:

```
sage: a = 1/2
sage: a.squarefree_part()
2
sage: b = a/a.squarefree_part()
sage: b, b.is_square()
(1/4, True)
sage: a = 24/5
sage: a.squarefree_part()
30
```

str(*base=10*)

Return a string representation of `self` in the given `base`.

INPUT:

- `base` – integer (default: 10); base must be between 2 and 36.

OUTPUT: string

EXAMPLES:

```
sage: (-4/17).str()
'-4/17'
sage: (-4/17).str(2)
'-100/10001'
```

Note that the base must be at most 36.

```
sage: (-4/17).str(40)
Traceback (most recent call last):
...
ValueError: base (=40) must be between 2 and 36
sage: (-4/17).str(1)
Traceback (most recent call last):
...
ValueError: base (=1) must be between 2 and 36
```

support()

Return a sorted list of the primes where this rational number has non-zero valuation.

OUTPUT: The set of primes appearing in the factorization of this rational with nonzero exponent, as a sorted list.

EXAMPLES:

```
sage: (-4/17).support()
[2, 17]
```

Trying to find the support of 0 gives an arithmetic error:

```
sage: (0/1).support()
Traceback (most recent call last):
...
ArithmeticError: Support of 0 not defined.
```

trace()

Returns the trace from $\mathbf{Q}$ to $\mathbf{Q}$ of x (which is just x). This was added for compatibility with `NumberFields`.

OUTPUT:

- `Rational` - reference to self

EXAMPLES:

```
sage: (1/3).trace()
 1/3
```

AUTHORS:

- Craig Citro

trunc()

Round this rational number to the nearest integer toward zero.

EXAMPLES:

```
sage: (5/3).trunc()
1
sage: (-5/3).trunc()
-1
sage: QQ(42).trunc()
42
sage: QQ(-42).trunc()
-42
```

val_unit(p)

Returns a pair: the p-adic valuation of `self`, and the p-adic unit of `self`, as a `Rational`.

We do not require the p be prime, but it must be at least 2. For more documentation see `Integer.val_unit()`.

INPUT:

- `p` - a prime

OUTPUT:

- `int` - the p-adic valuation of this rational
- `Rational` - p-adic unit part of `self`

EXAMPLES:

```
sage: (-4/17).val_unit(2)
(2, -1/17)
sage: (-4/17).val_unit(17)
(-1, -4)
sage: (0/1).val_unit(17)
(+Infinity, 1)
```

AUTHORS:

- David Roe (2007-04-12)

valuation(p)

Return the power of `p` in the factorization of self.

INPUT:

- `p` - a prime number

OUTPUT:

(integer or infinity) `Infinity` if `self` is zero, otherwise the (positive or negative) integer e such that `self` = $m * p^e$ with m coprime to p.

Note: See also `val_unit()` which returns the pair (e, m). The function `ord()` is an alias for `valuation()`.

EXAMPLES:

```
sage: x = -5/9
sage: x.valuation(5)
1
sage: x.ord(5)
1
sage: x.valuation(3)
-2
sage: x.valuation(2)
0
```

Some edge cases:

```
sage: (0/1).valuation(4)
+Infinity
sage: (7/16).valuation(4)
-2
```

class `sage.rings.rational.`**Z_to_Q**

Bases: `sage.categories.morphism.Morphism`

A morphism from **Z** to **Q**.

section()
Return a section of this morphism.

EXAMPLES:

```
sage: f = QQ.coerce_map_from(ZZ).section(); f
Generic map:
  From: Rational Field
  To:   Integer Ring
```

This map is a morphism in the category of sets with partial maps (see trac ticket #15618):

```
sage: f.parent()
Set of Morphisms from Rational Field to Integer Ring in Category of sets with partial maps
```

class sage.rings.rational.**int_to_Q**
Bases: sage.categories.morphism.Morphism

A morphism from `int` to **Q**.

sage.rings.rational.**integer_rational_power**(a, b)
Compute a^b as an integer, if it is integral, or return None. The positive real root is taken for even denominators.

INPUT:

```
a -- an Integer
b -- a positive Rational
```

OUTPUT:

```
`a^b` as an ``Integer`` or ``None``
```

EXAMPLES:

```
sage: from sage.rings.rational import integer_rational_power
sage: integer_rational_power(49, 1/2)
7
sage: integer_rational_power(27, 1/3)
3
sage: integer_rational_power(-27, 1/3) is None
True
sage: integer_rational_power(-27, 2/3) is None
True
sage: integer_rational_power(512, 7/9)
128

sage: integer_rational_power(27, 1/4) is None
True
sage: integer_rational_power(-16, 1/4) is None
True

sage: integer_rational_power(0, 7/9)
0
sage: integer_rational_power(1, 7/9)
1
sage: integer_rational_power(-1, 7/9) is None
True
sage: integer_rational_power(-1, 8/9) is None
True
sage: integer_rational_power(-1, 9/8) is None
True
```

sage.rings.rational.**is_Rational**(*x*)

Return true if x is of the Sage rational number type.

EXAMPLES:

```
sage: from sage.rings.rational import is_Rational
sage: is_Rational(2)
False
sage: is_Rational(2/1)
True
sage: is_Rational(int(2))
False
sage: is_Rational(long(2))
False
sage: is_Rational('5')
False
```

sage.rings.rational.**make_rational**(*s*)

Make a rational number from `s` (a string in base 32)

INPUT:

- `s` - string in base 32

OUTPUT: Rational

EXAMPLES:

```
sage: (-7/15).str(32)
'-7/f'
sage: sage.rings.rational.make_rational('-7/f')
-7/15
```

sage.rings.rational.**rational_power_parts**(*a*, *b*, *factor_limit=100000*)

Compute rationals or integers c and d such that $a^b = c * d^b$ with d small. This is used for simplifying radicals.

INPUT:

```
- ``a`` -- a rational or integer
- ``b`` -- a rational
- ``factor_limit`` -- the limit used in factoring ``a``
```

EXAMPLES:

```
sage: from sage.rings.rational import rational_power_parts
sage: rational_power_parts(27, 1/2)
(3, 3)
sage: rational_power_parts(-128, 3/4)
(8, -8)
sage: rational_power_parts(-4, 1/2)
(2, -1)
sage: rational_power_parts(-4, 1/3)
(1, -4)
sage: rational_power_parts(9/1000, 1/2)
(3/10, 1/10)
```

TESTS:

Check if trac ticket #8540 is fixed:

```
sage: rational_power_parts(3/4, -1/2)
(2, 3)
sage: t = (3/4)^(-1/2); t
```

```
2/3*sqrt(3)
sage: t^2
4/3
```

CHAPTER

SEVEN

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 α 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.

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:

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

sage: k = GF(5^2,'c'); type(k)
<class 'sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro_with_category'>

sage: k = GF(2^16,'c'); type(k)
<class 'sage.rings.finite_rings.finite_field_ntl_gf2e.FiniteField_ntl_gf2e_with_category'>

sage: k = GF(3^16,'c'); type(k)
<class 'sage.rings.finite_rings.finite_field_pari_ffelt.FiniteField_pari_ffelt_with_category'>
```

Finite Fields support iteration, starting with 0.

```
sage: k = GF(9, 'a')
sage: for i,x in enumerate(k):  print 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
sage: for a in GF(5):
...    print a
0
1
2
3
4
```

We output the base rings of several finite fields.

```
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: 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.constructor.**FiniteFieldFactory**
Bases: sage.structure.factory.UniqueFactory

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

INPUT:

- `order` – a prime power
- `name` – string; must be specified unless `order` is prime.
- `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.
- `impl` – (optional) a string specifying the implementation of the finite field. Possible values are:
  - `'modn'` – ring of integers modulo p (only for prime fields).
  - `'givaro'` – Givaro, which uses Zech logs (only for fields of at most 65521 elements).
  - `'ntl'` – NTL using GF2X (only in characteristic 2).
  - `'pari_ffelt'` – PARI's `FFELT` type (only for extension fields).
  - `'pari_mod'` – Older PARI implementation using `POLMOD``s (slower than ``'pari_ffelt'`, only for extension fields).
- `elem_cache` – cache all elements to avoid creation time (default: order < 500)
- `check_irreducible` – verify that the polynomial modulus is irreducible
- `proof` – bool (default: `True`): if `True`, use provable primality test; otherwise only use pseudoprimality test.
- `args` – additional parameters passed to finite field implementations
- `kwds` – additional keyword parameters passed to finite field implementations

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

EXAMPLES:

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

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

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

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

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

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

```
sage: K.<a> = GF(5^40)
sage: a.multiplicative_order()
4547473508864641189575195312
sage: a.is_square()
True
sage: K.<b> = GF(5^40, modulus="primitive")
sage: b.multiplicative_order()
909494701772928237915039062 4
```

The modulus must be irreducible:

```
sage: K.<a> = GF(5**5, name='a', modulus=x^5 - x)
Traceback (most recent call last):
...
ValueError: finite field modulus must be irreducible but it is not
```

You can't accidentally fool the constructor into thinking the modulus is irreducible when it is not, since it actually tests irreducibility modulo p. Also, the modulus has to be of the right degree (this is always checked):

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

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

```
sage: K.<a> = GF(13^3, modulus=[1,0,0,2])
sage: K.<a> = GF(13^10, modulus=pari("ffinit(13,10)"))
sage: var('x')
x
sage: K.<a> = GF(13^2, modulus=x^2 - 2)
sage: K.<a> = GF(13^2, modulus=sin(x))
Traceback (most recent call last):
...
TypeError: unable to convert sin(x) to an integer
```

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: 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, conway=True, prefix='z'); a = k.gen()
sage: l = GF(5^5, conway=True, prefix='z'); b = l.gen()
sage: a + b
3*z10^5 + z10^4 + z10^2 + 3*z10 + 1
```

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

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

```
True
```

Another check that embeddings are defined properly:

```
sage: k = GF(3**10, conway=True, prefix='z')
sage: l = GF(3**20, conway=True, prefix='z')
sage: l(k.gen()**10) == l(k.gen())**10
True
```

Check that trac ticket #16934 has been fixed:

```
sage: k1.<a> = GF(17^14, impl="pari_ffelt")
sage: _ = a/2
sage: k2.<a> = GF(17^14, impl="pari_ffelt")
sage: k1 is k2
True
```

create_key_and_extra_args(*order*, *name=None*, *modulus=None*, *names=None*, *impl=None*, *proof=None*, *check_irreducible=True*, ***kwds*)

EXAMPLES:

```
sage: GF.create_key_and_extra_args(9, 'a')
((9, ('a',), x^2 + 2*x + 2, 'givaro', '{}', 3, 2, True), {})
sage: GF.create_key_and_extra_args(9, 'a', foo='value')
((9, ('a',), x^2 + 2*x + 2, 'givaro', "{'foo': 'value'}", 3, 2, True), {'foo': 'value'})
```

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

EXAMPLES:

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

We try to create finite fields with various implementations:

```
sage: k = GF(2, impl='modn')
sage: k = GF(2, impl='givaro')
sage: k = GF(2, impl='ntl')
sage: k = GF(2, impl='pari_ffelt')
Traceback (most recent call last):
...
ValueError: the degree must be at least 2
sage: k = GF(2, impl='pari_mod')
Traceback (most recent call last):
...
ValueError: The size of the finite field must not be prime.
sage: k = GF(2, impl='supercalifragilisticexpialidocious')
Traceback (most recent call last):
...
ValueError: no such finite field implementation: 'supercalifragilisticexpialidocious'
sage: k.<a> = GF(2^15, impl='modn')
Traceback (most recent call last):
...
ValueError: the 'modn' implementation requires a prime order
sage: k.<a> = GF(2^15, impl='givaro')
sage: k.<a> = GF(2^15, impl='ntl')
sage: k.<a> = GF(2^15, impl='pari_ffelt')
sage: k.<a> = GF(2^15, impl='pari_mod')
sage: k.<a> = GF(3^60, impl='modn')
Traceback (most recent call last):
...
```

```
ValueError: the 'modn' implementation requires a prime order
sage: k.<a> = GF(3^60, impl='givaro')
Traceback (most recent call last):
...
ValueError: q must be < 2^16
sage: k.<a> = GF(3^60, impl='ntl')
Traceback (most recent call last):
...
ValueError: q must be a 2-power
sage: k.<a> = GF(3^60, impl='pari_ffelt')
sage: k.<a> = GF(3^60, impl='pari_mod')
```

sage.rings.finite_rings.constructor.**is_PrimeFiniteField**(*x*)

Returns True if x is a prime finite field.

EXAMPLES:

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

CHAPTER

EIGHT

BASE CLASS FOR FINITE FIELD ELEMENTS

AUTHORS:

- David Roe (2010-1-14) -- factored out of sage.structure.element

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

Bases: sage.rings.finite_rings.element_base.FiniteRingElement

Elements represented as polynomials modulo a given ideal.

TESTS:

```
sage: k.<a> = GF(64)
sage: TestSuite(a).run()
```

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='matrix'*)

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

INPUT:

- `var` - string (default: 'x')
- `algorithm` - string (default: 'matrix')
  - 'matrix' - return the charpoly computed from the matrix of left multiplication by self
  - 'pari' – use pari's charpoly routine on polymods, which is not very good except in small cases

The result is not cached.

EXAMPLES:

```
sage: k.<a> = GF(19^2)
sage: parent(a)
Finite Field in a of size 19^2
sage: a.charpoly('X')
X^2 + 18*X + 2
sage: a^2 + 18*a + 2
```

```
0
sage: a.charpoly('X', algorithm='pari')
X^2 + 18*X + 2
```

frobenius(*k=1*)

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

INPUT:

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

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

EXAMPLES:

```
sage: 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
```

is_square()

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

EXAMPLES:

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

sage: k(0).is_square()
True
```

minimal_polynomial(*var='x'*)

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

EXAMPLES:

```
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'*)

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

EXAMPLES:

```
sage: k.<a> = FiniteField(19^2)
sage: parent(a)
Finite Field in a of size 19^2
sage: b=a**20;p=b.charpoly("x");p
x^2 + 15*x + 4
sage: factor(p)
(x + 17)^2
sage: b.minpoly('x')
x + 17
```

multiplicative_order()

Return the multiplicative order of this field element.

EXAMPLE:

```
sage: S.<a> = GF(5^3); S
Finite Field in a of size 5^3
sage: a.multiplicative_order()
124
sage: (a^8).multiplicative_order()
31
sage: S(0).multiplicative_order()
Traceback (most recent call last):
...
ArithmeticError: Multiplicative order of 0 not defined.
```

norm()

Return the norm of self down to the prime subfield.

This is the product of the Galois conjugates of self.

EXAMPLES:

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

Next we consider a cubic extension:

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

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

Returns an nth root of `self`.

INPUT:

- `n` - integer ≥ 1
- `extend` - bool (default: `False`); if `True`, return an nth root in an extension ring, if necessary. Otherwise, raise a ValueError if the root is not in the base ring. Warning: this option is not implemented!
- `all` - bool (default: `False`); if `True`, return all nth roots of `self`, instead of just one.
- `algorithm` - string (default: `None`); '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 nth root, returns one (if `all` is `False`) or a list of all of them (if `all` is `True`). Otherwise, raises a `ValueError` (if `extend` is `False`) or a `NotImplementedError` (if `extend` is `True`).

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

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
sage: b.nth_root(5, all = True)
[]
sage: b.nth_root(3, all = True)
[14*a + 18, 10*a + 13, 5*a + 27]

sage: k.<a> = GF(29^5)
sage: b = a^2 + 5*a + 1
sage: b.nth_root(5)
19*a^4 + 2*a^3 + 2*a^2 + 16*a + 3
sage: b.nth_root(7)
Traceback (most recent call last):
...
ValueError: no nth root
```

```
sage: b.nth_root(4, all=True)
[]
```

TESTS:

```
sage: for p in [2,3,5,7,11]:  # long time, random because of PARI warnings
....:     for n in [2,5,10]:
....:         q = p^n
....:         K.<a> = GF(q)
....:         for r in (q-1).divisors():
....:             if r == 1: continue
....:             x = K.random_element()
....:             y = x^r
....:             assert y.nth_root(r)^r == y
....:             assert (y^41).nth_root(41*r)^(41*r) == y^41
....:             assert (y^307).nth_root(307*r)^(307*r) == y^307
sage: k.<a> = GF(4)
sage: a.nth_root(0,all=True)
[]
sage: k(1).nth_root(0,all=True)
[a, a + 1, 1]
```

ALGORITHMS:

- The default is currently an algorithm described in the following paper:

Johnston, Anna M. A generalized qth root algorithm. Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms. Baltimore, 1999: pp 929-930.

AUTHOR:

- David Roe (2010-02-13)

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:

```
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
```

pth_root (*k=1*)

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

INPUT:

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

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

EXAMPLES:

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

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

See :meth:square_root().

EXAMPLES:

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

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

The square root function.

INPUT:

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

  Warning: This option is not implemented!

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

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

EXAMPLES:

```
sage: F = FiniteField(7^2, 'a')
sage: F(2).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.
```

trace()

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

EXAMPLES:

```
sage: S.<a> = GF(5^3); S
Finite Field in a of size 5^3
sage: a.trace()
0
sage: a.charpoly('t')
t^3 + 3*t + 3
sage: a + a^5 + a^25
0
sage: z = a^2 + a + 1
sage: z.trace()
2
sage: z.charpoly('t')
t^3 + 3*t^2 + 2*t + 2
sage: z + z^5 + z^25
2
```

class sage.rings.finite_rings.element_base.**FiniteRingElement**

Bases: sage.structure.element.CommutativeRingElement

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

Returns if x is a finite field element.

EXAMPLE:

```
sage: from sage.rings.finite_rings.element_base import is_FiniteFieldElement
sage: is_FiniteFieldElement(1)
False
sage: is_FiniteFieldElement(IntegerRing())
False
sage: is_FiniteFieldElement(GF(5)(2))
True
```

CHAPTER

NINE

INDICES AND TABLES

- Index
- Module Index
- Search Page

[IEEEP1363] IEEE P1363 / D13 (Draft Version 13). Standard Specifications for Public Key Cryptography Annex A (Informative). Number-Theoretic Background. Section A.2.4

[Pos88] H. Postl. 'Fast evaluation of Dickson Polynomials' Contrib. to General Algebra, Vol. 6 (1988) pp. 223-225

r

sage.rings.finite_rings.constructor, 131
sage.rings.finite_rings.element_base, 139
sage.rings.finite_rings.integer_mod, 71
sage.rings.finite_rings.integer_mod_ring, 57
sage.rings.integer, 13
sage.rings.integer_ring, 1
sage.rings.rational, 107
sage.rings.rational_field, 95

A

absolute_degree() (sage.rings.integer_ring.IntegerRing_class method), 4
absolute_degree() (sage.rings.rational_field.RationalField method), 96
absolute_discriminant() (sage.rings.rational_field.RationalField method), 97
absolute_norm() (sage.rings.rational.Rational method), 108
additive_order() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 139
additive_order() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 72
additive_order() (sage.rings.integer.Integer method), 15
additive_order() (sage.rings.rational.Rational method), 109
algebraic_closure() (sage.rings.rational_field.RationalField method), 97

B

binary() (sage.rings.integer.Integer method), 16
binomial() (sage.rings.integer.Integer method), 16
bits() (sage.rings.integer.Integer method), 17

C

cardinality() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 61
ceil() (sage.rings.integer.Integer method), 17
ceil() (sage.rings.rational.Rational method), 109
centerlift() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 72
characteristic() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 62
characteristic() (sage.rings.integer_ring.IntegerRing_class method), 4
characteristic() (sage.rings.rational_field.RationalField method), 97
charpoly() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 139
charpoly() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 72
charpoly() (sage.rings.rational.Rational method), 109
class_number() (sage.rings.integer.Integer method), 17
class_number() (sage.rings.rational_field.RationalField method), 97
completion() (sage.rings.integer_ring.IntegerRing_class method), 4
completion() (sage.rings.rational_field.RationalField method), 97
complex_embedding() (sage.rings.rational_field.RationalField method), 97
conjugate() (sage.rings.integer.Integer method), 18
conjugate() (sage.rings.rational.Rational method), 109
construction() (sage.rings.rational_field.RationalField method), 98
content() (sage.rings.rational.Rational method), 109
continued_fraction() (sage.rings.rational.Rational method), 110

continued_fraction_list() (sage.rings.rational.Rational method), 110
coprime_integers() (sage.rings.integer.Integer method), 18
create_key_and_extra_args() (sage.rings.finite_rings.constructor.FiniteFieldFactory method), 136
create_key_and_extra_args() (sage.rings.finite_rings.integer_mod_ring.IntegerModFactory method), 59
create_object() (sage.rings.finite_rings.constructor.FiniteFieldFactory method), 136
create_object() (sage.rings.finite_rings.integer_mod_ring.IntegerModFactory method), 59
crt() (in module sage.rings.finite_rings.integer_mod_ring), 70
crt() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 73
crt() (sage.rings.integer.Integer method), 19
crt_basis() (in module sage.rings.integer_ring), 11

D

degree() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 62
degree() (sage.rings.integer_ring.IntegerRing_class method), 5
degree() (sage.rings.rational_field.RationalField method), 98
denom() (sage.rings.rational.Rational method), 111
denominator() (sage.rings.integer.Integer method), 19
denominator() (sage.rings.rational.Rational method), 111
digits() (sage.rings.integer.Integer method), 19
discriminant() (sage.rings.rational_field.RationalField method), 98
divide_knowing_divisible_by() (sage.rings.integer.Integer method), 21
divides() (sage.rings.integer.Integer method), 21
divisors() (sage.rings.integer.Integer method), 21

E

embeddings() (sage.rings.rational_field.RationalField method), 98
euclidean_degree() (sage.rings.integer.Integer method), 23
exact_log() (sage.rings.integer.Integer method), 23
exp() (sage.rings.integer.Integer method), 24
extension() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 62
extension() (sage.rings.integer_ring.IntegerRing_class method), 5
extension() (sage.rings.rational_field.RationalField method), 98

F

factor() (sage.rings.integer.Integer method), 24
factor() (sage.rings.rational.Rational method), 111
factored_order() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 62
factored_unit_order() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 62
factorial() (sage.rings.integer.Integer method), 26
field() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 62
FiniteFieldFactory (class in sage.rings.finite_rings.constructor), 132
FinitePolyExtElement (class in sage.rings.finite_rings.element_base), 139
FiniteRingElement (class in sage.rings.finite_rings.element_base), 145
floor() (sage.rings.integer.Integer method), 26
floor() (sage.rings.rational.Rational method), 112
frac() (in module sage.rings.rational_field), 104
fraction_field() (sage.rings.integer_ring.IntegerRing_class method), 5
free_integer_pool() (in module sage.rings.integer), 54
frobenius() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 140

G

gamma() (sage.rings.integer.Integer method), 26
gamma() (sage.rings.rational.Rational method), 112
gcd() (sage.rings.finite_rings.integer_mod.IntegerMod_gmp method), 84
gcd() (sage.rings.finite_rings.integer_mod.IntegerMod_int method), 85
gcd() (sage.rings.finite_rings.integer_mod.IntegerMod_int64 method), 88
gcd() (sage.rings.integer.Integer method), 26
GCD_list() (in module sage.rings.integer), 14
gen() (sage.rings.integer_ring.IntegerRing_class method), 5
gen() (sage.rings.rational_field.RationalField method), 99
generalised_log() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 73
gens() (sage.rings.integer_ring.IntegerRing_class method), 6
gens() (sage.rings.rational_field.RationalField method), 99
get_object() (sage.rings.finite_rings.integer_mod_ring.IntegerModFactory method), 59
global_height() (sage.rings.integer.Integer method), 27
global_height() (sage.rings.rational.Rational method), 112
global_height_arch() (sage.rings.rational.Rational method), 113
global_height_non_arch() (sage.rings.rational.Rational method), 113

H

height() (sage.rings.rational.Rational method), 114

I

imag() (sage.rings.integer.Integer method), 27
imag() (sage.rings.rational.Rational method), 114
Int_to_IntegerMod (class in sage.rings.finite_rings.integer_mod), 71
int_to_Q (class in sage.rings.rational), 128
int_to_Z (class in sage.rings.integer), 54
Integer (class in sage.rings.integer), 15
integer_rational_power() (in module sage.rings.rational), 128
Integer_to_IntegerMod (class in sage.rings.finite_rings.integer_mod), 89
IntegerMod() (in module sage.rings.finite_rings.integer_mod), 72
IntegerMod_abstract (class in sage.rings.finite_rings.integer_mod), 72
IntegerMod_gmp (class in sage.rings.finite_rings.integer_mod), 84
IntegerMod_hom (class in sage.rings.finite_rings.integer_mod), 85
IntegerMod_int (class in sage.rings.finite_rings.integer_mod), 85
IntegerMod_int64 (class in sage.rings.finite_rings.integer_mod), 87
IntegerMod_to_Integer (class in sage.rings.finite_rings.integer_mod), 88
IntegerMod_to_IntegerMod (class in sage.rings.finite_rings.integer_mod), 88
IntegerModFactory (class in sage.rings.finite_rings.integer_mod_ring), 57
IntegerModRing_generic (class in sage.rings.finite_rings.integer_mod_ring), 59
IntegerRing() (in module sage.rings.integer_ring), 1
IntegerRing_class (class in sage.rings.integer_ring), 1
IntegerWrapper (class in sage.rings.integer), 53
inverse_mod() (sage.rings.integer.Integer method), 27
inverse_of_unit() (sage.rings.integer.Integer method), 28
is_absolute() (sage.rings.rational_field.RationalField method), 99
is_field() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 63
is_field() (sage.rings.integer_ring.IntegerRing_class method), 6
is_field() (sage.rings.rational_field.RationalField method), 99

is_finite() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 64
is_finite() (sage.rings.integer_ring.IntegerRing_class method), 6
is_finite() (sage.rings.rational_field.RationalField method), 99
is_FiniteFieldElement() (in module sage.rings.finite_rings.element_base), 145
is_Integer() (in module sage.rings.integer), 54
is_integer() (sage.rings.integer.Integer method), 28
is_integer() (sage.rings.rational.Rational method), 115
is_IntegerMod() (in module sage.rings.finite_rings.integer_mod), 90
is_IntegerModRing() (in module sage.rings.finite_rings.integer_mod_ring), 70
is_IntegerRing() (in module sage.rings.integer_ring), 11
is_integral() (sage.rings.integer.Integer method), 28
is_integral() (sage.rings.rational.Rational method), 115
is_integral_domain() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 64
is_integrally_closed() (sage.rings.integer_ring.IntegerRing_class method), 6
is_irreducible() (sage.rings.integer.Integer method), 28
is_nilpotent() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 74
is_noetherian() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 64
is_noetherian() (sage.rings.integer_ring.IntegerRing_class method), 6
is_norm() (sage.rings.integer.Integer method), 29
is_norm() (sage.rings.rational.Rational method), 115
is_nth_power() (sage.rings.rational.Rational method), 116
is_one() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 74
is_one() (sage.rings.finite_rings.integer_mod.IntegerMod_gmp method), 84
is_one() (sage.rings.finite_rings.integer_mod.IntegerMod_int method), 85
is_one() (sage.rings.finite_rings.integer_mod.IntegerMod_int64 method), 88
is_one() (sage.rings.integer.Integer method), 29
is_one() (sage.rings.rational.Rational method), 117
is_padic_square() (sage.rings.rational.Rational method), 117
is_perfect_power() (sage.rings.integer.Integer method), 29
is_perfect_power() (sage.rings.rational.Rational method), 117
is_power_of() (sage.rings.integer.Integer method), 30
is_prime() (sage.rings.integer.Integer method), 31
is_prime_field() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 64
is_prime_field() (sage.rings.rational_field.RationalField method), 99
is_prime_power() (sage.rings.integer.Integer method), 32
is_PrimeFiniteField() (in module sage.rings.finite_rings.constructor), 137
is_primitive_root() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 74
is_pseudoprime() (sage.rings.integer.Integer method), 33
is_pseudoprime_power() (sage.rings.integer.Integer method), 34
is_Rational() (in module sage.rings.rational), 128
is_RationalField() (in module sage.rings.rational_field), 104
is_S_integral() (sage.rings.rational.Rational method), 114
is_S_unit() (sage.rings.rational.Rational method), 114
is_square() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 140
is_square() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 75
is_square() (sage.rings.integer.Integer method), 34
is_square() (sage.rings.rational.Rational method), 118
is_squarefree() (sage.rings.integer.Integer method), 34
is_subring() (sage.rings.integer_ring.IntegerRing_class method), 6
is_subring() (sage.rings.rational_field.RationalField method), 99

is_unique_factorization_domain() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 64
is_unit() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 75
is_unit() (sage.rings.finite_rings.integer_mod.IntegerMod_gmp method), 84
is_unit() (sage.rings.finite_rings.integer_mod.IntegerMod_int method), 86
is_unit() (sage.rings.finite_rings.integer_mod.IntegerMod_int64 method), 88
is_unit() (sage.rings.integer.Integer method), 35
isqrt() (sage.rings.integer.Integer method), 35

J

jacobi() (sage.rings.integer.Integer method), 35

K

kronecker() (sage.rings.integer.Integer method), 35
krull_dimension() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 64
krull_dimension() (sage.rings.integer_ring.IntegerRing_class method), 6

L

LCM_list() (in module sage.rings.integer), 53
lift() (sage.rings.finite_rings.integer_mod.IntegerMod_gmp method), 85
lift() (sage.rings.finite_rings.integer_mod.IntegerMod_int method), 86
lift() (sage.rings.finite_rings.integer_mod.IntegerMod_int64 method), 88
list() (sage.rings.integer.Integer method), 36
list() (sage.rings.rational.Rational method), 118
list_of_elements_of_multiplicative_group() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 65
local_height() (sage.rings.rational.Rational method), 119
local_height_arch() (sage.rings.rational.Rational method), 119
log() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 75
log() (sage.rings.integer.Integer method), 36
long_to_Z (class in sage.rings.integer), 55
lucas() (in module sage.rings.finite_rings.integer_mod), 90
lucas_q1() (in module sage.rings.finite_rings.integer_mod), 91

M

make_integer() (in module sage.rings.integer), 55
make_rational() (in module sage.rings.rational), 129
makeNativeIntStruct() (in module sage.rings.finite_rings.integer_mod), 91
maximal_order() (sage.rings.rational_field.RationalField method), 100
minimal_polynomial() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 140
minimal_polynomial() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 77
minpoly() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 141
minpoly() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 77
minpoly() (sage.rings.rational.Rational method), 119
Mod() (in module sage.rings.finite_rings.integer_mod), 89
mod() (in module sage.rings.finite_rings.integer_mod), 91
mod_ui() (sage.rings.rational.Rational method), 120
modulus() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 77
modulus() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 65
multifactorial() (sage.rings.integer.Integer method), 37
multiplicative_generator() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 65

multiplicative_group_is_cyclic() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 65
multiplicative_order() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 141
multiplicative_order() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 77
multiplicative_order() (sage.rings.integer.Integer method), 37
multiplicative_order() (sage.rings.rational.Rational method), 120
multiplicative_subgroups() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 66

N

NativeIntStruct (class in sage.rings.finite_rings.integer_mod), 89
nbits() (sage.rings.integer.Integer method), 37
ndigits() (sage.rings.integer.Integer method), 38
next_prime() (sage.rings.integer.Integer method), 38
next_prime_power() (sage.rings.integer.Integer method), 39
next_probable_prime() (sage.rings.integer.Integer method), 39
ngens() (sage.rings.integer_ring.IntegerRing_class method), 7
ngens() (sage.rings.rational_field.RationalField method), 100
norm() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 141
norm() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 77
norm() (sage.rings.rational.Rational method), 120
nth_root() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 142
nth_root() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 77
nth_root() (sage.rings.integer.Integer method), 40
nth_root() (sage.rings.rational.Rational method), 120
number_field() (sage.rings.rational_field.RationalField method), 100
numer() (sage.rings.rational.Rational method), 121
numerator() (sage.rings.integer.Integer method), 41
numerator() (sage.rings.rational.Rational method), 121

O

odd_part() (sage.rings.integer.Integer method), 41
ord() (sage.rings.integer.Integer method), 41
ord() (sage.rings.rational.Rational method), 121
order() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 66
order() (sage.rings.integer_ring.IntegerRing_class method), 7
order() (sage.rings.rational_field.RationalField method), 100
ordinal_str() (sage.rings.integer.Integer method), 42

P

parameter() (sage.rings.integer_ring.IntegerRing_class method), 7
perfect_power() (sage.rings.integer.Integer method), 42
period() (sage.rings.rational.Rational method), 122
places() (sage.rings.rational_field.RationalField method), 100
polynomial() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 79
popcount() (sage.rings.integer.Integer method), 43
power_basis() (sage.rings.rational_field.RationalField method), 101
powermod() (sage.rings.integer.Integer method), 43
powermodm_ui() (sage.rings.integer.Integer method), 44
precompute_table() (sage.rings.finite_rings.integer_mod.NativeIntStruct method), 90
previous_prime() (sage.rings.integer.Integer method), 44
previous_prime_power() (sage.rings.integer.Integer method), 44

prime_divisors() (sage.rings.integer.Integer method), 45
prime_factors() (sage.rings.integer.Integer method), 45
prime_to_m_part() (sage.rings.integer.Integer method), 45
prime_to_S_part() (sage.rings.rational.Rational method), 122
primes_of_bounded_norm_iter() (sage.rings.rational_field.RationalField method), 101
pth_power() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 143
pth_root() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 143
Python Enhancement Proposals
    PEP 3127, 15

Q

Q_to_Z (class in sage.rings.rational), 107
quadratic_nonresidue() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 66
quo_rem() (sage.rings.integer.Integer method), 46
quotient() (sage.rings.integer_ring.IntegerRing_class method), 7

R

radical() (sage.rings.integer.Integer method), 46
random_element() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 67
random_element() (sage.rings.integer_ring.IntegerRing_class method), 7
random_element() (sage.rings.rational_field.RationalField method), 101
range() (sage.rings.integer_ring.IntegerRing_class method), 9
range_by_height() (sage.rings.rational_field.RationalField method), 102
Rational (class in sage.rings.rational), 107
rational_power_parts() (in module sage.rings.rational), 129
rational_reconstruction() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 80
rational_reconstruction() (sage.rings.integer.Integer method), 47
RationalField (class in sage.rings.rational_field), 95
real() (sage.rings.integer.Integer method), 47
real() (sage.rings.rational.Rational method), 123
relative_discriminant() (sage.rings.rational_field.RationalField method), 102
relative_norm() (sage.rings.rational.Rational method), 123
residue_field() (sage.rings.integer_ring.IntegerRing_class method), 9
residue_field() (sage.rings.rational_field.RationalField method), 102
round() (sage.rings.rational.Rational method), 123

S

sage.rings.finite_rings.constructor (module), 131
sage.rings.finite_rings.element_base (module), 139
sage.rings.finite_rings.integer_mod (module), 71
sage.rings.finite_rings.integer_mod_ring (module), 57
sage.rings.integer (module), 13
sage.rings.integer_ring (module), 1
sage.rings.rational (module), 107
sage.rings.rational_field (module), 95
section() (sage.rings.finite_rings.integer_mod.Integer_to_IntegerMod method), 89
section() (sage.rings.rational.Q_to_Z method), 107
section() (sage.rings.rational.Z_to_Q method), 127
selmer_group() (sage.rings.rational_field.RationalField method), 103
selmer_group_iterator() (sage.rings.rational_field.RationalField method), 103

sign() (sage.rings.integer.Integer method), 47
sign() (sage.rings.rational.Rational method), 124
signature() (sage.rings.rational_field.RationalField method), 104
slow_lucas() (in module sage.rings.finite_rings.integer_mod), 92
sqrt() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 144
sqrt() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 80
sqrt() (sage.rings.finite_rings.integer_mod.IntegerMod_int method), 86
sqrt() (sage.rings.integer.Integer method), 48
sqrt() (sage.rings.rational.Rational method), 124
sqrtrem() (sage.rings.integer.Integer method), 48
square_root() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 144
square_root() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 81
square_root_mod_prime() (in module sage.rings.finite_rings.integer_mod), 92
square_root_mod_prime_power() (in module sage.rings.finite_rings.integer_mod), 93
square_roots_of_one() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 67
squarefree_part() (sage.rings.integer.Integer method), 49
squarefree_part() (sage.rings.rational.Rational method), 125
str() (sage.rings.integer.Integer method), 49
str() (sage.rings.rational.Rational method), 125
support() (sage.rings.integer.Integer method), 50
support() (sage.rings.rational.Rational method), 126

T

test_bit() (sage.rings.integer.Integer method), 50
trace() (sage.rings.finite_rings.element_base.FinitePolyExtElement method), 144
trace() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 83
trace() (sage.rings.rational.Rational method), 126
trailing_zero_bits() (sage.rings.integer.Integer method), 51
trial_division() (sage.rings.integer.Integer method), 51
trunc() (sage.rings.rational.Rational method), 126

U

unit_gens() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 67
unit_group() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 68
unit_group_exponent() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 69
unit_group_order() (sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic method), 70

V

val_unit() (sage.rings.integer.Integer method), 52
val_unit() (sage.rings.rational.Rational method), 126
valuation() (sage.rings.finite_rings.integer_mod.IntegerMod_abstract method), 83
valuation() (sage.rings.integer.Integer method), 52
valuation() (sage.rings.rational.Rational method), 127

X

xgcd() (sage.rings.integer.Integer method), 53

Z

Z_to_Q (class in sage.rings.rational), 127
zeta() (sage.rings.integer_ring.IntegerRing_class method), 10

zeta() (sage.rings.rational_field.RationalField method), 104