Standard Commutative Rings

Release 9.7

The Sage Development Team

Jul 21, 2024

CONTENTS

1 Integers **1**
- 1.1 Ring **Z** of Integers . . . 1
- 1.2 Elements of the ring **Z** of integers . . . 13
- 1.3 Cython wrapper for bernmm library . . . 61
- 1.4 Bernoulli numbers modulo p . . . 62
- 1.5 Integer factorization functions . . . 65
- 1.6 Basic arithmetic with C integers . . . 68
- 1.7 Fast decomposition of small integers into sums of squares . . . 69
- 1.8 Fast Arithmetic Functions . . . 71
- 1.9 Generic implementation of powering . . . 73
- 1.10 Utility classes for multi-modular algorithms . . . 74
- 1.11 Miscellaneous arithmetic functions . . . 77

2 Rationals **145**
- 2.1 Field **Q** of Rational Numbers . . . 145
- 2.2 Rational Numbers . . . 159

3 Indices and Tables **187**

Python Module Index **189**

Index **191**

CHAPTER

ONE

INTEGERS

1.1 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); a
1234
sage: b = Z(5678); b
5678
sage: type(a)
<class 'sage.rings.integer.Integer'>
sage: a + b
6912
sage: Z('9480384908398593485983458394 5394')
948038490839859348598345839455394
```

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
     and Category of metric spaces
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')
<... '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)
```

(continues on next page)

(continued from previous page)

```
928270
sage: 14 + 16*32 + 10*32^2 + 28*32^3
928270
```

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

```
sage: a = Z(1234); a
1234
sage: b = Z(5678); b
5678
sage: type(a)
<class '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)
<class 'sage.rings.rational.Rational'>
sage: a/a
1
sage: type(a/a)
<class 'sage.rings.rational.Rational'>
```

For floor division, use the // operator instead:

```
sage: a // b
0
sage: type(a//b)
<class '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
```

(continues on next page)

(continued from previous page)

```
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.

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 metric 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)
Integer Ring
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*, ***kwds*)
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())
<class '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])
<class 'sage.rings.integer.Integer'>
```

is_field(*proof=True*)

Return False since the integers are not a field.

EXAMPLES:

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

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*, ***kwds*)
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.DiscreteGaussianDistributionIntegerSampler 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().parent() is ZZ
True
```

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

```
sage: from collections import defaultdict
sage: def add_samples(*args, **kwds):
....:     global dic, counter
....:     for _ in range(100):
....:         counter += 1
....:         dic[ZZ.random_element(*args, **kwds)] += 1
```

(continues on next page)

(continued from previous page)

```
sage: prob = lambda x : 1/5
sage: dic = defaultdict(Integer)
sage: counter = 0.0
sage: add_samples(distribution="uniform")
sage: while any(abs(dic[i]/counter - prob(i)) > 0.01 for i in dic):
....:     add_samples(distribution="uniform")
```

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

```
sage: def prob(n):
....:     if n == 0:
....:         return 1/5
....:     return 2/(5*abs(n)*(abs(n) + 1))
sage: dic = defaultdict(Integer)
sage: counter = 0.0
sage: add_samples(distribution="1/n")
sage: while any(abs(dic[i]/counter - prob(i)) > 0.01 for i in dic):
....:     add_samples(distribution="1/n")
```

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

```
sage: -10 <= ZZ.random_element(-10, 10) < 10
True
sage: prob = lambda x : 1/20
sage: dic = defaultdict(Integer)
sage: counter = 0.0
sage: add_samples(-10, 10)
sage: while any(abs(dic[i]/counter - prob(i)) > 0.01 for i in dic):
....:     add_samples(-10, 10)

sage: 0 <= ZZ.random_element(5) < 5
True
sage: prob = lambda x : 1/5
sage: dic = defaultdict(Integer)
sage: counter = 0.0
sage: add_samples(5)
sage: while any(abs(dic[i]/counter - prob(i)) > 0.01 for i in dic):
....:     add_samples(5)

sage: while ZZ.random_element(10^50) < 10^49:
....:     pass
```

Notice that the right endpoint is not included:

```
sage: all(ZZ.random_element(-2, 2) < 2 for _ in range(100))
True
```

We return a sample from a discrete Gaussian distribution:

```
sage: ZZ.random_element(11.0, distribution="gaussian").parent() is ZZ
True
```

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, 4835703278458516698824704,␣
↪6044629098073145873530880, 7253554917687775048237056,␣
↪8462480737302404222943232]
```

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*, *names=None*)
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
- `names` - ignored (for compatibility with number fields)

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
```

(continues on next page)

(continued from previous page)

```
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
```

valuation(*p*)

Return the discrete valuation with uniformizer p.

EXAMPLES:

```
sage: v = ZZ.valuation(3); v
3-adic valuation
sage: v(3)
1
```

See also:

Order.valuation(), *RationalField.valuation()*

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**

EXAMPLES:

```
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
```

(continues on next page)

(continued from previous page)

```
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 $\mathbf{Z}$ of integers.

1.2 Elements of the ring $\mathbb{Z}$ of integers

Sage has highly optimized and extensive functionality for arithmetic with integers and the ring of integers.

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: [2,3] * 4
[2, 3, 2, 3, 2, 3, 2, 3]
```

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

COERCIONS:

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

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

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)
- Vincent Delecroix (2017-05-03): faster integer-rational comparisons
- Vincent Klein (2017-05-11): add __mpz__() to class Integer
- Vincent Klein (2017-05-22): Integer constructor support gmpy2.mpz parameter
- Samuel Lelièvre (2018-08-02): document that divisors are sorted (trac ticket #25983)

`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)
<class '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)
<class '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
```

Conversion from gmpy2:

```
sage: from gmpy2 import mpz
sage: Integer(mpz(3))
3
```

__pow__(*left*, *right*, *modulus*)
Return `(left ^ right) % modulus`.

EXAMPLES:

```
sage: 2^-6
1/64
sage: 2^6
64
sage: 2^0
1
sage: 2^-0
1
sage: (-1)^(1/3)
(-1)^(1/3)
```

For consistency with Python and MPFR, 0^0 is defined to be 1 in Sage:

```
sage: 0^0
1
```

See also http://www.faqs.org/faqs/sci-math-faq/0to0/ and https://math.stackexchange.com/questions/11150/zero-to-the-zero-power-is-00-1.

The base need not be a Sage integer. If it is a Python type, the result is a Python type too:

```
sage: r = int(2) ^ 10; r; type(r)
1024
<... 'int'>
sage: r = int(3) ^ -3; r; type(r)
0.037037037037037035
<... 'float'>
sage: r = float(2.5) ^ 10; r; type(r)
9536.7431640625
<... 'float'>
```

We raise 2 to various interesting exponents:

```
sage: 2^x               # symbolic x
2^x
sage: 2^1.5             # real number
2.82842712474619
sage: 2^float(1.5)      # python float  abs tol 3e-16
2.8284271247461903
sage: 2^I               # complex number
2^I
sage: r = 2 ^ int(-3); r; type(r)
1/8
<class 'sage.rings.rational.Rational'>
```

(continues on next page)

(continued from previous page)

```
sage: f = 2^(sin(x)-cos(x)); f
2^(-cos(x) + sin(x))
sage: f(x=3)
2^(-cos(3) + sin(3))
```

A symbolic sum:

```
sage: x,y,z = var('x,y,z')
sage: 2^(x+y+z)
2^(x + y + z)
sage: 2^(1/2)
sqrt(2)
sage: 2^(-1/2)
1/2*sqrt(2)
```

additive_order()

Return the additive order of self.

EXAMPLES:

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

as_integer_ratio()

Return the pair `(self.numerator(), self.denominator())`, which is `(self, 1)`.

EXAMPLES:

```
sage: x = -12
sage: x.as_integer_ratio()
(-12, 1)
```

balanced_digits(*base=10*, *positive_shift=True*)

Return the list of balanced digits for `self` in the given base.

The balanced base `b` uses `b` digits centered around zero. Thus if `b` is odd, there is only one possibility, namely digits between `-b//2` and `b//2` (both included). For instance in base 9, one uses digits from -4 to 4. If `b` is even, one has to choose between digits from `-b//2` to `b//2 - 1` or `-b//2 + 1` to `b//2` (base 10 for instance: either -5 to 4 or -4 to 5), and this is defined by the value of `positive_shift`.

INPUT:

- `base` – integer (default: 10); when `base` is 2, only the nonnegative or the nonpositive integers can be represented by `balanced_digits`. Thus we say base must be greater than 2.
- `positive_shift` – boolean (default: True); for even bases, the representation uses digits from `-b//2 + 1` to `b//2` if set to True, and from `-b//2` to `b//2 - 1` otherwise. This has no effect for odd bases.

EXAMPLES:

```
sage: 8.balanced_digits(3)
[-1, 0, 1]
sage: (-15).balanced_digits(5)
[0, 2, -1]
```

(continues on next page)

(continued from previous page)

```
sage: 17.balanced_digits(6)
[-1, 3]
sage: 17.balanced_digits(6, positive_shift=False)
[-1, -3, 1]
sage: (-46).balanced_digits()
[4, 5, -1]
sage: (-46).balanced_digits(positive_shift=False)
[4, -5]
sage: (-23).balanced_digits(12)
[1, -2]
sage: (-23).balanced_digits(12, positive_shift=False)
[1, -2]
sage: 0.balanced_digits(7)
[]
sage: 14.balanced_digits(5.8)
Traceback (most recent call last):
...
ValueError: base must be an integer
sage: 14.balanced_digits(2)
Traceback (most recent call last):
...
ValueError: base must be > 2
```

See also:

digits

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())
11011010111011000111100011100100101001110100011010100011111111000101000000000010111100001000001
```

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

Return the binomial coefficient "self choose m".

INPUT:

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

OUTPUT:

- integer

EXAMPLES:

```
sage: 10.binomial(2)
45
```

(continues on next page)

(continued from previous page)

```
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
```

bit_length()

Return the number of bits required to represent this integer.

Identical to `int.bit_length()`.

EXAMPLES:

```
sage: 500.bit_length()
9
sage: 5.bit_length()
3
sage: 0.bit_length() == len(0.bits()) == 0.ndigits(base=2)
True
sage: 12345.bit_length() == len(12345.binary())
True
sage: 1023.bit_length()
10
sage: 1024.bit_length()
11
```

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:

- *bit_length()*, a faster way to compute `len(x.bits())`
- *binary()*, which returns a string in perhaps 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*)

Return 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 discriminants 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)]
```

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 non-negative integers $< m$ that are coprime to this integer.

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, 40, 41, 43, 46, 47, 49, 50, 52, 53, 56, 58, 59, 61, 62, 64, 65,␣
↪67, 68, 70, 71, 73, 74, 76, 79, 80, 82, 83, 85, 86, 89, 91, 92, 94, 95, 97,␣
↪98]
```

AUTHORS:

- Naqi Jaffery (2006-01-24): examples
- David Roe (2017-10-02): Use sieving
- Jeroen Demeyer (2018-06-25): allow returning zero (only relevant for 1.coprime_integers(n))

ALGORITHM:

Create an integer with m bits and set bits at every multiple of a prime p that divides this integer and is less than m. Then return a list of integers corresponding to the unset bits.

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), base)
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 = list(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(*method=None*)
Return the list of all positive integer divisors of this integer, sorted in increasing order.

EXAMPLES:

```
sage: (-3).divisors()
[1, 3]
sage: 6.divisors()
[1, 2, 3, 6]
sage: 28.divisors()
[1, 2, 4, 7, 14, 28]
sage: (2^5).divisors()
[1, 2, 4, 8, 16, 32]
sage: 100.divisors()
[1, 2, 4, 5, 10, 20, 25, 50, 100]
sage: 1.divisors()
[1]
sage: 0.divisors()
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: n must be nonzero
sage: (2^3 * 3^2 * 17).divisors()
[1, 2, 3, 4, 6, 8, 9, 12, 17, 18, 24, 34, 36, 51, 68, 72,
102, 136, 153, 204, 306, 408, 612, 1224]
sage: a = odd_part(factorial(31))
sage: v = a.divisors()
sage: 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
```

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 an 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 provable.
sage: n=90153710570912709517902579010793251709257901270941709247901209742124;
↪m=213509721309572
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^
↪25024207011349079210459585279553675697932183658421565260323592409432707306554163224876110094
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 kash)
  - `'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 * 418356405837450192658313794638158223810946522 31
```

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
100000000000000000039 * 1000000000000000000117
```

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 * 1000000000000000000117
```

factorial()

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

If the input does not fit in an `unsigned long int` an `OverflowError` is raised.

EXAMPLES:

```
sage: for n in srange(7):
....:     print("{} {}".format(n, n.factorial()))
```

(continues on next page)

(continued from previous page)

```
0 1
1 1
2 2
3 6
4 24
5 120
6 720
```

Large integers raise an `OverflowError`:

```
sage: (2**64).factorial()
Traceback (most recent call last):
...
OverflowError: argument too large for factorial
```

And negative ones a `ValueError`:

```
sage: (-1).factorial()
Traceback (most recent call last):
...
ValueError: factorial only defined for non-negative integers
```

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
```

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
```

hex()

Return the hexadecimal digits of `self` in lower case.

Note: '0x' is *not* prepended to the result like is done by the corresponding Python function on `int` or `long`. This is for efficiency sake–adding and stripping the string wastes time; since this function is used for conversions from integers to other C-library structures, it is important that it be fast.

EXAMPLES:

```
sage: print(Integer(15).hex())
f
sage: print(Integer(16).hex())
10
sage: print(Integer(16938402384092843092843098243).hex())
36bb1e3929d1a8fe2802f083
```

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 of Mod(189, 1890) does not exist
sage: a = Integer(19)**100000  # long time
sage: c = a.inverse_mod(a*a)   # long time
Traceback (most recent call last):
...
ZeroDivisionError: inverse of Mod(..., ...) does not exist
```

We check that trac ticket #10625 is fixed:

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

We check that trac ticket #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):
...
ArithmeticError: inverse does not exist
sage: 0.inverse_of_unit()
Traceback (most recent call last):
...
ArithmeticError: 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
```

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
```

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
```

(continues on next page)

(continued from previous page)

```
sage: t=cputime()
sage: for a in range(2, 1000): k = b.is_perfect_power()
sage: cputime(t)      # random
0.532032999999999976
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
0.0320020000000000308
```

```
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 from the example above
sage: cputime(t)      # random
0.172011000000000036
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
```

(continues on next page)

(continued from previous page)

```
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(*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 Baillie-PSW that are not actually prime. However it is conjectured that there are infinitely many.

See Wikipedia article Baillie-PSW_primality_test

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
```

is_rational()
Return `True` as an integer is a rational number.

EXAMPLES:

```
sage: 5.is_rational()
True
```

is_square()
Returns `True` if self is a perfect square.

EXAMPLES:

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

(continues on next page)

(continued from previous page)

```
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("{} {}".format(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.

EXAMPLES:

```
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
```

(continues on next page)

(continued from previous page)

```
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 argument). 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)
3.0000000000000
sage: log(Integer(125))
3*log(5)
```

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.
↪000000000000
```

```
sage: ZZ(8).log(int(2))
3
```

multifactorial(k)

Compute the k-th factorial $n!^{(k)}$ of self.

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

This function uses direct call to GMP if k and n are non-negative and uses simple transformation for other cases.

EXAMPLES:

```
sage: 5.multifactorial(1)
120
sage: 5.multifactorial(2)
15
sage: 5.multifactorial(3)
10

sage: 23.multifactorial(2)
316234143225
sage: prod([1..23, step=2])
316234143225

sage: (-29).multifactorial(7)
1/2640
sage: (-3).multifactorial(5)
1/2
sage: (-9).multifactorial(3)
Traceback (most recent call last):
...
ValueError: multifactorial undefined
```

When entries are too large an `OverflowError` is raised:

```
sage: (2**64).multifactorial(2)
Traceback (most recent call last):
...
OverflowError: argument too large for multifactorial
```

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
```

(continues on next page)

(continued from previous page)

```
sage: ZZ(2).multiplicative_order()
+Infinity
```

nbits()
Alias for *bit_length()*.

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()
100000000000000000000000000000000000000000000000000151
```

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
```

next_probable_prime()

Return 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()
13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298160
sage: 0.next_probable_prime()
2
sage: 126.next_probable_prime()
127
```

(continues on next page)

(continued from previous page)

```
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
```

oct()

Return the digits of `self` in base 8.

Note: '0' (or '0o') is *not* prepended to the result like is done by the corresponding Python function on `int` or `long`. This is for efficiency sake–adding and stripping the string wastes time; since this function is used for conversions from integers to other C-library structures, it is important that it be fast.

EXAMPLES:

```
sage: print(Integer(800).oct())
1440
sage: print(Integer(8).oct())
10
sage: print(Integer(-50).oct())
-62
sage: print(Integer(-899).oct())
-1603
sage: print(Integer(169384023840928430928430983243).oct())
1553543616224721521770500057020 3
```

Behavior of Sage integers vs. Python integers:

```
sage: Integer(10).oct()
'12'
sage: oct(int(10))
'0o12'

sage: Integer(-23).oct()
'-27'
sage: oct(int(-23))
'-0o27'
```

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))
12252984425690655138679 6875
```

ord(p)

Return the p-adic valuation of self.

INPUT:

- p - an integer at least 2.

EXAMPLES:

```
sage: n = 60
sage: n.valuation(2)
2
sage: n.valuation(3)
1
sage: n.valuation(7)
```

(continues on next page)

(continued from previous page)

```
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'
```

p_primary_part(*p*)

Return the p-primary part of `self`.

INPUT:

- `p` – a prime integer.

OUTPUT: Largest power of `p` dividing `self`.

EXAMPLES:

```
sage: n = 40
sage: n.p_primary_part(2)
8
sage: n.p_primary_part(5)
5
sage: n.p_primary_part(7)
1
sage: n.p_primary_part(6)
Traceback (most recent call last):
...
ValueError: 6 is not a prime number
```

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'
```

(continues on next page)

(continued from previous page)

```
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
```

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`).

See also:

- *next_prime()*

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
```

prime_divisors()

Return the prime divisors of this integer, sorted in increasing order.

If this integer is negative, we do *not* include -1 among its 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]
```

(continues on next page)

(continued from previous page)

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

prime_factors()

Return the prime divisors of this integer, sorted in increasing order.

If this integer is negative, we do *not* include -1 among its 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)
```

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
```

real()

Returns the real part of self, which is self.

EXAMPLES:

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

round(*mode='away'*)

Returns the nearest integer to `self`, which is self since self is an integer.

EXAMPLES:

This example addresses trac ticket #23502:

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

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, return an exact square root; otherwise return a numerical square root, 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 (a list of length 0, 1 or 2).

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):
...
ArithmeticError: square root of 2 is not an integer
sage: (-1).sqrt(extend=False)
Traceback (most recent call last):
...
ArithmeticError: square root of -1 is not an integer
sage: Integer(144).sqrt(all=True)
[12, -12]
```

(continues on next page)

(continued from previous page)

```
sage: Integer(0).sqrt(all=True)
[0]
```

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)
```

(continues on next page)

(continued from previous page)

```
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
```

trunc()

Round this number to the nearest integer, which is self since self is an integer.

EXAMPLES:

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

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` / $p^{v_p(\text{self})}$

EXAMPLES:

```
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.

EXAMPLES:

```
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.**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)
sage: type(f)
<class 'sage.rings.integer.long_to_Z'>
sage: f(5r)
5
sage: type(f(5r))
<class 'sage.rings.integer.Integer'>
sage: 1 + 2r
3
sage: type(1 + 2r)
<class 'sage.rings.integer.Integer'>
```

This is intended 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 class '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('5')
False
```

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

Bases: sage.categories.morphism.Morphism

EXAMPLES:

```
sage: f = ZZ.coerce_map_from(int)
sage: f
Native morphism:
  From: Set of Python objects of class 'int'
  To:   Integer Ring
sage: f(1rL)
1
```

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):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: expected str...Integer found
```

1.3 Cython wrapper for bernmm library

AUTHOR:

- David Harvey (2008-06): initial version

sage.rings.bernmm.**bernmm_bern_modp**(p, k)

Computes $B_k \mod p$, where B_k is the k-th Bernoulli number.

If B_k is not p-integral, returns -1.

INPUT:

p – a prime k – non-negative integer

COMPLEXITY:

Pretty much linear in p.

EXAMPLES:

```
sage: from sage.rings.bernmm import bernmm_bern_modp

sage: bernoulli(0) % 5, bernmm_bern_modp(5, 0)
(1, 1)
sage: bernoulli(1) % 5, bernmm_bern_modp(5, 1)
(2, 2)
sage: bernoulli(2) % 5, bernmm_bern_modp(5, 2)
(1, 1)
sage: bernoulli(3) % 5, bernmm_bern_modp(5, 3)
(0, 0)
sage: bernoulli(4), bernmm_bern_modp(5, 4)
(-1/30, -1)
sage: bernoulli(18) % 5, bernmm_bern_modp(5, 18)
(4, 4)
sage: bernoulli(19) % 5, bernmm_bern_modp(5, 19)
(0, 0)

sage: p = 10000019; k = 1000
sage: bernoulli(k) % p
1972762
sage: bernmm_bern_modp(p, k)
1972762
```

sage.rings.bernmm.**bernmm_bern_rat**(*k*, *num_threads=1*)

Computes k-th Bernoulli number using a multimodular algorithm. (Wrapper for bernmm library.)

INPUT:

- k – non-negative integer
- num_threads – integer >= 1, number of threads to use

COMPLEXITY:

Pretty much quadratic in k. See the paper "A multimodular algorithm for computing Bernoulli numbers", David Harvey, 2008, for more details.

EXAMPLES:

```
sage: from sage.rings.bernmm import bernmm_bern_rat

sage: bernmm_bern_rat(0)
1
sage: bernmm_bern_rat(1)
-1/2
sage: bernmm_bern_rat(2)
1/6
sage: bernmm_bern_rat(3)
0
sage: bernmm_bern_rat(100)
-
↪94598037819122125295227433069493721872702841533066936133385696204311395415197247711/
↪33330
sage: bernmm_bern_rat(100, 3)
-
↪94598037819122125295227433069493721872702841533066936133385696204311395415197247711/
↪33330
```

1.4 Bernoulli numbers modulo p

AUTHOR:

- David Harvey (2006-07-26): initial version
- William Stein (2006-07-28): some touch up.
- David Harvey (2006-08-06): new, faster algorithm, also using faster NTL interface
- David Harvey (2007-08-31): algorithm for a single Bernoulli number mod p
- David Harvey (2008-06): added interface to bernmm, removed old code

`sage.rings.bernoulli_mod_p.bernoulli_mod_p(p)`

Return the Bernoulli numbers $B_0, B_2, ...B_{p-3}$ modulo p.

INPUT:

p – integer, a prime

OUTPUT:

list – Bernoulli numbers modulo p as a list of integers [B(0), B(2), … B(p-3)].

ALGORITHM:

Described in accompanying latex file.

PERFORMANCE:

Should be complexity $O(p \log p)$.

EXAMPLES:

Check the results against PARI's C-library implementation (that computes exact rationals) for $p = 37$:

```
sage: bernoulli_mod_p(37)
[1, 31, 16, 15, 16, 4, 17, 32, 22, 31, 15, 15, 17, 12, 29, 2, 0, 2]
sage: [bernoulli(n) % 37 for n in range(0, 36, 2)]
[1, 31, 16, 15, 16, 4, 17, 32, 22, 31, 15, 15, 17, 12, 29, 2, 0, 2]
```

Boundary case:

```
sage: bernoulli_mod_p(3)
 [1]
```

AUTHOR:

– David Harvey (2006-08-06)

sage.rings.bernoulli_mod_p.**bernoulli_mod_p_single**(p, k)

Return the Bernoulli number B_k mod p.

If B_k is not p-integral, an ArithmeticError is raised.

INPUT:

- p – integer, a prime
- k – non-negative integer

OUTPUT:

The k-th Bernoulli number mod p.

EXAMPLES:

```
sage: bernoulli_mod_p_single(1009, 48)
628
sage: bernoulli(48) % 1009
628

sage: bernoulli_mod_p_single(1, 5)
Traceback (most recent call last):
...
ValueError: p (=1) must be a prime >= 3

sage: bernoulli_mod_p_single(100, 4)
Traceback (most recent call last):
...
ValueError: p (=100) must be a prime

sage: bernoulli_mod_p_single(19, 5)
0

sage: bernoulli_mod_p_single(19, 18)
Traceback (most recent call last):
...
ArithmeticError: B_k is not integral at p

sage: bernoulli_mod_p_single(19, -4)
Traceback (most recent call last):
...
ValueError: k must be non-negative
```

Check results against bernoulli_mod_p:

```
sage: bernoulli_mod_p(37)
 [1, 31, 16, 15, 16, 4, 17, 32, 22, 31, 15, 15, 17, 12, 29, 2, 0, 2]
sage: [bernoulli_mod_p_single(37, n) % 37 for n in range(0, 36, 2)]
 [1, 31, 16, 15, 16, 4, 17, 32, 22, 31, 15, 15, 17, 12, 29, 2, 0, 2]

sage: bernoulli_mod_p(31)
 [1, 26, 1, 17, 1, 9, 11, 27, 14, 23, 13, 22, 14, 8, 14]
sage: [bernoulli_mod_p_single(31, n) % 31 for n in range(0, 30, 2)]
 [1, 26, 1, 17, 1, 9, 11, 27, 14, 23, 13, 22, 14, 8, 14]

sage: bernoulli_mod_p(3)
 [1]
sage: [bernoulli_mod_p_single(3, n) % 3 for n in range(0, 2, 2)]
 [1]

sage: bernoulli_mod_p(5)
 [1, 1]
sage: [bernoulli_mod_p_single(5, n) % 5 for n in range(0, 4, 2)]
 [1, 1]

sage: bernoulli_mod_p(7)
 [1, 6, 3]
sage: [bernoulli_mod_p_single(7, n) % 7 for n in range(0, 6, 2)]
 [1, 6, 3]
```

AUTHOR:

– David Harvey (2007-08-31) – David Harvey (2008-06): rewrote to use bernmm library

sage.rings.bernoulli_mod_p.**verify_bernoulli_mod_p**(*data*)

Compute checksum for Bernoulli numbers.

It checks the identity

$$\sum_{n=0}^{(p-3)/2} 2^{2n}(2n+1)B_{2n} \equiv -2 \pmod{p}$$

(see "Irregular Primes to One Million", Buhler et al)

INPUT:

data – list, same format as output of bernoulli_mod_p function

OUTPUT:

bool – True if checksum passed

EXAMPLES:

```
sage: from sage.rings.bernoulli_mod_p import verify_bernoulli_mod_p
sage: verify_bernoulli_mod_p(bernoulli_mod_p(next_prime(3)))
True
sage: verify_bernoulli_mod_p(bernoulli_mod_p(next_prime(1000)))
True
sage: verify_bernoulli_mod_p([1, 2, 4, 5, 4])
```

(continues on next page)

(continued from previous page)

```
True
sage: verify_bernoulli_mod_p([1, 2, 3, 4, 5])
False
```

This one should test that long longs are working:

```
sage: verify_bernoulli_mod_p(bernoulli_mod_p(next_prime(20000)))
True
```

AUTHOR: David Harvey

1.5 Integer factorization functions

AUTHORS:

- Andre Apitzsch (2011-01-13): initial version

`sage.rings.factorint.`**`aurifeuillian`**(*n*, *m*, *F=None*, *check=True*)

Return the Aurifeuillian factors $F_n^{\pm}(m^2 n)$.

This is based off Theorem 3 of [Bre1993].

INPUT:

- `n` – integer
- `m` – integer
- `F` – integer (default: `None`)
- `check` – boolean (default: `True`)

OUTPUT:

A list of factors.

EXAMPLES:

```
sage: from sage.rings.factorint import aurifeuillian
sage: aurifeuillian(2,2)
[5, 13]
sage: aurifeuillian(2,2^5)
[1985, 2113]
sage: aurifeuillian(5,3)
[1471, 2851]
sage: aurifeuillian(15,1)
[19231, 142111]
sage: aurifeuillian(12,3)
Traceback (most recent call last):
...
ValueError: n has to be square-free
sage: aurifeuillian(1,2)
Traceback (most recent call last):
...
ValueError: n has to be greater than 1
sage: aurifeuillian(2,0)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: m has to be positive
```

Note: There is no need to set F. It's only for increasing speed of *factor_aurifeuillian()*.

sage.rings.factorint.**factor_aurifeuillian**(*n*, *check=True*)

Return Aurifeuillian factors of n if $n = x^{(2k-1)x} \pm 1$ (where the sign is '-' if x = 1 mod 4, and '+' otherwise) else n

INPUT:

- n – integer

OUTPUT:

List of factors of n found by Aurifeuillian factorization.

EXAMPLES:

```
sage: from sage.rings.factorint import factor_aurifeuillian as fa
sage: fa(2^6+1)
[5, 13]
sage: fa(2^58+1)
[536838145, 536903681]
sage: fa(3^3+1)
[4, 1, 7]
sage: fa(5^5-1)
[4, 11, 71]
sage: prod(_) == 5^5-1
True
sage: fa(2^4+1)
[17]
sage: fa((6^2*3)^3+1)
[109, 91, 127]
```

REFERENCES:

- http://mathworld.wolfram.com/AurifeuilleanFactorization.html
- [Bre1993] Theorem 3

sage.rings.factorint.**factor_cunningham**(*m*, *proof=None*)

Return factorization of self obtained using trial division for all primes in the so called Cunningham table. This is efficient if self has some factors of type $b^n + 1$ or $b^n - 1$, with b in $\{2, 3, 5, 6, 7, 10, 11, 12\}$.

You need to install an optional package to use this method, this can be done with the following command line: `sage -i cunningham_tables`.

INPUT:

- proof – bool (default: None); whether or not to prove primality of each factor, this is only for factors not in the Cunningham table

EXAMPLES:

```
sage: from sage.rings.factorint import factor_cunningham
sage: factor_cunningham(2^257-1) # optional - cunningham_tables
535006138814359 * 1155685395246619182673033 *␣
↪37455059850181093658177663009631318139 3
sage: factor_cunningham((3^101+1)*(2^60).next_prime(),proof=False) # optional -␣
↪cunningham_tables
2^2 * 379963 * 1152921504606847009 * 10172915271987232922083093546585785077827527
```

sage.rings.factorint.**factor_trial_division**(*m*, *limit='LONG_MAX'*)

Return partial factorization of self obtained using trial division for all primes up to limit, where limit must fit in a C signed long.

INPUT:

- `limit` – integer (default: `LONG_MAX`) that fits in a C signed long

EXAMPLES:

```
sage: from sage.rings.factorint import factor_trial_division
sage: n = 92038409284239042384829034820394809238408234908 2
sage: factor_trial_division(n, 1000)
2 * 11 * 418356405837450192658313794638158223810946522 31
sage: factor_trial_division(n, 2000)
2 * 11 * 1531 * 2732569600505879769159463060993848620580970 1
```

sage.rings.factorint.**factor_using_pari**(*n*, *int_=False*, *debug_level=0*, *proof=None*)

Factor this integer using PARI.

This function returns a list of pairs, not a `Factorization` object. The first element of each pair is the factor, of type `Integer` if `int_` is `False` or `int` otherwise, the second element is the positive exponent, of type `int`.

INPUT:

- `int_` – (default: `False`), whether the factors are of type `int` instead of `Integer`
- `debug_level` – (default: 0), debug level of the call to PARI
- `proof` – (default: `None`), whether the factors are required to be proven prime; if `None`, the global default is used

OUTPUT:

A list of pairs.

EXAMPLES:

```
sage: factor(-2**72 + 3, algorithm='pari')  # indirect doctest
-1 * 83 * 131 * 294971519 * 1472414939
```

Check that PARI's debug level is properly reset (trac ticket #18792):

```
sage: alarm(0.5); factor(2^1000 - 1, verbose=5)
Traceback (most recent call last):
...
AlarmInterrupt
sage: pari.get_debug_level()
0
```

1.6 Basic arithmetic with C integers

class sage.rings.fast_arith.**arith_int**
Bases: object

gcd_int(*a*, *b*)

inverse_mod_int(*a*, *m*)

rational_recon_int(*a*, *m*)
Rational reconstruction of a modulo m.

xgcd_int(*a*, *b*)

class sage.rings.fast_arith.**arith_llong**
Bases: object

gcd_longlong(*a*, *b*)

inverse_mod_longlong(*a*, *m*)

rational_recon_longlong(*a*, *m*)
Rational reconstruction of a modulo m.

sage.rings.fast_arith.**prime_range**(*start*, *stop=None*, *algorithm=None*, *py_ints=False*)
Return a list of all primes between `start` and `stop - 1`, inclusive.

If the second argument is omitted, this returns the primes up to the first argument.

The sage command *primes()* is an alternative that uses less memory (but may be slower), because it returns an iterator, rather than building a list of the primes.

INPUT:

- `start` – integer, lower bound (default: 1)
- `stop` – integer, upper bound
- `algorithm` – optional string (default: `None`), one of:
  - `None`: Use algorithm `"pari_primes"` if `stop` <= 436273009 (approximately 4.36E8). Otherwise use algorithm `"pari_isprime"`.
  - `"pari_primes"`: Use PARI's pari:primes function to generate all primes from 2 to stop. This is fast but may crash if there is insufficient memory. Raises an error if `stop` > 436273009.
  - `"pari_isprime"`: Wrapper for `list(primes(start, stop))`. Each (odd) integer in the specified range is tested for primality by applying PARI's pari:isprime function. This is slower but will work for much larger input.
- `py_ints` – optional boolean (default `False`), return Python ints rather than Sage Integers (faster). Ignored unless algorithm `"pari_primes"` is being used.

EXAMPLES:

```
sage: prime_range(10)
[2, 3, 5, 7]
sage: prime_range(7)
[2, 3, 5]
sage: prime_range(2000,2020)
[2003, 2011, 2017]
sage: prime_range(2,2)
[]
```

(continues on next page)

(continued from previous page)

```
sage: prime_range(2,3)
[2]
sage: prime_range(5,10)
[5, 7]
sage: prime_range(-100,10,"pari_isprime")
[2, 3, 5, 7]
sage: prime_range(2,2,algorithm="pari_isprime")
[]
sage: prime_range(10**16,10**16+100,"pari_isprime")
[10000000000000061, 10000000000000069, 10000000000000079, 10000000000000099]
sage: prime_range(10**30,10**30+100,"pari_isprime")
[1000000000000000000000000000057, 1000000000000000000000000000099]
sage: type(prime_range(8)[0])
<class 'sage.rings.integer.Integer'>
sage: type(prime_range(8,algorithm="pari_isprime")[0])
<class 'sage.rings.integer.Integer'>
```

Note: `start` and `stop` should be integers, but real numbers will also be accepted as input. In this case, they will be rounded to nearby integers start* and stop*, so the output will be the primes between start* and stop* - 1, which may not be exactly the same as the primes between `start` and `stop - 1`.

AUTHORS:

- William Stein (original version)
- Craig Citro (rewrote for massive speedup)
- Kevin Stueve (added primes iterator option) 2010-10-16
- Robert Bradshaw (speedup using Pari prime table, py_ints option)

1.7 Fast decomposition of small integers into sums of squares

Implement fast version of decomposition of (small) integers into sum of squares by direct method not relying on factorisation.

AUTHORS:

- Vincent Delecroix (2014): first implementation (trac ticket #16374)

`sage.rings.sum_of_squares.`**`four_squares_pyx`**`(`*`n`*`)`

Return a 4-tuple of non-negative integers `(i,j,k,l)` such that $i^2 + j^2 + k^2 + l^2 = n$ and $i \leq j \leq k \leq l$.

The input must be lesser than $2^{32} = 4294967296$, otherwise an `OverflowError` is raised.

See also:

`four_squares()` is much more suited for large input

EXAMPLES:

```
sage: from sage.rings.sum_of_squares import four_squares_pyx
sage: four_squares_pyx(15447)
(2, 5, 17, 123)
```

(continues on next page)

(continued from previous page)

```
sage: 2^2 + 5^2 + 17^2 + 123^2
15447

sage: four_squares_pyx(523439)
(3, 5, 26, 723)
sage: 3^2 + 5^2 + 26^2 + 723^2
523439

sage: four_squares_pyx(2**32)
Traceback (most recent call last):
...
OverflowError: ...
```

sage.rings.sum_of_squares.**is_sum_of_two_squares_pyx**(n)

Return `True` if `n` is a sum of two squares and `False` otherwise.

The input must be smaller than $2^{32} = 4294967296$, otherwise an `OverflowError` is raised.

EXAMPLES:

```
sage: from sage.rings.sum_of_squares import is_sum_of_two_squares_pyx
sage: [x for x in range(30) if is_sum_of_two_squares_pyx(x)]
[0, 1, 2, 4, 5, 8, 9, 10, 13, 16, 17, 18, 20, 25, 26, 29]

sage: is_sum_of_two_squares_pyx(2**32)
Traceback (most recent call last):
...
OverflowError: ...
```

sage.rings.sum_of_squares.**three_squares_pyx**(n)

If `n` is a sum of three squares return a 3-tuple `(i,j,k)` of Sage integers such that $i^2+j^2+k^2 = n$ and $i \leq j \leq k$. Otherwise raise a `ValueError`.

The input must be lesser than $2^{32} = 4294967296$, otherwise an `OverflowError` is raised.

EXAMPLES:

```
sage: from sage.rings.sum_of_squares import three_squares_pyx
sage: three_squares_pyx(0)
(0, 0, 0)
sage: three_squares_pyx(1)
(0, 0, 1)
sage: three_squares_pyx(2)
(0, 1, 1)
sage: three_squares_pyx(3)
(1, 1, 1)
sage: three_squares_pyx(4)
(0, 0, 2)
sage: three_squares_pyx(5)
(0, 1, 2)
sage: three_squares_pyx(6)
(1, 1, 2)
sage: three_squares_pyx(7)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: 7 is not a sum of 3 squares
sage: three_squares_pyx(107)
(1, 5, 9)

sage: three_squares_pyx(2**32)
Traceback (most recent call last):
...
OverflowError: ...
```

sage.rings.sum_of_squares.**two_squares_pyx**(*n*)

Return a pair of non-negative integers `(i,j)` such that $i^2 + j^2 = n$.

If `n` is not a sum of two squares, a `ValueError` is raised. The input must be lesser than $2^{32} = 4294967296$, otherwise an `OverflowError` is raised.

See also:

`two_squares()` is much more suited for large inputs

EXAMPLES:

```
sage: from sage.rings.sum_of_squares import two_squares_pyx
sage: two_squares_pyx(0)
(0, 0)
sage: two_squares_pyx(1)
(0, 1)
sage: two_squares_pyx(2)
(1, 1)
sage: two_squares_pyx(3)
Traceback (most recent call last):
...
ValueError: 3 is not a sum of 2 squares
sage: two_squares_pyx(106)
(5, 9)

sage: two_squares_pyx(2**32)
Traceback (most recent call last):
...
OverflowError: ...
```

1.8 Fast Arithmetic Functions

sage.arith.functions.**LCM_list**(*v*)

Return the LCM of an iterable `v`.

Elements of `v` are converted to Sage objects if they aren't already.

This function is used, e.g., by *lcm()*.

INPUT:

- `v` – an iterable

OUTPUT: integer

EXAMPLES:

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

The inputs are converted to Sage integers:

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

sage.arith.functions.**lcm**(*a*, *b=None*)

The least common multiple of a and b, or if a is a list and b is omitted the least common multiple of all elements of a.

Note that LCM is an alias for lcm.

INPUT:

- a,b – two elements of a ring with lcm or
- a – a list or tuple of elements of a ring with lcm

OUTPUT:

First, the given elements are coerced into a common parent. Then, their least common multiple *in that parent* is returned.

EXAMPLES:

```
sage: lcm(97,100)
9700
sage: LCM(97,100)
9700
sage: LCM(0,2)
0
sage: LCM(-3,-5)
15
sage: LCM([1,2,3,4,5])
60
sage: v = LCM(range(1,10000))    # *very* fast!
sage: len(str(v))
4349
```

1.9 Generic implementation of powering

This implements powering of arbitrary objects using a square-and-multiply algorithm.

sage.arith.power.**generic_power**(a, n)

Return a^n.

If n is negative, return $(1/a)^(- n)$.

INPUT:

- a – any object supporting multiplication (and division if n < 0)
- n – any integer (in the duck typing sense)

EXAMPLES:

```
sage: from sage.arith.power import generic_power
sage: generic_power(int(12), int(0))
1
sage: generic_power(int(0), int(100))
0
sage: generic_power(Integer(10), Integer(0))
1
sage: generic_power(Integer(0), Integer(23))
0
sage: sum([generic_power(2,i) for i in range(17)]) #test all 4-bit combinations
131071
sage: F = Zmod(5)
sage: a = generic_power(F(2), 5); a
2
sage: a.parent() is F
True
sage: a = generic_power(F(1), 2)
sage: a.parent() is F
True

sage: generic_power(int(5), 0)
1
sage: generic_power(2, 5/4)
Traceback (most recent call last):
...
NotImplementedError: non-integral exponents not supported
```

```
sage: class SymbolicMul(str):
....:     def __mul__(self, other):
....:         s = "({}*{})".format(self, other)
....:         return type(self)(s)
sage: x = SymbolicMul("x")
sage: print(generic_power(x, 7))
(((x*x)*(x*x))*((x*x)*x))
```

1.10 Utility classes for multi-modular algorithms

class sage.arith.multi_modular.**MultiModularBasis**
Bases: *sage.arith.multi_modular.MultiModularBasis_base*

Class used for storing a MultiModular bases of a fixed length.

class sage.arith.multi_modular.**MultiModularBasis_base**
Bases: object

This class stores a list of machine-sized prime numbers, and can do reduction and Chinese Remainder Theorem lifting modulo these primes.

Lifting implemented via Garner's algorithm, which has the advantage that all reductions are word-sized. For each i, precompute $\prod_j = 1^{i-1} m_j$ and $\prod_j = 1^{i-1} m_j^{-1} (mod m_i)$.

This class can be initialized in two ways, either with a list of prime moduli or an upper bound for the product of the prime moduli. The prime moduli are generated automatically in the second case.

EXAMPLES:

```
sage: from sage.arith.multi_modular import MultiModularBasis_base
sage: mm = MultiModularBasis_base([3, 5, 7]); mm
MultiModularBasis with moduli [3, 5, 7]

sage: height = 52348798724
sage: mm = MultiModularBasis_base(height); mm
MultiModularBasis with moduli [...]
sage: mm.prod() >= 2*height
True
```

crt(b)
Calculate lift mod $\prod_{i=0}^{len(b)-1} m_i$.

In the case that offset > 0, z[j] remains unchanged mod $\prod_{i=0}^{offset-1} m_i$

INPUT:

- b - a list of length at most self.n

OUTPUT:

Integer z where $z = b[i] mod m_i$ for 0 <= i < len(b)

EXAMPLES:

```
sage: from sage.arith.multi_modular import MultiModularBasis_base
sage: mm = MultiModularBasis_base([10007, 10009, 10037, 10039, 17351])
sage: res = mm.crt([3,5,7,9]); res
8474803647063985
sage: res % 10007
3
sage: res % 10009
5
sage: res % 10037
7
sage: res % 10039
9
```

extend_with_primes(*plist*, *partial_products=None*, *check=True*)
Extend the stored list of moduli with the given primes in `plist`.

EXAMPLES:

```
sage: from sage.arith.multi_modular import MultiModularBasis_base
sage: mm = MultiModularBasis_base([1009, 10007]); mm
MultiModularBasis with moduli [1009, 10007]
sage: mm.extend_with_primes([10037, 10039])
4
sage: mm
MultiModularBasis with moduli [1009, 10007, 10037, 10039]
```

list()
Return a list with the prime moduli.

EXAMPLES:

```
sage: from sage.arith.multi_modular import MultiModularBasis_base
sage: mm = MultiModularBasis_base([46307, 10007])
sage: mm.list()
[46307, 10007]
```

partial_product(*n*)
Return a list containing precomputed partial products.

EXAMPLES:

```
sage: from sage.arith.multi_modular import MultiModularBasis_base
sage: mm = MultiModularBasis_base([46307, 10007]); mm
MultiModularBasis with moduli [46307, 10007]
sage: mm.partial_product(0)
46307
sage: mm.partial_product(1)
463394149
```

precomputation_list()
Return a list of the precomputed coefficients $\prod_j = 1^{i-1} m_j^{-1} (mod m_i)$ where m_i are the prime moduli.

EXAMPLES:

```
sage: from sage.arith.multi_modular import MultiModularBasis_base
sage: mm = MultiModularBasis_base([46307, 10007]); mm
MultiModularBasis with moduli [46307, 10007]
sage: mm.precomputation_list()
[1, 4013]
```

prod()
Return the product of the prime moduli.

EXAMPLES:

```
sage: from sage.arith.multi_modular import MultiModularBasis_base
sage: mm = MultiModularBasis_base([46307]); mm
MultiModularBasis with moduli [46307]
sage: mm.prod()
46307
```

(continues on next page)

(continued from previous page)

```
sage: mm = MultiModularBasis_base([46307, 10007]); mm
MultiModularBasis with moduli [46307, 10007]
sage: mm.prod()
463394149
```

class sage.arith.multi_modular.**MutableMultiModularBasis**

Bases: *sage.arith.multi_modular.MultiModularBasis*

Class used for performing multi-modular methods, with the possibility of removing bad primes.

next_prime()

Pick a new random prime between the bounds given during the initialization of this object, update the precomputed data, and return the new prime modulus.

EXAMPLES:

```
sage: from sage.arith.multi_modular import MutableMultiModularBasis
sage: mm = MutableMultiModularBasis([10007])
sage: p = mm.next_prime()
sage: 1024 < p < 32768
True
sage: p != 10007
True
sage: mm.list() == [10007, p]
True
```

replace_prime(*ix*)

Replace the prime moduli at the given index with a different one, update the precomputed data accordingly, and return the new prime.

INPUT:

- `ix` – index into list of moduli

OUTPUT: the new prime modulus

EXAMPLES:

```
sage: from sage.arith.multi_modular import MutableMultiModularBasis
sage: mm = MutableMultiModularBasis([10007, 10009, 10037, 10039])
sage: mm
MultiModularBasis with moduli [10007, 10009, 10037, 10039]
sage: prev_prod = mm.prod(); prev_prod
10092272478850909
sage: mm.precomputation_list()
[1, 5004, 6536, 6060]
sage: mm.partial_product(2)
1005306552331
sage: p = mm.replace_prime(1)
sage: mm.list() == [10007, p, 10037, 10039]
True
sage: mm.prod()*10009 == prev_prod*p
True
sage: precomputed = mm.precomputation_list()
sage: precomputed == [prod(Integers(mm[i])(1 / mm[j])
....:                      for j in range(i))
```

(continues on next page)

(continued from previous page)

```
....:                 for i in range(4)]
True
sage: mm.partial_product(2) == prod(mm.list()[:3])
True
```

1.11 Miscellaneous arithmetic functions

sage.arith.misc.**CRT**(*a*, *b*, *m=None*, *n=None*)

Return a solution to a Chinese Remainder Theorem problem.

INPUT:

- a, b - two residues (elements of some ring for which extended gcd is available), or two lists, one of residues and one of moduli.
- m, n - (default: None) two moduli, or None.

OUTPUT:

If m, n are not None, returns a solution x to the simultaneous congruences $x \equiv a \bmod m$ and $x \equiv b \bmod n$, if one exists. By the Chinese Remainder Theorem, a solution to the simultaneous congruences exists if and only if $a \equiv b \pmod{\gcd(m,n)}$. The solution x is only well-defined modulo $\text{lcm}(m,n)$.

If a and b are lists, returns a simultaneous solution to the congruences $x \equiv a_i \pmod{b_i}$, if one exists.

See also:

- *CRT_list()*

EXAMPLES:

Using crt by giving it pairs of residues and moduli:

```
sage: crt(2, 1, 3, 5)
11
sage: crt(13, 20, 100, 301)
28013
sage: crt([2, 1], [3, 5])
11
sage: crt([13, 20], [100, 301])
28013
```

You can also use upper case:

```
sage: c = CRT(2,3, 3, 5); c
8
sage: c % 3 == 2
True
sage: c % 5 == 3
True
```

Note that this also works for polynomial rings:

```
sage: K.<a> = NumberField(x^3 - 7)
sage: R.<y> = K[]
sage: f = y^2 + 3
sage: g = y^3 - 5
sage: CRT(1,3,f,g)
-3/26*y^4 + 5/26*y^3 + 15/26*y + 53/26
sage: CRT(1,a,f,g)
(-3/52*a + 3/52)*y^4 + (5/52*a - 5/52)*y^3 + (15/52*a - 15/52)*y + 27/52*a + 25/52
```

You can also do this for any number of moduli:

```
sage: K.<a> = NumberField(x^3 - 7)
sage: R.<x> = K[]
sage: CRT([], [])
0
sage: CRT([a], [x])
a
sage: f = x^2 + 3
sage: g = x^3 - 5
sage: h = x^5 + x^2 - 9
sage: k = CRT([1, a, 3], [f, g, h]); k
(127/26988*a - 5807/386828)*x^9 + (45/8996*a - 33677/1160484)*x^8 + (2/173*a - 6/
↪173)*x^7 + (133/6747*a - 5373/96707)*x^6 + (-6/2249*a + 18584/290121)*x^5 + (-277/
↪8996*a + 38847/386828)*x^4 + (-135/4498*a + 42673/193414)*x^3 + (-1005/8996*a +␣
↪470245/1160484)*x^2 + (-1215/8996*a + 141165/386828)*x + 621/8996*a + 836445/
↪386828
sage: k.mod(f)
1
sage: k.mod(g)
a
sage: k.mod(h)
3
```

If the moduli are not coprime, a solution may not exist:

```
sage: crt(4,8,8,12)
20
sage: crt(4,6,8,12)
Traceback (most recent call last):
...
ValueError: no solution to crt problem since gcd(8,12) does not divide 4-6

sage: x = polygen(QQ)
sage: crt(2,3,x-1,x+1)
-1/2*x + 5/2
sage: crt(2,x,x^2-1,x^2+1)
-1/2*x^3 + x^2 + 1/2*x + 1
sage: crt(2,x,x^2-1,x^3-1)
Traceback (most recent call last):
...
ValueError: no solution to crt problem since gcd(x^2 - 1,x^3 - 1) does not divide 2-
↪x
```

(continues on next page)

(continued from previous page)

```
sage: crt(int(2), int(3), int(7), int(11))
58
```

crt also work with numpy and gmpy2 numbers:

```
sage: import numpy
sage: crt(numpy.int8(2), numpy.int8(3), numpy.int8(7), numpy.int8(11))
58
sage: from gmpy2 import mpz
sage: crt(mpz(2), mpz(3), mpz(7), mpz(11))
58
sage: crt(mpz(2), 3, mpz(7), numpy.int8(11))
58
```

sage.arith.misc.**CRT_basis**(*moduli*)

Return a CRT basis for the given moduli.

INPUT:

- **moduli - list of pairwise coprime moduli m which admit an** extended Euclidean algorithm

OUTPUT:

- a list of elements a_i of the same length as m such that a_i is congruent to 1 modulo m_i and to 0 modulo m_j for $j \neq i$.

Note: The pairwise coprimality of the input is not checked.

EXAMPLES:

```
sage: a1 = ZZ(mod(42,5))
sage: a2 = ZZ(mod(42,13))
sage: c1,c2 = CRT_basis([5,13])
sage: mod(a1*c1+a2*c2,5*13)
42
```

A polynomial example:

```
sage: x=polygen(QQ)
sage: mods = [x,x^2+1,2*x-3]
sage: b = CRT_basis(mods)
sage: b
[-2/3*x^3 + x^2 - 2/3*x + 1, 6/13*x^3 - x^2 + 6/13*x, 8/39*x^3 + 8/39*x]
sage: [[bi % mj for mj in mods] for bi in b]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

sage.arith.misc.**CRT_list**(*v*, *moduli*)

Given a list `v` of elements and a list of corresponding `moduli`, find a single element that reduces to each element of `v` modulo the corresponding moduli.

See also:

- *crt()*

EXAMPLES:

```
sage: CRT_list([2,3,2], [3,5,7])
23
sage: x = polygen(QQ)
sage: c = CRT_list([3], [x]); c
3
sage: c.parent()
Univariate Polynomial Ring in x over Rational Field
```

It also works if the moduli are not coprime:

```
sage: CRT_list([32,2,2],[60,90,150])
452
```

But with non coprime moduli there is not always a solution:

```
sage: CRT_list([32,2,1],[60,90,150])
Traceback (most recent call last):
...
ValueError: no solution to crt problem since gcd(180,150) does not divide 92-1
```

The arguments must be lists:

```
sage: CRT_list([1,2,3],"not a list")
Traceback (most recent call last):
...
ValueError: arguments to CRT_list should be lists
sage: CRT_list("not a list",[2,3])
Traceback (most recent call last):
...
ValueError: arguments to CRT_list should be lists
```

The list of moduli must have the same length as the list of elements:

```
sage: CRT_list([1,2,3],[2,3,5])
23
sage: CRT_list([1,2,3],[2,3])
Traceback (most recent call last):
...
ValueError: arguments to CRT_list should be lists of the same length
sage: CRT_list([1,2,3],[2,3,5,7])
Traceback (most recent call last):
...
ValueError: arguments to CRT_list should be lists of the same length
```

sage.arith.misc.**CRT_vectors**(*X*, *moduli*)

Vector form of the Chinese Remainder Theorem: given a list of integer vectors v_i and a list of coprime moduli m_i, find a vector w such that $w = v_i \pmod{m}_i$ for all i. This is more efficient than applying *CRT()* to each entry.

INPUT:

- `X` - list or tuple, consisting of lists/tuples/vectors/etc of integers of the same length
- `moduli` - list of len(X) moduli

OUTPUT:

- `list` - application of CRT componentwise.

EXAMPLES:

```
sage: CRT_vectors([[3,5,7],[3,5,11]], [2,3])
[3, 5, 5]

sage: CRT_vectors([vector(ZZ, [2,3,1]), Sequence([1,7,8],ZZ)], [8,9])
[10, 43, 17]
```

class sage.arith.misc.**Euler_Phi**

Bases: `object`

Return the value of the Euler phi function on the integer n. We defined this to be the number of positive integers <= n that are relatively prime to n. Thus if n<=0 then `euler_phi(n)` is defined and equals 0.

INPUT:

- `n` - an integer

EXAMPLES:

```
sage: euler_phi(1)
1
sage: euler_phi(2)
1
sage: euler_phi(3)
2
sage: euler_phi(12)
4
sage: euler_phi(37)
36
```

Notice that euler_phi is defined to be 0 on negative numbers and 0.

```
sage: euler_phi(-1)
0
sage: euler_phi(0)
0
sage: type(euler_phi(0))
<class 'sage.rings.integer.Integer'>
```

We verify directly that the phi function is correct for 21.

```
sage: euler_phi(21)
12
sage: [i for i in range(21) if gcd(21,i) == 1]
[1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20]
```

The length of the list of integers 'i' in range(n) such that the gcd(i,n) == 1 equals euler_phi(n).

```
sage: len([i for i in range(21) if gcd(21,i) == 1]) == euler_phi(21)
True
```

The phi function also has a special plotting method.

```
sage: P = plot(euler_phi, -3, 71)
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: euler_phi(int8(37))
36
sage: from gmpy2 import mpz
sage: euler_phi(mpz(37))
36
```

AUTHORS:

- William Stein
- Alex Clemesha (2006-01-10): some examples

plot(*xmin=1*, *xmax=50*, *pointsize=30*, *rgbcolor=(0, 0, 1)*, *join=True*, ***kwds*)

Plot the Euler phi function.

INPUT:

- `xmin` - default: 1
- `xmax` - default: 50
- `pointsize` - default: 30
- `rgbcolor` - default: (0,0,1)
- `join` - default: True; whether to join the points.
- `**kwds` - passed on

EXAMPLES:

```
sage: from sage.arith.misc import Euler_Phi
sage: p = Euler_Phi().plot()
sage: p.ymax()
46.0
```

sage.arith.misc.**GCD**(*a*, *b=None*, ***kwargs*)

Return the greatest common divisor of `a` and `b`.

If `a` is a list and `b` is omitted, return instead the greatest common divisor of all elements of `a`.

INPUT:

- `a,b` – two elements of a ring with gcd or
- `a` – a list or tuple of elements of a ring with gcd

Additional keyword arguments are passed to the respectively called methods.

OUTPUT:

The given elements are first coerced into a common parent. Then, their greatest common divisor *in that common parent* is returned.

EXAMPLES:

```
sage: GCD(97,100)
1
sage: GCD(97*10^15, 19^20*97^2)
97
```

(continues on next page)

(continued from previous page)

```
sage: GCD(2/3, 4/5)
2/15
sage: GCD([2,4,6,8])
2
sage: GCD(srange(0,10000,10))  # fast  !!
10
```

Note that to take the gcd of n elements for $n \neq 2$ you must put the elements into a list by enclosing them in `[..]`. Before trac ticket #4988 the following wrongly returned 3 since the third parameter was just ignored:

```
sage: gcd(3, 6, 2)
Traceback (most recent call last):
...
TypeError: ...gcd() takes ...
sage: gcd([3, 6, 2])
1
```

Similarly, giving just one element (which is not a list) gives an error:

```
sage: gcd(3)
Traceback (most recent call last):
...
TypeError: 'sage.rings.integer.Integer' object is not iterable
```

By convention, the gcd of the empty list is (the integer) 0:

```
sage: gcd([])
0
sage: type(gcd([]))
<class 'sage.rings.integer.Integer'>
```

class sage.arith.misc.**Moebius**

Bases: `object`

Return the value of the Möbius function of abs(n), where n is an integer.

DEFINITION: $\mu(n)$ is 0 if n is not square free, and otherwise equals $(-1)^r$, where n has r distinct prime factors.

For simplicity, if $n = 0$ we define $\mu(n) = 0$.

IMPLEMENTATION: Factors or - for integers - uses the PARI C library.

INPUT:

- `n` - anything that can be factored.

OUTPUT: 0, 1, or -1

EXAMPLES:

```
sage: moebius(-5)
-1
sage: moebius(9)
0
sage: moebius(12)
0
sage: moebius(-35)
```

(continues on next page)

(continued from previous page)

```
1
sage: moebius(-1)
1
sage: moebius(7)
-1
```

```
sage: moebius(0)   # potentially nonstandard!
0
```

The moebius function even makes sense for non-integer inputs.

```
sage: x = GF(7)['x'].0
sage: moebius(x+2)
-1
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: moebius(int8(-5))
-1
sage: from gmpy2 import mpz
sage: moebius(mpz(-5))
-1
```

plot(*xmin=0*, *xmax=50*, *pointsize=30*, *rgbcolor=(0, 0, 1)*, *join=True*, ***kwds*)

Plot the Möbius function.

INPUT:

- `xmin` - default: 0
- `xmax` - default: 50
- `pointsize` - default: 30
- `rgbcolor` - default: (0,0,1)
- `join` - default: True; whether to join the points (very helpful in seeing their order).
- `**kwds` - passed on

EXAMPLES:

```
sage: from sage.arith.misc import Moebius
sage: p = Moebius().plot()
sage: p.ymax()
1.0
```

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

Return the Möbius function evaluated at the given range of values, i.e., the image of the list range(start, stop, step) under the Möbius function.

This is much faster than directly computing all these values with a list comprehension.

EXAMPLES:

```
sage: v = moebius.range(-10,10); v
[1, 0, 0, -1, 1, -1, 0, -1, -1, 1, 0, 1, -1, -1, 0, -1, 1, -1, 0, 0]
sage: v == [moebius(n) for n in range(-10,10)]
True
sage: v = moebius.range(-1000, 2000, 4)
sage: v == [moebius(n) for n in range(-1000,2000, 4)]
True
```

class sage.arith.misc.**Sigma**

Bases: `object`

Return the sum of the k-th powers of the divisors of n.

INPUT:

- `n` - integer
- `k` - integer (default: 1)

OUTPUT: integer

EXAMPLES:

```
sage: sigma(5)
6
sage: sigma(5,2)
26
```

The sigma function also has a special plotting method.

```
sage: P = plot(sigma, 1, 100)
```

This method also works with k-th powers.

```
sage: P = plot(sigma, 1, 100, k=2)
```

AUTHORS:

- William Stein: original implementation
- Craig Citro (2007-06-01): rewrote for huge speedup

plot(*xmin=1*, *xmax=50*, *k=1*, *pointsize=30*, *rgbcolor=(0, 0, 1)*, *join=True*, ***kwds*)

Plot the sigma (sum of k-th powers of divisors) function.

INPUT:

- `xmin` - default: 1
- `xmax` - default: 50
- `k` - default: 1
- `pointsize` - default: 30
- `rgbcolor` - default: (0,0,1)
- `join` - default: True; whether to join the points.
- `**kwds` - passed on

EXAMPLES:

```
sage: from sage.arith.misc import Sigma
sage: p = Sigma().plot()
sage: p.ymax()
124.0
```

sage.arith.misc.**XGCD**(a, b)

Return a triple `(g,s,t)` such that $g = s \cdot a + t \cdot b = \gcd(a, b)$.

Note: One exception is if a and b are not in a principal ideal domain (see Wikipedia article Principal_ideal_domain), e.g., they are both polynomials over the integers. Then this function can't in general return `(g,s,t)` as above, since they need not exist. Instead, over the integers, we first multiply g by a divisor of the resultant of a/g and b/g, up to sign.

INPUT:

- `a, b` - integers or more generally, element of a ring for which the xgcd make sense (e.g. a field or univariate polynomials).

OUTPUT:

- `g, s, t` - such that $g = s \cdot a + t \cdot b$

Note: There is no guarantee that the returned cofactors (s and t) are minimal.

EXAMPLES:

```
sage: xgcd(56, 44)
(4, 4, -5)
sage: 4*56 + (-5)*44
4

sage: g, a, b = xgcd(5/1, 7/1); g, a, b
(1, 3, -2)
sage: a*(5/1) + b*(7/1) == g
True

sage: x = polygen(QQ)
sage: xgcd(x^3 - 1, x^2 - 1)
(x - 1, 1, -x)

sage: K.<g> = NumberField(x^2-3)
sage: g.xgcd(g+2)
(1, 1/3*g, 0)

sage: R.<a,b> = K[]
sage: S.<y> = R.fraction_field()[]
sage: xgcd(y^2, a*y+b)
(1, a^2/b^2, ((-a)/b^2)*y + 1/b)
sage: xgcd((b+g)*y^2, (a+g)*y+b)
(1, (a^2 + (2*g)*a + 3)/(b^3 + g*b^2), ((-a + (-g))/b^2)*y + 1/b)
```

Here is an example of a xgcd for two polynomials over the integers, where the linear combination is not the gcd but the gcd multiplied by the resultant:

```
sage: R.<x> = ZZ[]
sage: gcd(2*x*(x-1), x^2)
x
sage: xgcd(2*x*(x-1), x^2)
(2*x, -1, 2)
sage: (2*(x-1)).resultant(x)
2
```

Tests with numpy and gmpy2 types:

```
sage: from numpy import int8
sage: xgcd(4,int8(8))
(4, 1, 0)
sage: xgcd(int8(4),int8(8))
(4, 1, 0)
sage: from gmpy2 import mpz
sage: xgcd(mpz(4), mpz(8))
(4, 1, 0)
sage: xgcd(4, mpz(8))
(4, 1, 0)
```

sage.arith.misc.**algdep**(*z*, *degree*, *known_bits=None*, *use_bits=None*, *known_digits=None*, *use_digits=None*, *height_bound=None*, *proof=False*)

Return an irreducible polynomial of degree at most $degree$ which is approximately satisfied by the number z.

You can specify the number of known bits or digits of z with `known_bits=k` or `known_digits=k`. PARI is then told to compute the result using $0.8k$ of these bits/digits. Or, you can specify the precision to use directly with `use_bits=k` or `use_digits=k`. If none of these are specified, then the precision is taken from the input value.

A height bound may be specified to indicate the maximum coefficient size of the returned polynomial; if a sufficiently small polynomial is not found, then `None` will be returned. If `proof=True` then the result is returned only if it can be proved correct (i.e. the only possible minimal polynomial satisfying the height bound, or no such polynomial exists). Otherwise a `ValueError` is raised indicating that higher precision is required.

ALGORITHM: Uses LLL for real/complex inputs, PARI C-library `algdep` command otherwise.

Note that `algebraic_dependency` is a synonym for `algdep`.

INPUT:

- `z` - real, complex, or p-adic number
- `degree` - an integer
- **`height_bound` - an integer (default: `None`) specifying the maximum** coefficient size for the returned polynomial
- `proof` - a boolean (default: `False`), requires height_bound to be set

EXAMPLES:

```
sage: algdep(1.888888888888888, 1)
9*x - 17
sage: algdep(0.12121212121212,1)
33*x - 4
sage: algdep(sqrt(2),2)
x^2 - 2
```

This example involves a complex number:

```
sage: z = (1/2)*(1 + RDF(sqrt(3)) *CC.0); z
0.500000000000000 + 0.866025403784439*I
sage: algdep(z, 6)
x^2 - x + 1
```

This example involves a p-adic number:

```
sage: K = Qp(3, print_mode = 'series')
sage: a = K(7/19); a
1 + 2*3 + 3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^8 + 2*3^9 + 3^11 + 3^12 + 2*3^15 + 2*3^16 +␣
↪3^17 + 2*3^19 + O(3^20)
sage: algdep(a, 1)
19*x - 7
```

These examples show the importance of proper precision control. We compute a 200-bit approximation to $sqrt(2)$ which is wrong in the 33'rd bit:

```
sage: z = sqrt(RealField(200)(2)) + (1/2)^33
sage: p = algdep(z, 4); p
227004321085*x^4 - 216947902586*x^3 - 99411220986*x^2 + 82234881648*x - 211871195088
sage: factor(p)
227004321085*x^4 - 216947902586*x^3 - 99411220986*x^2 + 82234881648*x - 211871195088
sage: algdep(z, 4, known_bits=32)
x^2 - 2
sage: algdep(z, 4, known_digits=10)
x^2 - 2
sage: algdep(z, 4, use_bits=25)
x^2 - 2
sage: algdep(z, 4, use_digits=8)
x^2 - 2
```

Using the `height_bound` and `proof` parameters, we can see that pi is not the root of an integer polynomial of degree at most 5 and coefficients bounded above by 10:

```
sage: algdep(pi.n(), 5, height_bound=10, proof=True) is None
True
```

For stronger results, we need more precision:

```
sage: algdep(pi.n(), 5, height_bound=100, proof=True) is None
Traceback (most recent call last):
...
ValueError: insufficient precision for non-existence proof
sage: algdep(pi.n(200), 5, height_bound=100, proof=True) is None
True

sage: algdep(pi.n(), 10, height_bound=10, proof=True) is None
Traceback (most recent call last):
...
ValueError: insufficient precision for non-existence proof
sage: algdep(pi.n(200), 10, height_bound=10, proof=True) is None
True
```

We can also use `proof=True` to get positive results:

```
sage: a = sqrt(2) + sqrt(3) + sqrt(5)
sage: algdep(a.n(), 8, height_bound=1000, proof=True)
Traceback (most recent call last):
...
ValueError: insufficient precision for uniqueness proof
sage: f = algdep(a.n(1000), 8, height_bound=1000, proof=True); f
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
sage: f(a).expand()
0
```

sage.arith.misc.**algebraic_dependency**(*z*, *degree*, *known_bits=None*, *use_bits=None*, *known_digits=None*, *use_digits=None*, *height_bound=None*, *proof=False*)

Return an irreducible polynomial of degree at most $degree$ which is approximately satisfied by the number z.

You can specify the number of known bits or digits of z with `known_bits=k` or `known_digits=k`. PARI is then told to compute the result using $0.8k$ of these bits/digits. Or, you can specify the precision to use directly with `use_bits=k` or `use_digits=k`. If none of these are specified, then the precision is taken from the input value.

A height bound may be specified to indicate the maximum coefficient size of the returned polynomial; if a sufficiently small polynomial is not found, then `None` will be returned. If `proof=True` then the result is returned only if it can be proved correct (i.e. the only possible minimal polynomial satisfying the height bound, or no such polynomial exists). Otherwise a `ValueError` is raised indicating that higher precision is required.

ALGORITHM: Uses LLL for real/complex inputs, PARI C-library `algdep` command otherwise.

Note that `algebraic_dependency` is a synonym for `algdep`.

INPUT:

- `z` - real, complex, or p-adic number
- `degree` - an integer
- **`height_bound` - an integer (default: `None`) specifying the maximum** coefficient size for the returned polynomial
- `proof` - a boolean (default: `False`), requires height_bound to be set

EXAMPLES:

```
sage: algdep(1.888888888888888, 1)
9*x - 17
sage: algdep(0.12121212121212,1)
33*x - 4
sage: algdep(sqrt(2),2)
x^2 - 2
```

This example involves a complex number:

```
sage: z = (1/2)*(1 + RDF(sqrt(3)) *CC.0); z
0.500000000000000 + 0.866025403784439*I
sage: algdep(z, 6)
x^2 - x + 1
```

This example involves a p-adic number:

```
sage: K = Qp(3, print_mode = 'series')
sage: a = K(7/19); a
```

(continues on next page)

(continued from previous page)

```
1 + 2*3 + 3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^8 + 2*3^9 + 3^11 + 3^12 + 2*3^15 + 2*3^16 +␣
↪3^17 + 2*3^19 + O(3^20)
sage: algdep(a, 1)
19*x - 7
```

These examples show the importance of proper precision control. We compute a 200-bit approximation to $sqrt(2)$ which is wrong in the 33'rd bit:

```
sage: z = sqrt(RealField(200)(2)) + (1/2)^33
sage: p = algdep(z, 4); p
227004321085*x^4 - 216947902586*x^3 - 99411220986*x^2 + 82234881648*x - 211871195088
sage: factor(p)
227004321085*x^4 - 216947902586*x^3 - 99411220986*x^2 + 82234881648*x - 211871195088
sage: algdep(z, 4, known_bits=32)
x^2 - 2
sage: algdep(z, 4, known_digits=10)
x^2 - 2
sage: algdep(z, 4, use_bits=25)
x^2 - 2
sage: algdep(z, 4, use_digits=8)
x^2 - 2
```

Using the `height_bound` and `proof` parameters, we can see that pi is not the root of an integer polynomial of degree at most 5 and coefficients bounded above by 10:

```
sage: algdep(pi.n(), 5, height_bound=10, proof=True) is None
True
```

For stronger results, we need more precision:

```
sage: algdep(pi.n(), 5, height_bound=100, proof=True) is None
Traceback (most recent call last):
...
ValueError: insufficient precision for non-existence proof
sage: algdep(pi.n(200), 5, height_bound=100, proof=True) is None
True

sage: algdep(pi.n(), 10, height_bound=10, proof=True) is None
Traceback (most recent call last):
...
ValueError: insufficient precision for non-existence proof
sage: algdep(pi.n(200), 10, height_bound=10, proof=True) is None
True
```

We can also use `proof=True` to get positive results:

```
sage: a = sqrt(2) + sqrt(3) + sqrt(5)
sage: algdep(a.n(), 8, height_bound=1000, proof=True)
Traceback (most recent call last):
...
ValueError: insufficient precision for uniqueness proof
sage: f = algdep(a.n(1000), 8, height_bound=1000, proof=True); f
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
```

(continues on next page)

(continued from previous page)

```
sage: f(a).expand()
0
```

sage.arith.misc.**bernoulli**(*n*, *algorithm='default'*, *num_threads=1*)

Return the n-th Bernoulli number, as a rational number.

INPUT:

- `n` - an integer
- `algorithm`:
  - `'default'` – use 'flint' for n <= 20000, then 'arb' for n <= 300000 and 'bernmm' for larger values (this is just a heuristic, and not guaranteed to be optimal on all hardware)
  - `'arb'` – use the arb library
  - `'flint'` – use the FLINT library
  - `'pari'` – use the PARI C library
  - `'gap'` – use GAP
  - `'gp'` – use PARI/GP interpreter
  - `'magma'` – use MAGMA (optional)
  - `'bernmm'` – use bernmm package (a multimodular algorithm)
- `num_threads` - positive integer, number of threads to use (only used for bernmm algorithm)

EXAMPLES:

```
sage: bernoulli(12)
-691/2730
sage: bernoulli(50)
4950572052410796482124775 25/66
```

We demonstrate each of the alternative algorithms:

```
sage: bernoulli(12, algorithm='arb')
-691/2730
sage: bernoulli(12, algorithm='flint')
-691/2730
sage: bernoulli(12, algorithm='gap')
-691/2730
sage: bernoulli(12, algorithm='gp')
-691/2730
sage: bernoulli(12, algorithm='magma')           # optional - magma
-691/2730
sage: bernoulli(12, algorithm='pari')
-691/2730
sage: bernoulli(12, algorithm='bernmm')
-691/2730
sage: bernoulli(12, algorithm='bernmm', num_threads=4)
-691/2730
```

AUTHOR:

- David Joyner and William Stein

sage.arith.misc.**binomial**(*x*, *m*, ***kwds*)

Return the binomial coefficient

$$\binom{x}{m} = x(x-1)\cdots(x-m+1)/m!$$

which is defined for $m \in \mathbf{Z}$ and any x. We extend this definition to include cases when $x - m$ is an integer but m is not by

$$\binom{x}{m} = \binom{x}{x-m}$$

If $m < 0$, return 0.

INPUT:

- `x`, `m` - numbers or symbolic expressions. Either `m` or `x-m` must be an integer.

OUTPUT: number or symbolic expression (if input is symbolic)

EXAMPLES:

```
sage: from sage.arith.misc import binomial
sage: binomial(5,2)
10
sage: binomial(2,0)
1
sage: binomial(1/2, 0)
1
sage: binomial(3,-1)
0
sage: binomial(20,10)
184756
sage: binomial(-2, 5)
-6
sage: binomial(-5, -2)
0
sage: binomial(RealField()('2.5'), 2)
1.87500000000000
sage: n=var('n'); binomial(n,2)
1/2*(n - 1)*n
sage: n=var('n'); binomial(n,n)
1
sage: n=var('n'); binomial(n,n-1)
n
sage: binomial(2^100, 2^100)
1

sage: x = polygen(ZZ)
sage: binomial(x, 3)
1/6*x^3 - 1/2*x^2 + 1/3*x
sage: binomial(x, x-3)
1/6*x^3 - 1/2*x^2 + 1/3*x
```

If $x \in \mathbf{Z}$, there is an optional 'algorithm' parameter, which can be 'gmp' (faster for small values; alias: 'mpir') or 'pari' (faster for large values):

```
sage: a = binomial(100, 45, algorithm='gmp')
sage: b = binomial(100, 45, algorithm='pari')
sage: a == b
True
```

sage.arith.misc.**binomial_coefficients**(*n*)

Return a dictionary containing pairs $\{(k_1, k_2) : C_{k,n}\}$ where C_{k_n} are binomial coefficients and $n = k_1 + k_2$.

INPUT:

- `n` - an integer

OUTPUT: dict

EXAMPLES:

```
sage: sorted(binomial_coefficients(3).items())
[((0, 3), 1), ((1, 2), 3), ((2, 1), 3), ((3, 0), 1)]
```

Notice the coefficients above are the same as below:

```
sage: R.<x,y> = QQ[]
sage: (x+y)^3
x^3 + 3*x^2*y + 3*x*y^2 + y^3
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: sorted(binomial_coefficients(int8(3)).items())
[((0, 3), 1), ((1, 2), 3), ((2, 1), 3), ((3, 0), 1)]
sage: from gmpy2 import mpz
sage: sorted(binomial_coefficients(mpz(3)).items())
[((0, 3), 1), ((1, 2), 3), ((2, 1), 3), ((3, 0), 1)]
```

AUTHORS:

- Fredrik Johansson

sage.arith.misc.**continuant**(*v*, *n=None*)

Function returns the continuant of the sequence v (list or tuple).

Definition: see Graham, Knuth and Patashnik, *Concrete Mathematics*, section 6.7: Continuants. The continuant is defined by

- $K_0() = 1$
- $K_1(x_1) = x_1$
- $K_n(x_1, \cdots, x_n) = K_{n-1}(x_n, \cdots x_{n-1})x_n + K_{n-2}(x_1, \cdots, x_{n-2})$

If `n = None` or `n > len(v)` the default `n = len(v)` is used.

INPUT:

- `v` - list or tuple of elements of a ring
- `n` - optional integer

OUTPUT: element of ring (integer, polynomial, etcetera).

EXAMPLES:

```
sage: continuant([1,2,3])
10
sage: p = continuant([2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10])
sage: q = continuant([1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10])
sage: p/q
517656/190435
sage: continued_fraction([2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10]).
↪convergent(14)
517656/190435
sage: x = PolynomialRing(RationalField(),'x',5).gens()
sage: continuant(x)
x0*x1*x2*x3*x4 + x0*x1*x2 + x0*x1*x4 + x0*x3*x4 + x2*x3*x4 + x0 + x2 + x4
sage: continuant(x, 3)
x0*x1*x2 + x0 + x2
sage: continuant(x,2)
x0*x1 + 1
```

We verify the identity

$$K_n(z, z, \cdots, z) = \sum_{k=0}^{n} \binom{n-k}{k} z^{n-2k}$$

for $n = 6$ using polynomial arithmetic:

```
sage: z = QQ['z'].0
sage: continuant((z,z,z,z,z,z,z,z,z,z,z,z,z,z,z,z),6)
z^6 + 5*z^4 + 6*z^2 + 1

sage: continuant(9)
Traceback (most recent call last):
...
TypeError: object of type 'sage.rings.integer.Integer' has no len()
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: continuant([int8(1),int8(2),int8(3)])
10
sage: from gmpy2 import mpz
sage: continuant([mpz(1),mpz(2),mpz(3)])
mpz(10)
```

AUTHORS:

- Jaap Spies (2007-02-06)

sage.arith.misc.**crt**(*a*, *b*, *m=None*, *n=None*)

Return a solution to a Chinese Remainder Theorem problem.

INPUT:

- `a`, `b` - two residues (elements of some ring for which extended gcd is available), or two lists, one of residues and one of moduli.
- `m`, `n` - (default: `None`) two moduli, or `None`.

OUTPUT:

If `m`, `n` are not `None`, returns a solution x to the simultaneous congruences $x \equiv a \bmod m$ and $x \equiv b \bmod n$, if one exists. By the Chinese Remainder Theorem, a solution to the simultaneous congruences exists if and only if $a \equiv b \pmod{\gcd(m,n)}$. The solution x is only well-defined modulo $\text{lcm}(m,n)$.

If `a` and `b` are lists, returns a simultaneous solution to the congruences $x \equiv a_i \pmod{b_i}$, if one exists.

See also:

- *CRT_list()*

EXAMPLES:

Using `crt` by giving it pairs of residues and moduli:

```
sage: crt(2, 1, 3, 5)
11
sage: crt(13, 20, 100, 301)
28013
sage: crt([2, 1], [3, 5])
11
sage: crt([13, 20], [100, 301])
28013
```

You can also use upper case:

```
sage: c = CRT(2,3, 3, 5); c
8
sage: c % 3 == 2
True
sage: c % 5 == 3
True
```

Note that this also works for polynomial rings:

```
sage: K.<a> = NumberField(x^3 - 7)
sage: R.<y> = K[]
sage: f = y^2 + 3
sage: g = y^3 - 5
sage: CRT(1,3,f,g)
-3/26*y^4 + 5/26*y^3 + 15/26*y + 53/26
sage: CRT(1,a,f,g)
(-3/52*a + 3/52)*y^4 + (5/52*a - 5/52)*y^3 + (15/52*a - 15/52)*y + 27/52*a + 25/52
```

You can also do this for any number of moduli:

```
sage: K.<a> = NumberField(x^3 - 7)
sage: R.<x> = K[]
sage: CRT([], [])
0
sage: CRT([a], [x])
a
sage: f = x^2 + 3
sage: g = x^3 - 5
sage: h = x^5 + x^2 - 9
sage: k = CRT([1, a, 3], [f, g, h]); k
```

(continues on next page)

(continued from previous page)

```
(127/26988*a - 5807/386828)*x^9 + (45/8996*a - 33677/1160484)*x^8 + (2/173*a - 6/
↪173)*x^7 + (133/6747*a - 5373/96707)*x^6 + (-6/2249*a + 18584/290121)*x^5 + (-277/
↪8996*a + 38847/386828)*x^4 + (-135/4498*a + 42673/193414)*x^3 + (-1005/8996*a +␣
↪470245/1160484)*x^2 + (-1215/8996*a + 141165/386828)*x + 621/8996*a + 836445/
↪386828
sage: k.mod(f)
1
sage: k.mod(g)
a
sage: k.mod(h)
3
```

If the moduli are not coprime, a solution may not exist:

```
sage: crt(4,8,8,12)
20
sage: crt(4,6,8,12)
Traceback (most recent call last):
...
ValueError: no solution to crt problem since gcd(8,12) does not divide 4-6

sage: x = polygen(QQ)
sage: crt(2,3,x-1,x+1)
-1/2*x + 5/2
sage: crt(2,x,x^2-1,x^2+1)
-1/2*x^3 + x^2 + 1/2*x + 1
sage: crt(2,x,x^2-1,x^3-1)
Traceback (most recent call last):
...
ValueError: no solution to crt problem since gcd(x^2 - 1,x^3 - 1) does not divide 2-
↪x

sage: crt(int(2), int(3), int(7), int(11))
58
```

crt also work with numpy and gmpy2 numbers:

```
sage: import numpy
sage: crt(numpy.int8(2), numpy.int8(3), numpy.int8(7), numpy.int8(11))
58
sage: from gmpy2 import mpz
sage: crt(mpz(2), mpz(3), mpz(7), mpz(11))
58
sage: crt(mpz(2), 3, mpz(7), numpy.int8(11))
58
```

sage.arith.misc.**dedekind_psi**(*N*)

Return the value of the Dedekind psi function at N.

INPUT:

- N – a positive integer

OUTPUT:

an integer

The Dedekind psi function is the multiplicative function defined by

$$\psi(n) = n \prod_{p|n, pprime} (1 + 1/p).$$

See Wikipedia article Dedekind_psi_function and OEIS sequence A001615.

EXAMPLES:

```
sage: from sage.arith.misc import dedekind_psi
sage: [dedekind_psi(d) for d in range(1, 12)]
[1, 3, 4, 6, 6, 12, 8, 12, 12, 18, 12]
```

sage.arith.misc.**dedekind_sum**(*p*, *q*, *algorithm='default'*)

Return the Dedekind sum $s(p, q)$ defined for integers p, q as

$$s(p,q) = \sum_{i=0}^{q-1} \left(\left(\frac{i}{q}\right)\right) \left(\left(\frac{pi}{q}\right)\right)$$

where

$$((x)) = \begin{cases} x - \lfloor x \rfloor - \frac{1}{2} & \text{if } x \in \mathbf{Q} \setminus \mathbf{Z} \\ 0 & \text{if } x \in \mathbf{Z}. \end{cases}$$

Warning: Caution is required as the Dedekind sum sometimes depends on the algorithm or is left undefined when p and q are not coprime.

INPUT:

- `p`, `q` – integers
- `algorithm` – must be one of the following
  - `'default'` - (default) use FLINT
  - `'flint'` - use FLINT
  - `'pari'` - use PARI (gives different results if p and q are not coprime)

OUTPUT: a rational number

EXAMPLES:

Several small values:

```
sage: for q in range(10): print([dedekind_sum(p,q) for p in range(q+1)])
[0]
[0, 0]
[0, 0, 0]
[0, 1/18, -1/18, 0]
[0, 1/8, 0, -1/8, 0]
[0, 1/5, 0, 0, -1/5, 0]
[0, 5/18, 1/18, 0, -1/18, -5/18, 0]
[0, 5/14, 1/14, -1/14, 1/14, -1/14, -5/14, 0]
[0, 7/16, 1/8, 1/16, 0, -1/16, -1/8, -7/16, 0]
[0, 14/27, 4/27, 1/18, -4/27, 4/27, -1/18, -4/27, -14/27, 0]
```

Check relations for restricted arguments:

```
sage: q = 23; dedekind_sum(1, q); (q-1)*(q-2)/(12*q)
77/46
77/46
sage: p, q = 100, 723    # must be coprime
sage: dedekind_sum(p, q) + dedekind_sum(q, p)
31583/86760
sage: -1/4 + (p/q + q/p + 1/(p*q))/12
31583/86760
```

We check that evaluation works with large input:

```
sage: dedekind_sum(3^54 - 1, 2^93 + 1)
459340694971839990630374299870/297105609428491265975789813 79
sage: dedekind_sum(3^54 - 1, 2^93 + 1, algorithm='pari')
459340694971839990630374299870/297105609428491265975789813 79
```

We check consistency of the results:

```
sage: dedekind_sum(5, 7, algorithm='default')
-1/14
sage: dedekind_sum(5, 7, algorithm='flint')
-1/14
sage: dedekind_sum(5, 7, algorithm='pari')
-1/14
sage: dedekind_sum(6, 8, algorithm='default')
-1/8
sage: dedekind_sum(6, 8, algorithm='flint')
-1/8
sage: dedekind_sum(6, 8, algorithm='pari')
-1/8
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: dedekind_sum(int8(5), int8(7), algorithm='default')
-1/14
sage: from gmpy2 import mpz
sage: dedekind_sum(mpz(5), mpz(7), algorithm='default')
-1/14
```

REFERENCES:

- [Ap1997]
- Wikipedia article Dedekind_sum

sage.arith.misc.**differences**(*lis*, *n=1*)

Return the n successive differences of the elements in `lis`.

EXAMPLES:

```
sage: differences(prime_range(50))
[1, 2, 2, 4, 2, 4, 2, 4, 6, 2, 6, 4, 2, 4]
sage: differences([i^2 for i in range(1,11)])
```

(continues on next page)

(continued from previous page)

```
[3, 5, 7, 9, 11, 13, 15, 17, 19]
sage: differences([i^3 + 3*i for i in range(1,21)])
[10, 22, 40, 64, 94, 130, 172, 220, 274, 334, 400, 472, 550, 634, 724, 820, 922,␣
↪1030, 1144]
sage: differences([i^3 - i^2 for i in range(1,21)], 2)
[10, 16, 22, 28, 34, 40, 46, 52, 58, 64, 70, 76, 82, 88, 94, 100, 106, 112]
sage: differences([p - i^2 for i, p in enumerate(prime_range(50))], 3)
[-1, 2, -4, 4, -4, 4, 0, -6, 8, -6, 0, 4]
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: differences([int8(1),int8(4),int8(6),int8(19)])
[3, 2, 13]
sage: from gmpy2 import mpz
sage: differences([mpz(1),mpz(4),mpz(6),mpz(19)])
[mpz(3), mpz(2), mpz(13)]
```

AUTHORS:

- Timothy Clemans (2008-03-09)

sage.arith.misc.**divisors**(*n*)

Return the list of all divisors (up to units) of this element of a unique factorization domain.

For an integer, the list of all positive integer divisors of this integer, sorted in increasing order, is returned.

INPUT:

- `n` - the element

EXAMPLES:

Divisors of integers:

```
sage: divisors(-3)
[1, 3]
sage: divisors(6)
[1, 2, 3, 6]
sage: divisors(28)
[1, 2, 4, 7, 14, 28]
sage: divisors(2^5)
[1, 2, 4, 8, 16, 32]
sage: divisors(100)
[1, 2, 4, 5, 10, 20, 25, 50, 100]
sage: divisors(1)
[1]
sage: divisors(0)
Traceback (most recent call last):
...
ValueError: n must be nonzero
sage: divisors(2^3 * 3^2 * 17)
[1, 2, 3, 4, 6, 8, 9, 12, 17, 18, 24, 34, 36, 51, 68, 72,
102, 136, 153, 204, 306, 408, 612, 1224]
```

This function works whenever one has unique factorization:

```
sage: K.<a> = QuadraticField(7)
sage: divisors(K.ideal(7))
[Fractional ideal (1), Fractional ideal (-a), Fractional ideal (7)]
sage: divisors(K.ideal(3))
[Fractional ideal (1), Fractional ideal (3),
Fractional ideal (-a + 2), Fractional ideal (-a - 2)]
sage: divisors(K.ideal(35))
[Fractional ideal (1), Fractional ideal (5), Fractional ideal (-a),
Fractional ideal (7), Fractional ideal (-5*a), Fractional ideal (35)]
```

sage.arith.misc.**eratosthenes**(*n*)

Return a list of the primes $\leq n$.

This is extremely slow and is for educational purposes only.

INPUT:

- n - a positive integer

OUTPUT:

- a list of primes less than or equal to n.

EXAMPLES:

```
sage: eratosthenes(3)
[2, 3]
sage: eratosthenes(50)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
sage: len(eratosthenes(100))
25
sage: eratosthenes(213) == prime_range(213)
True
```

sage.arith.misc.**factor**(*n*, *proof=None*, *int_=False*, *algorithm='pari'*, *verbose=0*, ***kwds*)

Return the factorization of n. The result depends on the type of n.

If n is an integer, returns the factorization as an object of type Factorization.

If n is not an integer, `n.factor(proof=proof, **kwds)` gets called. See `n.factor??` for more documentation in this case.

Warning: This means that applying factor to an integer result of a symbolic computation will not factor the integer, because it is considered as an element of a larger symbolic ring.

EXAMPLES:

```
sage: f(n)=n^2
sage: is_prime(f(3))
False
sage: factor(f(3))
9
```

INPUT:

- n - an nonzero integer
- proof - bool or None (default: None)

- `int_` - bool (default: False) whether to return answers as Python ints
- `algorithm` - string
  - `'pari'` - (default) use the PARI c library
  - `'kash'` - use KASH computer algebra system (requires that kash be installed)
  - `'magma'` - use Magma (requires magma be installed)
- `verbose` - integer (default: 0); PARI's debug variable is set to this; e.g., set to 4 or 8 to see lots of output during factorization.

OUTPUT:

- factorization of n

The qsieve and ecm commands give access to highly optimized implementations of algorithms for doing certain integer factorization problems. These implementations are not used by the generic factor command, which currently just calls PARI (note that PARI also implements sieve and ecm algorithms, but they are not as optimized). Thus you might consider using them instead for certain numbers.

The factorization returned is an element of the class `Factorization`; see Factorization?? for more details, and examples below for usage. A Factorization contains both the unit factor (+1 or -1) and a sorted list of (prime, exponent) pairs.

The factorization displays in pretty-print format but it is easy to obtain access to the (prime,exponent) pairs and the unit, to recover the number from its factorization, and even to multiply two factorizations. See examples below.

EXAMPLES:

```
sage: factor(500)
2^2 * 5^3
sage: factor(-20)
-1 * 2^2 * 5
sage: f=factor(-20)
sage: list(f)
[(2, 2), (5, 1)]
sage: f.unit()
-1
sage: f.value()
-20
sage: factor( -next_prime(10^2) * next_prime(10^7) )
-1 * 101 * 10000019
```

```
sage: factor(-500, algorithm='kash')      # optional - kash
-1 * 2^2 * 5^3
```

```
sage: factor(-500, algorithm='magma')     # optional - magma
-1 * 2^2 * 5^3
```

```
sage: factor(0)
Traceback (most recent call last):
...
ArithmeticError: factorization of 0 is not defined
sage: factor(1)
1
```

(continues on next page)

(continued from previous page)

```
sage: factor(-1)
-1
sage: factor(2^(2^7)+1)
59649589127497217 * 5704689200685129054721
```

Sage calls PARI's factor, which has proof False by default. Sage has a global proof flag, set to True by default (see `sage.structure.proof.proof`, or proof.[tab]). To override the default, call this function with proof=False.

```
sage: factor(3^89-1, proof=False)
2 * 179 * 1611479891519807 * 504293943956596049162197
```

```
sage: factor(2^197 + 1)  # long time (2s)
3 * 197002597249 * 1348959352853811313 * 251951573867253012259144010843
```

Any object which has a factor method can be factored like this:

```
sage: K.<i> = QuadraticField(-1)
sage: factor(122 - 454*i)
(-3*i - 2) * (-i - 2)^3 * (i + 1)^3 * (i + 4)
```

To access the data in a factorization:

```
sage: f = factor(420); f
2^2 * 3 * 5 * 7
sage: [x for x in f]
[(2, 2), (3, 1), (5, 1), (7, 1)]
sage: [p for p,e in f]
[2, 3, 5, 7]
sage: [e for p,e in f]
[2, 1, 1, 1]
sage: [p^e for p,e in f]
[4, 3, 5, 7]
```

We can factor Python, numpy and gmpy2 numbers:

```
sage: factor(math.pi)
3.141592653589793
sage: import numpy
sage: factor(numpy.int8(30))
2 * 3 * 5
sage: import gmpy2
sage: factor(gmpy2.mpz(30))
2 * 3 * 5
```

`sage.arith.misc.`**`factorial`**(*n*, *algorithm='gmp'*)

Compute the factorial of n, which is the product $1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$.

INPUT:

- `n` - an integer
- `algorithm` - string (default: 'gmp'):
  - `'gmp'` - use the GMP C-library factorial function
  - `'pari'` - use PARI's factorial function

OUTPUT: an integer

EXAMPLES:

```
sage: from sage.arith.misc import factorial
sage: factorial(0)
1
sage: factorial(4)
24
sage: factorial(10)
3628800
sage: factorial(1) == factorial(0)
True
sage: factorial(6) == 6*5*4*3*2
True
sage: factorial(1) == factorial(0)
True
sage: factorial(71) == 71* factorial(70)
True
sage: factorial(-32)
Traceback (most recent call last):
...
ValueError: factorial -- must be nonnegative
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: factorial(int8(4))
24
sage: from gmpy2 import mpz
sage: factorial(mpz(4))
24
```

PERFORMANCE: This discussion is valid as of April 2006. All timings below are on a Pentium Core Duo 2Ghz MacBook Pro running Linux with a 2.6.16.1 kernel.

- It takes less than a minute to compute the factorial of 10^7 using the GMP algorithm, and the factorial of 10^6 takes less than 4 seconds.
- The GMP algorithm is faster and more memory efficient than the PARI algorithm. E.g., PARI computes 10^7 factorial in 100 seconds on the core duo 2Ghz.
- For comparison, computation in Magma $\leq$ 2.12-10 of $n!$ is best done using `*[1..n]`. It takes 113 seconds to compute the factorial of 10^7 and 6 seconds to compute the factorial of 10^6. Mathematica V5.2 compute the factorial of 10^7 in 136 seconds and the factorial of 10^6 in 7 seconds. (Mathematica is notably very efficient at memory usage when doing factorial calculations.)

`sage.arith.misc.`**`falling_factorial`**`(`x, a`)`

Return the falling factorial $(x)_a$.

The notation in the literature is a mess: often $(x)_a$, but there are many other notations: GKP: Concrete Mathematics uses $x^{\underline{a}}$.

Definition: for integer $a \geq 0$ we have $x(x-1)\cdots(x-a+1)$. In all other cases we use the GAMMA-function: $\frac{\Gamma(x+1)}{\Gamma(x-a+1)}$.

INPUT:

- `x` – element of a ring

- a – a non-negative integer or
- x and a – any numbers

OUTPUT: the falling factorial

See also:

rising_factorial()

EXAMPLES:

```
sage: falling_factorial(10, 3)
720
sage: falling_factorial(10, RR('3.0'))
720.000000000000
sage: falling_factorial(10, RR('3.3'))
1310.11633396601
sage: falling_factorial(10, 10)
3628800
sage: factorial(10)
3628800
sage: a = falling_factorial(1+I, I); a
gamma(I + 2)
sage: CC(a)
0.652965496420167 + 0.343065839816545*I
sage: falling_factorial(1+I, 4)
4*I + 2
sage: falling_factorial(I, 4)
-10

sage: M = MatrixSpace(ZZ, 4, 4)
sage: A = M([1,0,1,0,1,0,1,0,1,0,10,10,1,0,1,1])
sage: falling_factorial(A, 2) # A(A - I)
[  1   0  10  10]
[  1   0  10  10]
[ 20   0 101 100]
[  2   0  11  10]

sage: x = ZZ['x'].0
sage: falling_factorial(x, 4)
x^4 - 6*x^3 + 11*x^2 - 6*x
```

AUTHORS:

- Jaap Spies (2006-03-05)

sage.arith.misc.**four_squares**(*n*)

Write the integer n as a sum of four integer squares.

INPUT:

- n – an integer

OUTPUT: a tuple (a, b, c, d) of non-negative integers such that $n = a^2 + b^2 + c^2 + d^2$ with $a <= b <= c <= d$.

EXAMPLES:

```
sage: four_squares(3)
(0, 1, 1, 1)
sage: four_squares(13)
(0, 0, 2, 3)
sage: four_squares(130)
(0, 0, 3, 11)
sage: four_squares(11010110011004)
(90, 102, 1220, 1049290)
sage: four_squares(10^100-1)
(155024616290, 2612183768627, 14142135623730950488016887,␣
↪99999999999999999999999999999999999999999999999999)
sage: for i in range(2^129, 2^129+10000):  # long time
....:     S = four_squares(i)
....:     assert sum(x^2 for x in S) == i
```

sage.arith.misc.**fundamental_discriminant**(*D*)

Return the discriminant of the quadratic extension $K = Q(\sqrt{D})$, i.e. an integer d congruent to either 0 or 1, mod 4, and such that, at most, the only square dividing it is 4.

INPUT:

- D - an integer

OUTPUT:

- an integer, the fundamental discriminant

EXAMPLES:

```
sage: fundamental_discriminant(102)
408
sage: fundamental_discriminant(720)
5
sage: fundamental_discriminant(2)
8
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: fundamental_discriminant(int8(102))
408
sage: from gmpy2 import mpz
sage: fundamental_discriminant(mpz(102))
408
```

sage.arith.misc.**gauss_sum**(*char_value*, *finite_field*)

Return the Gauss sums for a general finite field.

INPUT:

- char_value – choice of multiplicative character, given by its value on the finite_field.multiplicative_generator()
- finite_field – a finite field

OUTPUT:

an element of the parent ring of char_value, that can be any field containing enough roots of unity, for example the UniversalCyclotomicField, QQbar or ComplexField

For a finite field F of characteristic p, the Gauss sum associated to a multiplicative character χ (with values in a ring K) is defined as

$$\sum_{x \in F^\times} \chi(x)\zeta_p^{\mathrm{Tr}\, x},$$

where $\zeta_p \in K$ is a primitive root of unity of order p and Tr is the trace map from F to its prime field $\mathbf{F}_p$.

For more info on Gauss sums, see Wikipedia article Gauss_sum.

Todo: Implement general Gauss sums for an arbitrary pair (`multiplicative_character`, `additive_character`)

EXAMPLES:

```
sage: from sage.arith.misc import gauss_sum
sage: F = GF(5); q = 5
sage: zq = UniversalCyclotomicField().zeta(q-1)
sage: L = [gauss_sum(zq**i,F) for i in range(5)]; L
[-1,
 E(20)^4 + E(20)^13 - E(20)^16 - E(20)^17,
 E(5) - E(5)^2 - E(5)^3 + E(5)^4,
 E(20)^4 - E(20)^13 - E(20)^16 + E(20)^17,
 -1]
sage: [g*g.conjugate() for g in L]
[1, 5, 5, 5, 1]

sage: F = GF(11**2); q = 11**2
sage: zq = UniversalCyclotomicField().zeta(q-1)
sage: g = gauss_sum(zq**4,F)
sage: g*g.conjugate()
121
```

See also:

- `sage.rings.padics.misc.gauss_sum()` for a p-adic version
- `sage.modular.dirichlet.DirichletCharacter.gauss_sum()` for prime finite fields
- `sage.modular.dirichlet.DirichletCharacter.gauss_sum_numerical()` for prime finite fields

`sage.arith.misc.`**`gcd`**(*a*, *b=None*, ***kwargs*)

Return the greatest common divisor of `a` and `b`.

If `a` is a list and `b` is omitted, return instead the greatest common divisor of all elements of `a`.

INPUT:

- `a,b` – two elements of a ring with gcd or
- `a` – a list or tuple of elements of a ring with gcd

Additional keyword arguments are passed to the respectively called methods.

OUTPUT:

The given elements are first coerced into a common parent. Then, their greatest common divisor *in that common parent* is returned.

EXAMPLES:

```
sage: GCD(97,100)
1
sage: GCD(97*10^15, 19^20*97^2)
97
sage: GCD(2/3, 4/5)
2/15
sage: GCD([2,4,6,8])
2
sage: GCD(srange(0,10000,10))  # fast  !!
10
```

Note that to take the gcd of n elements for $n \neq 2$ you must put the elements into a list by enclosing them in `[..]`. Before trac ticket #4988 the following wrongly returned 3 since the third parameter was just ignored:

```
sage: gcd(3, 6, 2)
Traceback (most recent call last):
...
TypeError: ...gcd() takes ...
sage: gcd([3, 6, 2])
1
```

Similarly, giving just one element (which is not a list) gives an error:

```
sage: gcd(3)
Traceback (most recent call last):
...
TypeError: 'sage.rings.integer.Integer' object is not iterable
```

By convention, the gcd of the empty list is (the integer) 0:

```
sage: gcd([])
0
sage: type(gcd([]))
<class 'sage.rings.integer.Integer'>
```

sage.arith.misc.**get_gcd**(*order*)

Return the fastest gcd function for integers of size no larger than order.

EXAMPLES:

```
sage: sage.arith.misc.get_gcd(4000)
<built-in method gcd_int of sage.rings.fast_arith.arith_int object at ...>
sage: sage.arith.misc.get_gcd(400000)
<built-in method gcd_longlong of sage.rings.fast_arith.arith_llong object at ...>
sage: sage.arith.misc.get_gcd(4000000000)
<function gcd at ...>
```

sage.arith.misc.**get_inverse_mod**(*order*)

Return the fastest inverse_mod function for integers of size no larger than order.

EXAMPLES:

```
sage: sage.arith.misc.get_inverse_mod(6000)
<built-in method inverse_mod_int of sage.rings.fast_arith.arith_int object at ...>
sage: sage.arith.misc.get_inverse_mod(600000)
```

(continues on next page)

(continued from previous page)

```
<built-in method inverse_mod_longlong of sage.rings.fast_arith.arith_llong object␣
↪at ...>
sage: sage.arith.misc.get_inverse_mod(6000000000)
<function inverse_mod at ...>
```

sage.arith.misc.**hilbert_conductor**(a, b)

Return the product of all (finite) primes where the Hilbert symbol is -1.

This is the (reduced) discriminant of the quaternion algebra (a, b) over $\mathbf{Q}$.

INPUT:

- a, b – integers

OUTPUT:

squarefree positive integer

EXAMPLES:

```
sage: hilbert_conductor(-1, -1)
2
sage: hilbert_conductor(-1, -11)
11
sage: hilbert_conductor(-2, -5)
5
sage: hilbert_conductor(-3, -17)
17
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: hilbert_conductor(int8(-3), int8(-17))
17
sage: from gmpy2 import mpz
sage: hilbert_conductor(mpz(-3), mpz(-17))
17
```

AUTHOR:

- Gonzalo Tornaria (2009-03-02)

sage.arith.misc.**hilbert_conductor_inverse**(d)

Finds a pair of integers (a, b) such that `hilbert_conductor(a,b) == d`.

The quaternion algebra (a, b) over $\mathbf{Q}$ will then have (reduced) discriminant d.

INPUT:

- d – square-free positive integer

OUTPUT: pair of integers

EXAMPLES:

```
sage: hilbert_conductor_inverse(2)
(-1, -1)
sage: hilbert_conductor_inverse(3)
(-1, -3)
```

(continues on next page)

(continued from previous page)

```
sage: hilbert_conductor_inverse(6)
(-1, 3)
sage: hilbert_conductor_inverse(30)
(-3, -10)
sage: hilbert_conductor_inverse(4)
Traceback (most recent call last):
...
ValueError: d needs to be squarefree
sage: hilbert_conductor_inverse(-1)
Traceback (most recent call last):
...
ValueError: d needs to be positive
```

AUTHOR:

- Gonzalo Tornaria (2009-03-02)

sage.arith.misc.**hilbert_symbol**(*a*, *b*, *p*, *algorithm='pari'*)

Return 1 if $ax^2 + by^2$ p-adically represents a nonzero square, otherwise returns -1. If either a or b is 0, returns 0.

INPUT:

- `a, b` - integers
- `p` - integer; either prime or -1 (which represents the archimedean place)
- `algorithm` - string
  - `'pari'` - (default) use the PARI C library
  - `'direct'` - use a Python implementation
  - `'all'` - use both PARI and direct and check that the results agree, then return the common answer

OUTPUT: integer (0, -1, or 1)

EXAMPLES:

```
sage: hilbert_symbol (-1, -1, -1, algorithm='all')
-1
sage: hilbert_symbol (2,3, 5, algorithm='all')
1
sage: hilbert_symbol (4, 3, 5, algorithm='all')
1
sage: hilbert_symbol (0, 3, 5, algorithm='all')
0
sage: hilbert_symbol (-1, -1, 2, algorithm='all')
-1
sage: hilbert_symbol (1, -1, 2, algorithm='all')
1
sage: hilbert_symbol (3, -1, 2, algorithm='all')
-1

sage: hilbert_symbol(QQ(-1)/QQ(4), -1, 2) == -1
True
sage: hilbert_symbol(QQ(-1)/QQ(4), -1, 3) == 1
True
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: hilbert_symbol(int8(2),int8(3),int8(5),algorithm='all')
1
sage: from gmpy2 import mpz
sage: hilbert_symbol(mpz(2),mpz(3),mpz(5),algorithm='all')
1
```

AUTHORS:

- William Stein and David Kohel (2006-01-05)

sage.arith.misc.**integer_ceil**(x)

Return the ceiling of x.

EXAMPLES:

```
sage: integer_ceil(5.4)
6
sage: integer_ceil(x)
Traceback (most recent call last):
...
NotImplementedError: computation of ceil of x not implemented
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import float32
sage: integer_ceil(float32(5.4))
6
sage: from gmpy2 import mpfr
sage: integer_ceil(mpfr(5.4))
6
```

sage.arith.misc.**integer_floor**(x)

Return the largest integer $\leq x$.

INPUT:

- x - an object that has a floor method or is coercible to int

OUTPUT: an Integer

EXAMPLES:

```
sage: integer_floor(5.4)
5
sage: integer_floor(float(5.4))
5
sage: integer_floor(-5/2)
-3
sage: integer_floor(RDF(-5/2))
-3

sage: integer_floor(x)
Traceback (most recent call last):
...
NotImplementedError: computation of floor of x not implemented
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import float32
sage: integer_floor(float32(5.4))
5
sage: from gmpy2 import mpfr
sage: integer_floor(mpfr(5.4))
5
```

sage.arith.misc.**integer_trunc**(*i*)

Truncate to the integer closer to zero

EXAMPLES:

```
sage: from sage.arith.misc import integer_trunc as trunc
sage: trunc(-3/2), trunc(-1), trunc(-1/2), trunc(0), trunc(1/2), trunc(1), trunc(3/
↪2)
(-1, -1, 0, 0, 0, 1, 1)
sage: isinstance(trunc(3/2), Integer)
True
```

sage.arith.misc.**inverse_mod**(*a*, *m*)

The inverse of the ring element a modulo m.

If no special inverse_mod is defined for the elements, it tries to coerce them into integers and perform the inversion there

```
sage: inverse_mod(7,1)
0
sage: inverse_mod(5,14)
3
sage: inverse_mod(3,-5)
2
```

Tests with numpy and mpz numbers:

```
sage: from numpy import int8
sage: inverse_mod(int8(5),int8(14))
3
sage: from gmpy2 import mpz
sage: inverse_mod(mpz(5),mpz(14))
3
```

sage.arith.misc.**is_power_of_two**(*n*)

Return whether n is a power of 2.

INPUT:

- n – integer

OUTPUT:

boolean

EXAMPLES:

```
sage: is_power_of_two(1024)
True
```

(continues on next page)

(continued from previous page)

```
sage: is_power_of_two(1)
True
sage: is_power_of_two(24)
False
sage: is_power_of_two(0)
False
sage: is_power_of_two(-4)
False
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: is_power_of_two(int8(16))
True
sage: is_power_of_two(int8(24))
False
sage: from gmpy2 import mpz
sage: is_power_of_two(mpz(16))
True
sage: is_power_of_two(mpz(24))
False
```

sage.arith.misc.**is_prime**(n)

Return `True` if n is a prime number, and `False` otherwise.

Use a provable primality test or a strong pseudo-primality test depending on the global `arithmetic proof flag`.

INPUT:

- `n` - the object for which to determine primality

See also:

- *is_pseudoprime()*
- *sage.rings.integer.Integer.is_prime()*

AUTHORS:

- Kevin Stueve kstueve@uw.edu (2010-01-17): delegated calculation to `n.is_prime()`

EXAMPLES:

```
sage: is_prime(389)
True
sage: is_prime(2000)
False
sage: is_prime(2)
True
sage: is_prime(-1)
False
sage: is_prime(1)
False
sage: is_prime(-2)
False
```

(continues on next page)

(continued from previous page)

```
sage: a = 2**2048 + 981
sage: is_prime(a)    # not tested - takes ~ 1min
sage: proof.arithmetic(False)
sage: is_prime(a)    # instantaneous!
True
sage: proof.arithmetic(True)
```

sage.arith.misc.**is_prime_power**(*n*, *get_data=False*)

Test whether n is a positive power of a prime number

This function simply calls the method *Integer.is_prime_power()* of Integers.

INPUT:

- n – an integer
- get_data – if set to True, return a pair (p,k) such that this integer equals p^k instead of True or (self, 0) instead of False

EXAMPLES:

```
sage: is_prime_power(389)
True
sage: is_prime_power(2000)
False
sage: is_prime_power(2)
True
sage: is_prime_power(1024)
True
sage: is_prime_power(1024, get_data=True)
(2, 10)
```

The same results can be obtained with:

```
sage: 389.is_prime_power()
True
sage: 2000.is_prime_power()
False
sage: 2.is_prime_power()
True
sage: 1024.is_prime_power()
True
sage: 1024.is_prime_power(get_data=True)
(2, 10)
```

sage.arith.misc.**is_pseudoprime**(*n*)

Test whether n is a pseudo-prime

The result is *NOT* proven correct - *this is a pseudo-primality test!*.

INPUT:

- n – an integer

Note: We do not consider negatives of prime numbers as prime.

EXAMPLES:

```
sage: is_pseudoprime(389)
True
sage: is_pseudoprime(2000)
False
sage: is_pseudoprime(2)
True
sage: is_pseudoprime(-1)
False
sage: factor(-6)
-1 * 2 * 3
sage: is_pseudoprime(1)
False
sage: is_pseudoprime(-2)
False
```

sage.arith.misc.**is_pseudoprime_power**(*n*, *get_data=False*)

Test if `n` is a power of a pseudoprime.

The result is *NOT* proven correct - *this IS a pseudo-primality test!*. Note that a prime power is a positive power of a prime number so that 1 is not a prime power.

INPUT:

- `n` - an integer
- `get_data` - (boolean) instead of a boolean return a pair (p, k) so that `n` equals p^k and p is a pseudoprime or $(n, 0)$ otherwise.

EXAMPLES:

```
sage: is_pseudoprime_power(389)
True
sage: is_pseudoprime_power(2000)
False
sage: is_pseudoprime_power(2)
True
sage: is_pseudoprime_power(1024)
True
sage: is_pseudoprime_power(-1)
False
sage: is_pseudoprime_power(1)
False
sage: is_pseudoprime_power(997^100)
True
```

Use of the get_data keyword:

```
sage: is_pseudoprime_power(3^1024, get_data=True)
(3, 1024)
sage: is_pseudoprime_power(2^256, get_data=True)
(2, 256)
sage: is_pseudoprime_power(31, get_data=True)
(31, 1)
sage: is_pseudoprime_power(15, get_data=True)
(15, 0)
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int16
sage: is_pseudoprime_power(int16(1024))
True
sage: from gmpy2 import mpz
sage: is_pseudoprime_power(mpz(1024))
True
```

sage.arith.misc.**is_square**(*n*, *root=False*)

Return whether or not `n` is square.

If `n` is a square also return the square root. If `n` is not square, also return `None`.

INPUT:

- `n` – an integer
- `root` – whether or not to also return a square root (default: `False`)

OUTPUT:

- `bool` – whether or not a square
- `object` – (optional) an actual square if found, and `None` otherwise.

EXAMPLES:

```
sage: is_square(2)
False
sage: is_square(4)
True
sage: is_square(2.2)
True
sage: is_square(-2.2)
False
sage: is_square(CDF(-2.2))
True
sage: is_square((x-1)^2)
Traceback (most recent call last):
...
NotImplementedError: is_square() not implemented for
non-constant or relational elements of Symbolic Ring
```

```
sage: is_square(4, True)
(True, 2)
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: is_square(int8(4))
True
sage: from gmpy2 import mpz
sage: is_square(mpz(4))
True
```

Tests with Polynomial:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: R.<a,b,c,d> = QQ[]
sage: p = a*b + c*d*a*d*a + 5
sage: is_square(p**2)
True
```

sage.arith.misc.**is_squarefree**(*n*)

Test whether n is square free.

EXAMPLES:

```
sage: is_squarefree(100)
False
sage: is_squarefree(101)
True

sage: R = ZZ['x']
sage: x = R.gen()
sage: is_squarefree((x^2+x+1) * (x-2))
True
sage: is_squarefree((x-1)**2 * (x-3))
False

sage: O = ZZ[sqrt(-1)]
sage: I = O.gen(1)
sage: is_squarefree(I+1)
True
sage: is_squarefree(O(2))
False
sage: O(2).factor()
(-I) * (I + 1)^2
```

This method fails on domains which are not Unique Factorization Domains:

```
sage: O = ZZ[sqrt(-5)]
sage: a = O.gen(1)
sage: is_squarefree(a - 3)
Traceback (most recent call last):
...
ArithmeticError: non-principal ideal in factorization
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: is_squarefree(int8(100))
False
sage: is_squarefree(int8(101))
True
sage: from gmpy2 import mpz
sage: is_squarefree(mpz(100))
False
sage: is_squarefree(mpz(101))
True
```

sage.arith.misc.**jacobi_symbol**(a, b)
The Jacobi symbol of integers a and b, where b is odd.

Note: The *kronecker_symbol()* command extends the Jacobi symbol to all integers b.

If

$b = p_1^{e_1} * ... * p_r^{e_r}$

then

$(a|b) = (a|p_1)^{e_1}...(a|p_r)^{e_r}$

where $(a|p_j)$ are Legendre Symbols.

INPUT:

- a - an integer
- b - an odd integer

EXAMPLES:

```
sage: jacobi_symbol(10,777)
-1
sage: jacobi_symbol(10,5)
0
sage: jacobi_symbol(10,2)
Traceback (most recent call last):
...
ValueError: second input must be odd, 2 is not odd
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int16
sage: jacobi_symbol(int16(10),int16(777))
-1
sage: from gmpy2 import mpz
sage: jacobi_symbol(mpz(10),mpz(777))
-1
```

sage.arith.misc.**kronecker**(x, y)
The Kronecker symbol $(x|y)$.

INPUT:

- x – integer
- y – integer

OUTPUT:

- an integer

EXAMPLES:

```
sage: kronecker_symbol(13,21)
-1
sage: kronecker_symbol(101,4)
1
```

This is also available as *kronecker()*:

```
sage: kronecker(3,5)
-1
sage: kronecker(3,15)
0
sage: kronecker(2,15)
1
sage: kronecker(-2,15)
-1
sage: kronecker(2/3,5)
1
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: kronecker_symbol(int8(13),int8(21))
-1
sage: from gmpy2 import mpz
sage: kronecker_symbol(mpz(13),mpz(21))
-1
```

sage.arith.misc.**kronecker_symbol**(x, y)

The Kronecker symbol $(x|y)$.

INPUT:

- x – integer
- y – integer

OUTPUT:

- an integer

EXAMPLES:

```
sage: kronecker_symbol(13,21)
-1
sage: kronecker_symbol(101,4)
1
```

This is also available as *kronecker()*:

```
sage: kronecker(3,5)
-1
sage: kronecker(3,15)
0
sage: kronecker(2,15)
1
sage: kronecker(-2,15)
-1
sage: kronecker(2/3,5)
1
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: kronecker_symbol(int8(13),int8(21))
-1
sage: from gmpy2 import mpz
sage: kronecker_symbol(mpz(13),mpz(21))
-1
```

sage.arith.misc.**legendre_symbol**(x, p)
The Legendre symbol $(x|p)$, for p prime.

Note: The *kronecker_symbol()* command extends the Legendre symbol to composite moduli and $p = 2$.

INPUT:

- x - integer
- p - an odd prime number

EXAMPLES:

```
sage: legendre_symbol(2,3)
-1
sage: legendre_symbol(1,3)
1
sage: legendre_symbol(1,2)
Traceback (most recent call last):
...
ValueError: p must be odd
sage: legendre_symbol(2,15)
Traceback (most recent call last):
...
ValueError: p must be a prime
sage: kronecker_symbol(2,15)
1
sage: legendre_symbol(2/3,7)
-1
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: legendre_symbol(int8(2),int8(3))
-1
sage: from gmpy2 import mpz
sage: legendre_symbol(mpz(2),mpz(3))
-1
```

sage.arith.misc.**mqrr_rational_reconstruction**(u, m, T)
Maximal Quotient Rational Reconstruction.

For research purposes only - this is pure Python, so slow.

INPUT:

- u, m, T - integers such that $m > u \geq 0, T > 0$.

OUTPUT:

Either integers n, d such that $d > 0$, $\gcd(n, d) = 1$, $n/d = u \bmod m$, and $T \cdot d \cdot |n| < m$, or None.

Reference: Monagan, Maximal Quotient Rational Reconstruction: An Almost Optimal Algorithm for Rational Reconstruction (page 11)

This algorithm is probabilistic.

EXAMPLES:

```
sage: mqrr_rational_reconstruction(21,3100,13)
(21, 1)
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int16
sage: mqrr_rational_reconstruction(int16(21),int16(3100),int16(13))
(21, 1)
sage: from gmpy2 import mpz
sage: mqrr_rational_reconstruction(mpz(21),mpz(3100),mpz(13))
(21, 1)
```

sage.arith.misc.**multinomial**(**ks*)

Return the multinomial coefficient.

INPUT:

- either an arbitrary number of integer arguments $k_1, \ldots, k_n$
- or an iterable (e.g. a list) of integers $[k_1, \ldots, k_n]$

OUTPUT:

Return the integer:

$$\binom{k_1 + \cdots + k_n}{k_1, \cdots, k_n} = \frac{(\sum_{i=1}^{n} k_i)!}{\prod_{i=1}^{n} k_i!} = \prod_{i=1}^{n} \binom{\sum_{j=1}^{i} k_j}{k_i}$$

EXAMPLES:

```
sage: multinomial(0, 0, 2, 1, 0, 0)
3
sage: multinomial([0, 0, 2, 1, 0, 0])
3
sage: multinomial(3, 2)
10
sage: multinomial(2^30, 2, 1)
618970023101454657175683075
sage: multinomial([2^30, 2, 1])
618970023101454657175683075
sage: multinomial(Composition([1, 3]))
4
sage: multinomial(Partition([4, 2]))
15
```

AUTHORS:

- Gabriel Ebner

sage.arith.misc.**multinomial_coefficients**(*m*, *n*)

Return a dictionary containing pairs $\{(k_1, k_2, ..., k_m) : C_{k,n}\}$ where $C_{k,n}$ are multinomial coefficients such that $n = k_1 + k_2 + ... + k_m$.

INPUT:

- `m` - integer
- `n` - integer

OUTPUT: dict

EXAMPLES:

```
sage: sorted(multinomial_coefficients(2, 5).items())
[((0, 5), 1), ((1, 4), 5), ((2, 3), 10), ((3, 2), 10), ((4, 1), 5), ((5, 0), 1)]
```

Notice that these are the coefficients of $(x + y)^5$:

```
sage: R.<x,y> = QQ[]
sage: (x+y)^5
x^5 + 5*x^4*y + 10*x^3*y^2 + 10*x^2*y^3 + 5*x*y^4 + y^5
```

```
sage: sorted(multinomial_coefficients(3, 2).items())
[((0, 0, 2), 1), ((0, 1, 1), 2), ((0, 2, 0), 1), ((1, 0, 1), 2), ((1, 1, 0), 2),␣
↪((2, 0, 0), 1)]
```

ALGORITHM: The algorithm we implement for computing the multinomial coefficients is based on the following result:

$$\binom{n}{k_1, \cdots, k_m} = \frac{k_1 + 1}{n - k_1} \sum_{i=2}^{m} \binom{n}{k_1 + 1, \cdots, k_i - 1, \cdots}$$

e.g.:

```
sage: k = (2, 4, 1, 0, 2, 6, 0, 0, 3, 5, 7, 1) # random value
sage: n = sum(k)
sage: s = 0
sage: for i in range(1, len(k)):
....:     ki = list(k)
....:     ki[0] += 1
....:     ki[i] -= 1
....:     s += multinomial(n, *ki)
sage: multinomial(n, *k) == (k[0] + 1) / (n - k[0]) * s
True
```

sage.arith.misc.**next_prime**(*n*, *proof=None*)

The next prime greater than the integer n. If n is prime, then this function does not return n, but the next prime after n. If the optional argument proof is False, this function only returns a pseudo-prime, as defined by the PARI nextprime function. If it is None, uses the global default (see `sage.structure.proof.proof`)

INPUT:

- `n` - integer
- `proof` - bool or None (default: None)

EXAMPLES:

```
sage: next_prime(-100)
2
sage: next_prime(1)
2
sage: next_prime(2)
3
sage: next_prime(3)
5
sage: next_prime(4)
5
```

Notice that the next_prime(5) is not 5 but 7.

```
sage: next_prime(5)
7
sage: next_prime(2004)
2011
```

sage.arith.misc.**next_prime_power**(*n*)

Return the smallest prime power greater than n.

Note that if n is a prime power, then this function does not return n, but the next prime power after n.

This function just calls the method *Integer.next_prime_power()* of Integers.

See also:

- *is_prime_power()* (and *Integer.is_prime_power()*)
- *previous_prime_power()* (and *Integer.previous_prime_power()*)

EXAMPLES:

```
sage: next_prime_power(1)
2
sage: next_prime_power(2)
3
sage: next_prime_power(10)
11
sage: next_prime_power(7)
8
sage: next_prime_power(99)
101
```

The same results can be obtained with:

```
sage: 1.next_prime_power()
2
sage: 2.next_prime_power()
3
sage: 10.next_prime_power()
11
```

Note that 2 is the smallest prime power:

```
sage: next_prime_power(-10)
2
sage: next_prime_power(0)
2
```

sage.arith.misc.**next_probable_prime**(*n*)
Return the next probable prime after self, as determined by PARI.

INPUT:

- n - an integer

EXAMPLES:

```
sage: next_probable_prime(-100)
2
sage: next_probable_prime(19)
23
sage: next_probable_prime(int(999999999))
1000000007
sage: next_probable_prime(2^768)
155251809230070893514897948846250255525688601711669661113905203802605095268637688633087840882864647
```

sage.arith.misc.**nth_prime**(*n*)
Return the n-th prime number (1-indexed, so that 2 is the 1st prime.)

INPUT:

- n – a positive integer

OUTPUT:

- the n-th prime number

EXAMPLES:

```
sage: nth_prime(3)
5
sage: nth_prime(10)
29
sage: nth_prime(10^7)
179424673
```

```
sage: nth_prime(0)
Traceback (most recent call last):
...
ValueError: nth prime meaningless for non-positive n (=0)
```

sage.arith.misc.**number_of_divisors**(*n*)
Return the number of divisors of the integer n.

INPUT:

- n - a nonzero integer

OUTPUT:

- an integer, the number of divisors of n

EXAMPLES:

```
sage: number_of_divisors(100)
9
sage: number_of_divisors(-720)
30
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: number_of_divisors(int8(100))
9
sage: from gmpy2 import mpz
sage: number_of_divisors(mpz(100))
9
```

sage.arith.misc.**odd_part**(n)

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

EXAMPLES:

```
sage: odd_part(5)
5
sage: odd_part(4)
1
sage: odd_part(factorial(31))
12252984425690655138679 6875
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: odd_part(int8(5))
5
sage: from gmpy2 import mpz
sage: odd_part(mpz(5))
5
```

sage.arith.misc.**power_mod**(a, n, m)

Return the n-th power of a modulo m, where a and m are elements of a ring that implements the modulo operator %.

ALGORITHM: square-and-multiply

EXAMPLES:

```
sage: power_mod(2,388,389)
1
sage: power_mod(2,390,391)
285
sage: power_mod(2,-1,7)
4
sage: power_mod(11,1,7)
4
```

This function works for fairly general rings:

```
sage: R.<x> = ZZ[]
sage: power_mod(3*x, 10, 7)
4*x^10
sage: power_mod(-3*x^2+4, 7, 2*x^3-5)
x^14 + x^8 + x^6 + x^3 + 962509*x^2 - 791910*x - 698281
```

sage.arith.misc.**previous_prime**(*n*)

The largest prime < n. The result is provably correct. If n <= 1, this function raises a ValueError.

EXAMPLES:

```
sage: previous_prime(10)
7
sage: previous_prime(7)
5
sage: previous_prime(8)
7
sage: previous_prime(7)
5
sage: previous_prime(5)
3
sage: previous_prime(3)
2
sage: previous_prime(2)
Traceback (most recent call last):
...
ValueError: no previous prime
sage: previous_prime(1)
Traceback (most recent call last):
...
ValueError: no previous prime
sage: previous_prime(-20)
Traceback (most recent call last):
...
ValueError: no previous prime
```

sage.arith.misc.**previous_prime_power**(*n*)

Return the largest prime power smaller than n.

The result is provably correct. If n is smaller or equal than 2 this function raises an error.

This function simply call the method *Integer.previous_prime_power()* of Integers.

See also:

- *is_prime_power()* (and *Integer.is_prime_power()*)
- *next_prime_power()* (and *Integer.next_prime_power()*)

EXAMPLES:

```
sage: previous_prime_power(3)
2
sage: previous_prime_power(10)
9
```

(continues on next page)

(continued from previous page)

```
sage: previous_prime_power(7)
5
sage: previous_prime_power(127)
125
```

The same results can be obtained with:

```
sage: 3.previous_prime_power()
2
sage: 10.previous_prime_power()
9
sage: 7.previous_prime_power()
5
sage: 127.previous_prime_power()
125
```

Input less than or equal to 2 raises errors:

```
sage: previous_prime_power(2)
Traceback (most recent call last):
...
ValueError: no prime power less than 2
sage: previous_prime_power(-10)
Traceback (most recent call last):
...
ValueError: no prime power less than 2
```

```
sage: n = previous_prime_power(2^16 - 1)
sage: while is_prime(n):
....:     n = previous_prime_power(n)
sage: factor(n)
251^2
```

sage.arith.misc.**prime_divisors**(*n*)

Return the list of prime divisors (up to units) of this element of a unique factorization domain.

INPUT:

- `n` – any object which can be decomposed into prime factors

OUTPUT:

A list of prime factors of `n`. For integers, this list is sorted in increasing order.

EXAMPLES:

Prime divisors of positive integers:

```
sage: prime_divisors(1)
[]
sage: prime_divisors(100)
[2, 5]
sage: prime_divisors(2004)
[2, 3, 167]
```

If `n` is negative, we do *not* include -1 among the prime divisors, since -1 is not a prime number:

```
sage: prime_divisors(-100)
[2, 5]
```

For polynomials we get all irreducible factors:

```
sage: R.<x> = PolynomialRing(QQ)
sage: prime_divisors(x^12 - 1)
[x - 1, x + 1, x^2 - x + 1, x^2 + 1, x^2 + x + 1, x^4 - x^2 + 1]
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: prime_divisors(int8(-100))
[2, 5]
sage: from gmpy2 import mpz
sage: prime_divisors(mpz(-100))
[2, 5]
```

sage.arith.misc.**prime_factors**(*n*)

Return the list of prime divisors (up to units) of this element of a unique factorization domain.

INPUT:

- n – any object which can be decomposed into prime factors

OUTPUT:

A list of prime factors of n. For integers, this list is sorted in increasing order.

EXAMPLES:

Prime divisors of positive integers:

```
sage: prime_divisors(1)
[]
sage: prime_divisors(100)
[2, 5]
sage: prime_divisors(2004)
[2, 3, 167]
```

If n is negative, we do *not* include -1 among the prime divisors, since -1 is not a prime number:

```
sage: prime_divisors(-100)
[2, 5]
```

For polynomials we get all irreducible factors:

```
sage: R.<x> = PolynomialRing(QQ)
sage: prime_divisors(x^12 - 1)
[x - 1, x + 1, x^2 - x + 1, x^2 + 1, x^2 + x + 1, x^4 - x^2 + 1]
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: prime_divisors(int8(-100))
[2, 5]
sage: from gmpy2 import mpz
```

(continues on next page)

(continued from previous page)

```
sage: prime_divisors(mpz(-100))
[2, 5]
```

sage.arith.misc.**prime_powers**(*start*, *stop=None*)

List of all positive primes powers between `start` and `stop`-1, inclusive. If the second argument is omitted, returns the prime powers up to the first argument.

INPUT:

- `start` - an integer. If two inputs are given, a lower bound for the returned set of prime powers. If this is the only input, then it is an upper bound.
- `stop` - an integer (default: `None`). An upper bound for the returned set of prime powers.

OUTPUT:

The set of all prime powers between `start` and `stop` or, if only one argument is passed, the set of all prime powers between 1 and `start`. The number n is a prime power if $n = p^k$, where p is a prime number and k is a positive integer. Thus, 1 is not a prime power.

EXAMPLES:

```
sage: prime_powers(20)
[2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17, 19]
sage: len(prime_powers(1000))
193
sage: len(prime_range(1000))
168

sage: a = [z for z in range(95,1234) if is_prime_power(z)]
sage: b = prime_powers(95,1234)
sage: len(b)
194
sage: len(a)
194
sage: a[:10]
[97, 101, 103, 107, 109, 113, 121, 125, 127, 128]
sage: b[:10]
[97, 101, 103, 107, 109, 113, 121, 125, 127, 128]
sage: a == b
True

sage: prime_powers(100) == [i for i in range(100) if is_prime_power(i)]
True

sage: prime_powers(10,7)
[]
sage: prime_powers(-5)
[]
sage: prime_powers(-1,3)
[2]
```

sage.arith.misc.**prime_to_m_part**(*n*, *m*)

Return the prime-to-`m` part of `n`.

This is the largest divisor of `n` that is coprime to `m`.

INPUT:

- n – Integer (nonzero)
- m – Integer

OUTPUT: Integer

EXAMPLES:

```
sage: prime_to_m_part(240,2)
15
sage: prime_to_m_part(240,3)
80
sage: prime_to_m_part(240,5)
48
sage: prime_to_m_part(43434,20)
21717
```

Note that integers also have a method with the same name:

```
sage: 240.prime_to_m_part(2)
15
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int16
sage: prime_to_m_part(int16(240), int16(2))
15
sage: from gmpy2 import mpz
sage: prime_to_m_part(mpz(240), mpz(2))
15
```

sage.arith.misc.**primes**(*start*, *stop=None*, *proof=None*)

Return an iterator over all primes between start and stop-1, inclusive. This is much slower than prime_range, but potentially uses less memory. As with *next_prime()*, the optional argument proof controls whether the numbers returned are guaranteed to be prime or not.

This command is like the Python 3 range command, except it only iterates over primes. In some cases it is better to use primes than prime_range, because primes does not build a list of all primes in the range in memory all at once. However, it is potentially much slower since it simply calls the *next_prime()* function repeatedly, and *next_prime()* is slow.

INPUT:

- start - an integer - lower bound for the primes
- stop - an integer (or infinity) optional argument - giving upper (open) bound for the primes
- proof - bool or None (default: None) If True, the function yields only proven primes. If False, the function uses a pseudo-primality test, which is much faster for really big numbers but does not provide a proof of primality. If None, uses the global default (see sage.structure.proof.proof)

OUTPUT:

- an iterator over primes from start to stop-1, inclusive

EXAMPLES:

```
sage: for p in primes(5,10):
....:     print(p)
5
7
sage: list(primes(13))
[2, 3, 5, 7, 11]
sage: list(primes(10000000000, 10000000100))
[10000000019, 10000000033, 10000000061, 10000000069, 10000000097]
sage: max(primes(10^100, 10^100+10^4, proof=False))
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000096
sage: next(p for p in primes(10^20, infinity) if is_prime(2*p+1))
100000000000000001243
```

sage.arith.misc.**primes_first_n**(*n*, *leave_pari=False*)

Return the first n primes.

INPUT:

- n - a nonnegative integer

OUTPUT:

- a list of the first n prime numbers.

EXAMPLES:

```
sage: primes_first_n(10)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
sage: len(primes_first_n(1000))
1000
sage: primes_first_n(0)
[]
```

sage.arith.misc.**primitive_root**(*n*, *check=True*)

Return a positive integer that generates the multiplicative group of integers modulo n, if one exists; otherwise, raise a `ValueError`.

A primitive root exists if $n = 4$ or $n = p^k$ or $n = 2p^k$, where p is an odd prime and k is a nonnegative number.

INPUT:

- `n` – a non-zero integer
- `check` – bool (default: True); if False, then n is assumed to be a positive integer possessing a primitive root, and behavior is undefined otherwise.

OUTPUT:

A primitive root of n. If n is prime, this is the smallest primitive root.

EXAMPLES:

```
sage: primitive_root(23)
5
sage: primitive_root(-46)
5
sage: primitive_root(25)
2
sage: print([primitive_root(p) for p in primes(100)])
```

(continues on next page)

(continued from previous page)

```
[1, 2, 2, 3, 2, 2, 3, 2, 5, 2, 3, 2, 6, 3, 5, 2, 2, 2, 2, 7, 5, 3, 2, 3, 5]
sage: primitive_root(8)
Traceback (most recent call last):
...
ValueError: no primitive root
```

Note: It takes extra work to check if n has a primitive root; to avoid this, use `check=False`, which may slightly speed things up (but could also result in undefined behavior). For example, the second call below is an order of magnitude faster than the first:

```
sage: n = 10^50 + 151   # a prime
sage: primitive_root(n)
11
sage: primitive_root(n, check=False)
11
```

sage.arith.misc.**quadratic_residues**(*n*)

Return a sorted list of all squares modulo the integer n in the range $0 \leq x < |n|$.

EXAMPLES:

```
sage: quadratic_residues(11)
[0, 1, 3, 4, 5, 9]
sage: quadratic_residues(1)
[0]
sage: quadratic_residues(2)
[0, 1]
sage: quadratic_residues(8)
[0, 1, 4]
sage: quadratic_residues(-10)
[0, 1, 4, 5, 6, 9]
sage: v = quadratic_residues(1000); len(v)
159
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: quadratic_residues(int8(11))
[0, 1, 3, 4, 5, 9]
sage: from gmpy2 import mpz
sage: quadratic_residues(mpz(11))
[0, 1, 3, 4, 5, 9]
```

sage.arith.misc.**radical**(*n*, **args*, ***kwds*)

Return the product of the prime divisors of n.

This calls `n.radical(*args, **kwds)`.

EXAMPLES:

```
sage: radical(2 * 3^2 * 5^5)
30
```

(continues on next page)

(continued from previous page)

```
sage: radical(0)
Traceback (most recent call last):
...
ArithmeticError: Radical of 0 not defined.
sage: K.<i> = QuadraticField(-1)
sage: radical(K(2))
i + 1
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: radical(int8(50))
10
sage: from gmpy2 import mpz
sage: radical(mpz(50))
10
```

sage.arith.misc.**random_prime**(*n*, *proof=None*, *lbound=2*)

Return a random prime p between `lbound` and n.

The returned prime p satisfies `lbound` $\leq p \leq n$.

The returned prime p is chosen uniformly at random from the set of prime numbers less than or equal to n.

INPUT:

- `n` - an integer >= 2.
- `proof` - bool or None (default: None) If False, the function uses a pseudo-primality test, which is much faster for really big numbers but does not provide a proof of primality. If None, uses the global default (see `sage.structure.proof.proof`)
- `lbound` - an integer >= 2, lower bound for the chosen primes

EXAMPLES:

```
sage: p = random_prime(100000)
sage: p.is_prime()
True
sage: p <= 100000
True
sage: random_prime(2)
2
```

Here we generate a random prime between 100 and 200:

```
sage: p = random_prime(200, lbound=100)
sage: p.is_prime()
True
sage: 100 <= p <= 200
True
```

If all we care about is finding a pseudo prime, then we can pass in `proof=False`

```
sage: p = random_prime(200, proof=False, lbound=100)
sage: p.is_pseudoprime()
```

(continues on next page)

(continued from previous page)

```
True
sage: 100 <= p <= 200
True
```

AUTHORS:

- Jon Hanke (2006-08-08): with standard Stein cleanup
- Jonathan Bober (2007-03-17)

sage.arith.misc.**rational_reconstruction**(*a*, *m*, *algorithm='fast'*)

This function tries to compute x/y, where x/y is a rational number in lowest terms such that the reduction of x/y modulo m is equal to a and the absolute values of x and y are both $\leq \sqrt{m/2}$. If such x/y exists, that pair is unique and this function returns it. If no such pair exists, this function raises ZeroDivisionError.

An efficient algorithm for computing rational reconstruction is very similar to the extended Euclidean algorithm. For more details, see Knuth, Vol 2, 3rd ed, pages 656-657.

INPUT:

- `a` – an integer
- `m` – a modulus
- `algorithm` – (default: 'fast')
  - `'fast'` - a fast implementation using direct GMP library calls in Cython.

OUTPUT:

Numerator and denominator n, d of the unique rational number $r = n/d$, if it exists, with n and $|d| \leq \sqrt{N/2}$. Return $(0, 0)$ if no such number exists.

The algorithm for rational reconstruction is described (with a complete nontrivial proof) on pages 656-657 of Knuth, Vol 2, 3rd ed. as the solution to exercise 51 on page 379. See in particular the conclusion paragraph right in the middle of page 657, which describes the algorithm thus:

> This discussion proves that the problem can be solved efficiently by applying Algorithm 4.5.2X with $u = m$ and $v = a$, but with the following replacement for step X2: If $v3 \leq \sqrt{m/2}$, the algorithm terminates. The pair $(x, y) = (|v2|, v3 * \text{sign}(v2))$ is then the unique solution, provided that x and y are coprime and $x \leq \sqrt{m/2}$; otherwise there is no solution. (Alg 4.5.2X is the extended Euclidean algorithm.)

Knuth remarks that this algorithm is due to Wang, Kornerup, and Gregory from around 1983.

EXAMPLES:

```
sage: m = 100000
sage: (119*inverse_mod(53,m))%m
11323
sage: rational_reconstruction(11323,m)
119/53
```

```
sage: rational_reconstruction(400,1000)
Traceback (most recent call last):
...
ArithmeticError: rational reconstruction of 400 (mod 1000) does not exist
```

```
sage: rational_reconstruction(3, 292393)
3
sage: a = Integers(292393)(45/97); a
204977
sage: rational_reconstruction(a, 292393, algorithm='fast')
45/97
sage: rational_reconstruction(293048, 292393)
Traceback (most recent call last):
...
ArithmeticError: rational reconstruction of 655 (mod 292393) does not exist
sage: rational_reconstruction(0, 0)
Traceback (most recent call last):
...
ZeroDivisionError: rational reconstruction with zero modulus
sage: rational_reconstruction(0, 1, algorithm="foobar")
Traceback (most recent call last):
...
ValueError: unknown algorithm 'foobar'
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int32
sage: rational_reconstruction(int32(3), int32(292393))
3
sage: from gmpy2 import mpz
sage: rational_reconstruction(mpz(3), mpz(292393))
3
```

sage.arith.misc.**rising_factorial**(x, a)

Return the rising factorial $(x)^a$.

The notation in the literature is a mess: often $(x)^a$, but there are many other notations: GKP: Concrete Mathematics uses $x^{\overline{a}}$.

The rising factorial is also known as the Pochhammer symbol, see Maple and Mathematica.

Definition: for integer $a \geq 0$ we have $x(x+1)\cdots(x+a-1)$. In all other cases we use the GAMMA-function: $\frac{\Gamma(x+a)}{\Gamma(x)}$.

INPUT:

- `x` – element of a ring
- `a` – a non-negative integer or
- `x and a` – any numbers

OUTPUT: the rising factorial

See also:

falling_factorial()

EXAMPLES:

```
sage: rising_factorial(10,3)
1320
```

(continues on next page)

(continued from previous page)

```
sage: rising_factorial(10,RR('3.0'))
1320.00000000000

sage: rising_factorial(10,RR('3.3'))
2826.38895824964

sage: a = rising_factorial(1+I, I); a
gamma(2*I + 1)/gamma(I + 1)
sage: CC(a)
0.266816390637832 + 0.122783354006372*I

sage: a = rising_factorial(I, 4); a
-10

sage: x = polygen(ZZ)
sage: rising_factorial(x, 4)
x^4 + 6*x^3 + 11*x^2 + 6*x
```

AUTHORS:

- Jaap Spies (2006-03-05)

sage.arith.misc.**sort_complex_numbers_for_display**(*nums*)

Given a list of complex numbers (or a list of tuples, where the first element of each tuple is a complex number), we sort the list in a "pretty" order.

Real numbers (with a zero imaginary part) come before complex numbers, and are sorted. Complex numbers are sorted by their real part unless their real parts are quite close, in which case they are sorted by their imaginary part.

This is not a useful function mathematically (not least because there is no principled way to determine whether the real components should be treated as equal or not). It is called by various polynomial root-finders; its purpose is to make doctest printing more reproducible.

We deliberately choose a cumbersome name for this function to discourage use, since it is mathematically meaningless.

EXAMPLES:

```
sage: import sage.arith.misc
sage: sort_c = sort_complex_numbers_for_display
sage: nums = [CDF(i) for i in range(3)]
sage: for i in range(3):
....:     nums.append(CDF(i + RDF.random_element(-3e-11, 3e-11),
....:                     RDF.random_element()))
....:     nums.append(CDF(i + RDF.random_element(-3e-11, 3e-11),
....:                     RDF.random_element()))
sage: shuffle(nums)
sage: nums = sort_c(nums)
sage: for i in range(len(nums)):
....:     if nums[i].imag():
....:         first_non_real = i
....:         break
....: else:
....:     first_non_real = len(nums)
```

(continues on next page)

(continued from previous page)

```
sage: assert first_non_real >= 3
sage: for i in range(first_non_real - 1):
....:     assert nums[i].real() <= nums[i + 1].real()

sage: def truncate(n):
....:     if n.real() < 1e-10:
....:         return 0
....:     else:
....:         return n.real().n(digits=9)
sage: for i in range(first_non_real, len(nums)-1):
....:     assert truncate(nums[i]) <= truncate(nums[i + 1])
....:     if truncate(nums[i]) == truncate(nums[i + 1]):
....:         assert nums[i].imag() <= nums[i+1].imag()
```

sage.arith.misc.**squarefree_divisors**(x)

Return an iterator over the squarefree divisors (up to units) of this ring element.

Depends on the output of the prime_divisors function.

Squarefree divisors of an integer are not necessarily yielded in increasing order.

INPUT:

- x – an element of any ring for which the prime_divisors function works.

EXAMPLES:

Integers with few prime divisors:

```
sage: list(squarefree_divisors(7))
[1, 7]
sage: list(squarefree_divisors(6))
[1, 2, 3, 6]
sage: list(squarefree_divisors(12))
[1, 2, 3, 6]
```

Squarefree divisors are not yielded in increasing order:

```
sage: list(squarefree_divisors(30))
[1, 2, 3, 6, 5, 10, 15, 30]
```

sage.arith.misc.**subfactorial**(n)

Subfactorial or rencontres numbers, or derangements: number of permutations of n elements with no fixed points.

INPUT:

- `n` - non negative integer

OUTPUT:

- `integer` - function value

EXAMPLES:

```
sage: subfactorial(0)
1
sage: subfactorial(1)
0
```

(continues on next page)

(continued from previous page)

```
sage: subfactorial(8)
14833
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: subfactorial(int8(8))
14833
sage: from gmpy2 import mpz
sage: subfactorial(mpz(8))
14833
```

AUTHORS:

- Jaap Spies (2007-01-23)

sage.arith.misc.**sum_of_k_squares**(k, n)

Write the integer n as a sum of k integer squares if possible; otherwise raise a `ValueError`.

INPUT:

- `k` – a non-negative integer
- `n` – an integer

OUTPUT: a tuple $(x_1, ..., x_k)$ of non-negative integers such that their squares sum to n.

EXAMPLES:

```
sage: sum_of_k_squares(2, 9634)
(15, 97)
sage: sum_of_k_squares(3, 9634)
(0, 15, 97)
sage: sum_of_k_squares(4, 9634)
(1, 2, 5, 98)
sage: sum_of_k_squares(5, 9634)
(0, 1, 2, 5, 98)
sage: sum_of_k_squares(6, 11^1111-1)
(192154008226459442538609204375863262 84, 372046451945859921742529156932675783 06,␣
↪347365481947739466585748422125613656780016108681583429709248877921686312 2,␣
↪586019179961767363354757261035179799672185073776803287636097891107462928784106157827083233032223 6
↪␣
↪204574232945581824940019198123790239925388022037307910197285434397653478513163665370946968966699 1
↪␣
↪311628095411678159849237738619458396497534696043580912225334269371611836910345930320700816649653 4
sage: sum_of_k_squares(7, 0)
(0, 0, 0, 0, 0, 0, 0)
sage: sum_of_k_squares(30,999999)
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 3, 7,
↪ 44, 999)
sage: sum_of_k_squares(1, 9)
(3,)
sage: sum_of_k_squares(1, 10)
Traceback (most recent call last):
...
ValueError: 10 is not a sum of 1 square
```

(continues on next page)

(continued from previous page)

```
sage: sum_of_k_squares(1, -10)
Traceback (most recent call last):
...
ValueError: -10 is not a sum of 1 square
sage: sum_of_k_squares(0, 9)
Traceback (most recent call last):
...
ValueError: 9 is not a sum of 0 squares
sage: sum_of_k_squares(0, 0)
()
sage: sum_of_k_squares(7, -1)
Traceback (most recent call last):
...
ValueError: -1 is not a sum of 7 squares
sage: sum_of_k_squares(-1, 0)
Traceback (most recent call last):
...
ValueError: k = -1 must be non-negative
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int16
sage: sum_of_k_squares(int16(2), int16(9634))
(15, 97)
sage: from gmpy2 import mpz
sage: sum_of_k_squares(mpz(2), mpz(9634))
(15, 97)
```

sage.arith.misc.**three_squares**(n)

Write the integer n as a sum of three integer squares if possible; otherwise raise a `ValueError`.

INPUT:

- n – an integer

OUTPUT: a tuple (a, b, c) of non-negative integers such that $n = a^2 + b^2 + c^2$ with $a <= b <= c$.

EXAMPLES:

```
sage: three_squares(389)
(1, 8, 18)
sage: three_squares(946)
(9, 9, 28)
sage: three_squares(2986)
(3, 24, 49)
sage: three_squares(7^100)
(0, 0, 17984650426474121466202803405696493492512 49)
sage: three_squares(11^111-1)
(61627416065597534015070644268 0, 90158293838573514329506074 6161,␣
↪62703823876357441403940013630653119679640999817885939472 33)
sage: three_squares(7 * 2^41)
(1048576, 2097152, 3145728)
sage: three_squares(7 * 2^42)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: 30786325577728 is not a sum of 3 squares
sage: three_squares(0)
(0, 0, 0)
sage: three_squares(-1)
Traceback (most recent call last):
...
ValueError: -1 is not a sum of 3 squares
```

ALGORITHM:

See https://schorn.ch/lagrange.html

sage.arith.misc.**trial_division**(*n*, *bound=None*)

Return the smallest prime divisor <= bound of the positive integer n, or n if there is no such prime. If the optional argument bound is omitted, then bound <= n.

INPUT:

- n - a positive integer
- bound - (optional) a positive integer

OUTPUT:

- int - a prime p=bound that divides n, or n if there is no such prime.

EXAMPLES:

```
sage: trial_division(15)
3
sage: trial_division(91)
7
sage: trial_division(11)
11
sage: trial_division(387833, 300)
387833
sage: # 300 is not big enough to split off a
sage: # factor, but 400 is.
sage: trial_division(387833, 400)
389
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: trial_division(int8(91))
7
sage: from gmpy2 import mpz
sage: trial_division(mpz(91))
7
```

sage.arith.misc.**two_squares**(*n*)

Write the integer n as a sum of two integer squares if possible; otherwise raise a ValueError.

INPUT:

- n – an integer

OUTPUT: a tuple (a, b) of non-negative integers such that $n = a^2 + b^2$ with $a <= b$.

EXAMPLES:

```
sage: two_squares(389)
(10, 17)
sage: two_squares(21)
Traceback (most recent call last):
...
ValueError: 21 is not a sum of 2 squares
sage: two_squares(21^2)
(0, 21)
sage: a,b = two_squares(100000000000000000129); a,b
(4418521500, 8970878873)
sage: a^2 + b^2
100000000000000000129
sage: two_squares(2^222+1)
(25380165950470862199142171245052 1, 25837127132133548984903046450 18692)
sage: two_squares(0)
(0, 0)
sage: two_squares(-1)
Traceback (most recent call last):
...
ValueError: -1 is not a sum of 2 squares
```

ALGORITHM:

See https://schorn.ch/lagrange.html

sage.arith.misc.**valuation**(*m*, **args*, ***kwds*)
Return the valuation of `m`.

This function simply calls the m.valuation() method. See the documentation of m.valuation() for a more precise description.

Note that the use of this functions is discouraged as it is better to use m.valuation() directly.

Note: This is not always a valuation in the mathematical sense. For more information see: sage.rings.finite_rings.integer_mod.IntegerMod_int.valuation

---

EXAMPLES:

```
sage: valuation(512,2)
9
sage: valuation(1,2)
0
sage: valuation(5/9, 3)
-2
```

Valuation of 0 is defined, but valuation with respect to 0 is not:

```
sage: valuation(0,7)
+Infinity
sage: valuation(3,0)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: You can only compute the valuation with respect to a integer larger␣
↪than 1.
```

Here are some other examples:

```
sage: valuation(100,10)
2
sage: valuation(200,10)
2
sage: valuation(243,3)
5
sage: valuation(243*10007,3)
5
sage: valuation(243*10007,10007)
1
sage: y = QQ['y'].gen()
sage: valuation(y^3, y)
3
sage: x = QQ[['x']].gen()
sage: valuation((x^3-x^2)/(x-4))
2
sage: valuation(4r,2r)
2
sage: valuation(1r,1r)
Traceback (most recent call last):
...
ValueError: You can only compute the valuation with respect to a integer larger␣
↪than 1.
sage: from numpy import int16
sage: valuation(int16(512), int16(2))
9
sage: from gmpy2 import mpz
sage: valuation(mpz(512), mpz(2))
9
```

sage.arith.misc.**xgcd**(a, b)

Return a triple `(g,s,t)` such that $g = s \cdot a + t \cdot b = \gcd(a, b)$.

Note: One exception is if a and b are not in a principal ideal domain (see Wikipedia article Principal_ideal_domain), e.g., they are both polynomials over the integers. Then this function can't in general return `(g,s,t)` as above, since they need not exist. Instead, over the integers, we first multiply g by a divisor of the resultant of a/g and b/g, up to sign.

INPUT:

- `a, b` - integers or more generally, element of a ring for which the xgcd make sense (e.g. a field or univariate polynomials).

OUTPUT:

- `g, s, t` - such that $g = s \cdot a + t \cdot b$

Note: There is no guarantee that the returned cofactors (s and t) are minimal.

EXAMPLES:

```
sage: xgcd(56, 44)
(4, 4, -5)
sage: 4*56 + (-5)*44
4

sage: g, a, b = xgcd(5/1, 7/1); g, a, b
(1, 3, -2)
sage: a*(5/1) + b*(7/1) == g
True

sage: x = polygen(QQ)
sage: xgcd(x^3 - 1, x^2 - 1)
(x - 1, 1, -x)

sage: K.<g> = NumberField(x^2-3)
sage: g.xgcd(g+2)
(1, 1/3*g, 0)

sage: R.<a,b> = K[]
sage: S.<y> = R.fraction_field()[]
sage: xgcd(y^2, a*y+b)
(1, a^2/b^2, ((-a)/b^2)*y + 1/b)
sage: xgcd((b+g)*y^2, (a+g)*y+b)
(1, (a^2 + (2*g)*a + 3)/(b^3 + g*b^2), ((-a + (-g))/b^2)*y + 1/b)
```

Here is an example of a xgcd for two polynomials over the integers, where the linear combination is not the gcd but the gcd multiplied by the resultant:

```
sage: R.<x> = ZZ[]
sage: gcd(2*x*(x-1), x^2)
x
sage: xgcd(2*x*(x-1), x^2)
(2*x, -1, 2)
sage: (2*(x-1)).resultant(x)
2
```

Tests with numpy and gmpy2 types:

```
sage: from numpy import int8
sage: xgcd(4,int8(8))
(4, 1, 0)
sage: xgcd(int8(4),int8(8))
(4, 1, 0)
sage: from gmpy2 import mpz
sage: xgcd(mpz(4), mpz(8))
(4, 1, 0)
sage: xgcd(4, mpz(8))
(4, 1, 0)
```

sage.arith.misc.**xkcd**(*n=''*)

This function is similar to the xgcd function, but behaves in a completely different way.

See https://xkcd.com/json.html for more details.

INPUT:

- n – an integer (optional)

OUTPUT: a fragment of HTML

EXAMPLES:

```
sage: xkcd(353)  # optional - internet
<h1>Python</h1><img src="https://imgs.xkcd.com/comics/python.png" title="I wrote 20␣
↪short programs in Python yesterday.  It was wonderful.  Perl, I'm leaving you.">
↪<div>Source: <a href="http://xkcd.com/353" target="_blank">http://xkcd.com/353</a>
↪</div>
```

sage.arith.misc.**xlcm**(*m*, *n*)

Extended lcm function: given two positive integers m, n, returns a triple (l, m_1, n_1) such that $l = \mathrm{lcm}(m, n) = m_1 \cdot n_1$ where $m_1 | m$, $n_1 | n$ and $\gcd(m_1, n_1) = 1$, all with no factorization.

Used to construct an element of order l from elements of orders m, n in any group: see sage/groups/generic.py for examples.

EXAMPLES:

```
sage: xlcm(120,36)
(360, 40, 9)
```

See also:

- sage.sets.integer_range
- sage.sets.positive_integers
- sage.sets.non_negative_integers
- sage.sets.primes

CHAPTER

TWO

RATIONALS

2.1 Field $\mathbb{Q}$ of Rational Numbers

The class *RationalField* represents the field $\mathbb{Q}$ 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
```

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().
- Anna Haensch (2018-03): Added function quadratic_defect()

sage.rings.rational_field.**Q = Rational Field**

sage.rings.rational_field.**QQ = Rational Field**

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

Bases: sage.misc.fast_methods.Singleton, sage.rings.number_field.number_field_base.NumberField

The class RationalField represents the field $\mathbf{Q}$ of rational numbers.

EXAMPLES:

```
sage: a = 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
0.25384186085591068433775892333...
```

(continues on next page)

(continued from previous page)

```
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
```

absolute_polynomial()
Return a defining polynomial of $\mathbf{Q}$, as for other number fields.

This is also aliased to `self.defining_polynomial()` and `self.absolute_polynomial()`.

EXAMPLES:

```
sage: QQ.polynomial()
x
```

algebraic_closure()
Return the algebraic closure of self (which is $\overline{\mathbf{Q}}$).

EXAMPLES:

```
sage: QQ.algebraic_closure()
Algebraic Field
```

automorphisms()
Return all Galois automorphisms of `self`.

OUTPUT:

- a sequence containing just the identity morphism

EXAMPLES:

```
sage: QQ.automorphisms()
[
Ring endomorphism of Rational Field
  Defn: 1 |--> 1
]
```

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)
```

defining_polynomial()

Return a defining polynomial of $\mathbf{Q}$, as for other number fields.

This is also aliased to `self.defining_polynomial()` and `self.absolute_polynomial()`.

EXAMPLES:

```
sage: QQ.polynomial()
x
```

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)
[Identity endomorphism of Rational Field]
sage: QQ.embeddings(CyclotomicField(5))
[Coercion map:
   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*, ***kwds*)
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 $\mathbf{Q}$.

There is only the 0-th generator which is 1.

EXAMPLES:

```
sage: QQ.gen()
1
```

gens()
Return a tuple of generators of $\mathbf{Q}$ which is only `(1,)`.

EXAMPLES:

```
sage: QQ.gens()
(1,)
```

hilbert_symbol_negative_at_S(*S*, *b*, *check=True*)
Returns an integer that has a negative Hilbert symbol with respect to a given rational number and a given set of primes (or places).

The function is algorithm 3.4.1 in [Kir2016]. It finds an integer a that has negative Hilbert symbol with respect to a given rational number exactly at a given set of primes (or places).

INPUT:

- `S` – a list of rational primes, the infinite place as real embedding of $\mathbf{Q}$ or as -1
- `b` – a non-zero rational number which is a non-square locally at every prime in `S`.
- `check` – `bool` (default:`True`) perform additional checks on input and confirm the output.

OUTPUT:

- An integer a that has negative Hilbert symbol $(a, b)_p$ for every place p in S and no other place.

EXAMPLES:

```
sage: QQ.hilbert_symbol_negative_at_S([-1,5,3,2,7,11,13,23], -10/7)
-9867
sage: QQ.hilbert_symbol_negative_at_S([3, 5, QQ.places()[0], 11], -15)
-33
sage: QQ.hilbert_symbol_negative_at_S([3, 5], 2)
15
```

AUTHORS:

- Simon Brandhorst, Juanita Duque, Anna Haensch, Manami Roy, Sandi Rudzinski (10-24-2017)

is_absolute()
$\mathbf{Q}$ is an absolute extension of $\mathbf{Q}$.

EXAMPLES:

```
sage: QQ.is_absolute()
True
```

is_prime_field()
Return `True` since $\mathbf{Q}$ is a prime field.

EXAMPLES:

```
sage: QQ.is_prime_field()
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
```

(continues on next page)

(continued from previous page)

```
  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 |--> 1.00000000000000000000000000000000000000000000000000000000000000000000000000000]
```

polynomial()

Return a defining polynomial of $\mathbf{Q}$, as for other number fields.

This is also aliased to `self.defining_polynomial()` and `self.absolute_polynomial()`.

EXAMPLES:

```
sage: QQ.polynomial()
x
```

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))
[]
```

quadratic_defect(*a*, *p*, *check=True*)

Return the valuation of the quadratic defect of a at p.

INPUT:

- `a` – an element of `self`

- p – a prime ideal or a prime number
- check – (default: True); check if p is prime

REFERENCE:

[Kir2016]

EXAMPLES:

```
sage: QQ.quadratic_defect(0, 7)
+Infinity
sage: QQ.quadratic_defect(5, 7)
0
sage: QQ.quadratic_defect(5, 2)
2
sage: QQ.quadratic_defect(5, 5)
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().parent() is QQ
True
sage: while QQ.random_element() != 0:
....:     pass
sage: while QQ.random_element() != -1/2:
....:     pass
```

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: q = QQ.random_element(5, 10)
sage: -5/1 <= q <= 5/1
True
sage: q.denominator() <= 10
True
sage: q.numerator() <= 5
True
```

Extra positional or keyword arguments are passed through:

```
sage: QQ.random_element(distribution='1/n').parent() is QQ
True
sage: QQ.random_element(distribution='1/n').parent() is QQ
True
```

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_generators(*S*, *m*, *proof=True*, *orders=False*)

Return generators of 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.

See also:

RationalField.selmer_space(), which gives additional output when $m = p$ is prime: as well as generators, it gives an abstract vector space over $GF(p)$ isomorphic to $\mathbf{Q}(S, p)$ and maps implementing the isomorphism between this space and $\mathbf{Q}(S, p)$ as a subgroup of $\mathbf{Q}^*/(\mathbf{Q}^*)^p$.

EXAMPLES:

```
sage: QQ.selmer_generators((), 2)
[-1]
sage: QQ.selmer_generators((3,), 2)
[-1, 3]
sage: QQ.selmer_generators((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_generators((2,3,5,7,), 2, orders=True)
([-1, 2, 3, 5, 7], [2, 2, 2, 2, 2])
sage: QQ.selmer_generators((2,3,5,7,), 3, orders=True)
([2, 3, 5, 7], [3, 3, 3, 3])
```

selmer_group(**args*, ***kwds*)

Deprecated: Use *selmer_generators()* instead. See trac ticket #31345 for details.

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_generators()* 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]
```

selmer_space(*S*, *p*, *proof=None*)

Compute the group $\mathbf{Q}(S, p)$ as a vector space with maps to and from $\mathbf{Q}^*$.

INPUT:

- `S` – a list of prime numbers
- `p` – a prime number

OUTPUT:

(tuple) `QSp`, `QSp_gens`, `from_QSp`, `to_QSp` where

- `QSp` is an abstract vector space over $GF(p)$ isomorphic to $\mathbf{Q}(S, p)$;
- `QSp_gens` is a list of elements of $\mathbf{Q}^*$ generating $\mathbf{Q}(S, p)$;
- `from_QSp` is a function from `QSp` to $\mathbf{Q}^*$ implementing the isomorphism from the abstract $\mathbf{Q}(S, p)$ to $\mathbf{Q}(S, p)$ as a subgroup of $\mathbf{Q}^*/(\mathbf{Q}^*)^p$;
- `to_QSP` is a partial function from $\mathbf{Q}^*$ to `QSp`, defined on elements a whose image in $\mathbf{Q}^*/(\mathbf{Q}^*)^p$ lies in $\mathbf{Q}(S, p)$, mapping them via the inverse isomorphism to the abstract vector space `QSp`.

The group $\mathbf{Q}(S, p)$ is the finite subgroup of $\mathbf{Q}^*/(\mathbf{Q}^*)^p$ consisting of elements whose valuation at all primes not in S is a multiple of p. It contains the subgroup of those $a \in \mathbf{Q}^*$ such that $\mathbf{Q}(\sqrt[p]{a})/\mathbf{Q}$ is unramified at all primes of $\mathbf{Q}$ outside of S, but may contain it properly when p is not in S.

EXAMPLES:

When S is empty, $\mathbf{Q}(S, p)$ is only nontrivial for $p = 2$:

```
sage: QS2, QS2gens, fromQS2, toQS2 = QQ.selmer_space([], 2)
sage: QS2
Vector space of dimension 1 over Finite Field of size 2
sage: QS2gens
[-1]

sage: all(QQ.selmer_space([], p)[0].dimension() == 0 for p in primes(3,10))
True
```

In general there is one generator for each $p \in S$, and an additional generator of -1 when $p = 2$:

```
sage: QS2, QS2gens, fromQS2, toQS2 = QQ.selmer_space([5,7], 2)
sage: QS2
Vector space of dimension 3 over Finite Field of size 2
sage: QS2gens
[5, 7, -1]
sage: toQS2(-7)
(0, 1, 1)
sage: fromQS2((0,1,1))
-7
```

The map `fromQS2` is only well-defined modulo p'th powers (in this case, modulo squares):

```
sage: toQS2(-5/7)
(1, 1, 1)
sage: fromQS2((1,1,1))
-35
sage: ((-5/7)/(-35)).is_square()
True
```

The map `toQS2` is not defined on all of $\mathbf{Q}^*$, only on those numbers which are squares away from 5 and 7:

```
sage: toQS2(210)
Traceback (most recent call last):
...
ValueError: argument 210 should have valuations divisible by 2 at all primes in␣
↪[5, 7]
```

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)
```

some_elements()

Return some elements of $\mathbf{Q}$.

See `TestSuite()` for a typical use case.

OUTPUT:

An iterator over 100 elements of $\mathbf{Q}$.

EXAMPLES:

```
sage: tuple(QQ.some_elements())
(1/2, -1/2, 2, -2,
 0, 1, -1, 42,
 2/3, -2/3, 3/2, -3/2,
 4/5, -4/5, 5/4, -5/4,
 6/7, -6/7, 7/6, -7/6,
 8/9, -8/9, 9/8, -9/8,
 10/11, -10/11, 11/10, -11/10,
 12/13, -12/13, 13/12, -13/12,
 14/15, -14/15, 15/14, -15/14,
```

(continues on next page)

(continued from previous page)

```
16/17, -16/17, 17/16, -17/16,
18/19, -18/19, 19/18, -19/18,
20/441, -20/441, 441/20, -441/20,
22/529, -22/529, 529/22, -529/22,
24/625, -24/625, 625/24, -625/24,
...)
```

valuation(*p*)

Return the discrete valuation with uniformizer `p`.

EXAMPLES:

```
sage: v = QQ.valuation(3); v
3-adic valuation
sage: v(1/3)
-1
```

See also:

`NumberField_generic.valuation()`, *IntegerRing_class.valuation()*

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)
```

(continues on next page)

(continued from previous page)

```
True
sage: is_RF(ZZ)
False
```

2.2 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 __bool__ ; 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
- Vincent Delecroix (2017-05-03): faster integer-rational comparison
- Vincent Klein (2017-05-11): add __mpq__() to class Rational
- Vincent Klein (2017-05-22): Rational constructor support gmpy2.mpq or gmpy2.mpz parameter. Add __mpz__ to class Rational.

class sage.rings.rational.**Q_to_Z**

Bases: sage.categories.map.Map

A morphism from $\mathbf{Q}$ to $\mathbf{Z}$.

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)
<class '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 fractions:

```
sage: import fractions
sage: f = fractions.Fraction(1r, 2r)
sage: Rational(f)
1/2
```

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
```

(continues on next page)

(continued from previous page)

```
sage: QQ(np.float16('12'))
12
```

Conversions from gmpy2:

```
sage: from gmpy2 import *
sage: QQ(mpq('3/4'))
3/4
sage: QQ(mpz(42))
42
sage: Rational(mpq(2/3))
2/3
sage: Rational(mpz(5))
5
```

absolute_norm()

Return 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
```

as_integer_ratio()

Return the pair `(self.numerator(), self.denominator())`.

EXAMPLES:

```
sage: x = -12/29
sage: x.as_integer_ratio()
(-12, 29)
```

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='x'*)

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
```

The default is var='x'. (trac ticket #20967):

```
sage: a = QQ(2); a.charpoly('x')
x - 2
```

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
```

(continues on next page)

(continued from previous page)

```
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, 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()

Return the denominator of this rational number. denom is an alias of denominator.

EXAMPLES:

```
sage: x = -5/11
sage: x.denominator()
11

sage: x = 9/3
sage: x.denominator()
1

sage: x = 5/13
sage: x.denom()
13
```

denominator()

Return the denominator of this rational number. denom is an alias of denominator.

EXAMPLES:

```
sage: x = -5/11
sage: x.denominator()
11

sage: x = 9/3
sage: x.denominator()
1

sage: x = 5/13
sage: x.denom()
13
```

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: factorization of 0 is 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*)

Return 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*)

Return 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*)

Return 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()

Return 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
```

(continues on next page)

(continued from previous page)

```
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)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
NotImplementedError: is_norm is not implemented unconditionally for norms from␣
↪non-Galois number fields
sage: 7.is_norm(K, proof=False)
False
```

AUTHORS:

- Craig Citro (2008-04-05)
- Marco Streng (2010-12-03)

is_nth_power(*n*)

Return `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*, *check=True*)

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
- check – (default: True); check if p is prime

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
```

is_perfect_power(*expected_value=False*)

Return 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_rational()

Return True since this is a rational number.

EXAMPLES:

```
sage: (3/4).is_rational()
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*)

Return 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*)

Return 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
```

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

Return the log of `self`.

INPUT:

- `m` – the base (default: natural log base e)
- `prec` – integer (optional); the precision in bits

OUTPUT:

When `prec` is not given, the log as an element in symbolic ring unless the logarithm is exact. Otherwise the log is a `RealField` approximation to `prec` bit precision.

EXAMPLES:

```
sage: (124/345).log(5)
log(124/345)/log(5)
sage: (124/345).log(5,100)
-0.63578895682825611710391773754
sage: log(QQ(125))
3*log(5)
sage: log(QQ(125), 5)
3
sage: log(QQ(125), 3)
3*log(5)/log(3)
sage: QQ(8).log(1/2)
-3
sage: (1/8).log(1/2)
```

(continues on next page)

(continued from previous page)

```
3
sage: (1/2).log(1/8)
1/3
sage: (1/2).log(8)
-1/3
sage: (16/81).log(8/27)
4/3
sage: (8/27).log(16/81)
3/4
sage: log(27/8, 16/81)
-3/4
sage: log(16/81, 27/8)
-4/3
sage: (125/8).log(5/2)
3
sage: (125/8).log(5/2,prec=53)
3.00000000000000
```

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()
Return 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. numer is an alias of numerator.

EXAMPLES:

```
sage: x = 5/11
sage: x.numerator()
5

sage: x = 9/3
sage: x.numerator()
3

sage: x = -5/11
sage: x.numer()
-5
```

numerator()

Return the numerator of this rational number. numer is an alias of numerator.

EXAMPLES:

```
sage: x = 5/11
sage: x.numerator()
5

sage: x = 9/3
sage: x.numerator()
3

sage: x = -5/11
sage: x.numer()
-5
```

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 $d = 2^a 5^b m$ where m is coprime to 10, then $s = \max(a, b)$ and t is the order of 10 modulo m.

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)
3.14150943396226415094339622641509433962264150943396226415094
```

prime_to_S_part(*S=[]*)

Return `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()

Return the real part of `self`, which is `self`.

EXAMPLES:

```
sage: (1/2).real()
1/2
```

relative_norm()

Return 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'*)

Return 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()

Return 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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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()

Return 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)

Return 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**.

is_surjective()

Return whether this morphism is surjective.

EXAMPLES:

```
sage: QQ.coerce_map_from(ZZ).is_surjective()
False
```

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 Python 2 `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 nonnegative real root is taken for even denominators.

INPUT:

- a – an Integer
- b – a nonnegative 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
```

TESTS (trac ticket #11228):

```
sage: integer_rational_power(-10, QQ(2))
100
sage: integer_rational_power(0, QQ(0))
1
```

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('5')
False
```

class sage.rings.rational.**long_to_Q**

Bases: sage.categories.morphism.Morphism

A morphism from Python 2 long/Python 3 int to $\mathbf{Q}$.

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)
```

CHAPTER

THREE

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

a

sage.arith.functions, 71
sage.arith.misc, 77
sage.arith.multi_modular, 74
sage.arith.power, 73

r

sage.rings.bernmm, 61
sage.rings.bernoulli_mod_p, 62
sage.rings.factorint, 65
sage.rings.fast_arith, 68
sage.rings.integer, 13
sage.rings.integer_ring, 1
sage.rings.rational, 159
sage.rings.rational_field, 145
sage.rings.sum_of_squares, 69

INDEX

Symbols

__pow__() (*sage.rings.integer.Integer method*), 16

A

absolute_degree() (*sage.rings.integer_ring.IntegerRing_class method*), 4
absolute_degree() (*sage.rings.rational_field.RationalField method*), 147
absolute_discriminant() (*sage.rings.rational_field.RationalField method*), 147
absolute_norm() (*sage.rings.rational.Rational method*), 161
absolute_polynomial() (*sage.rings.rational_field.RationalField method*), 147
additive_order() (*sage.rings.integer.Integer method*), 17
additive_order() (*sage.rings.rational.Rational method*), 161
algdep() (*in module sage.arith.misc*), 87
algebraic_closure() (*sage.rings.rational_field.RationalField method*), 147
algebraic_dependency() (*in module sage.arith.misc*), 89
arith_int (*class in sage.rings.fast_arith*), 68
arith_llong (*class in sage.rings.fast_arith*), 68
as_integer_ratio() (*sage.rings.integer.Integer method*), 17
as_integer_ratio() (*sage.rings.rational.Rational method*), 161
aurifeuillian() (*in module sage.rings.factorint*), 65
automorphisms() (*sage.rings.rational_field.RationalField method*), 147

B

balanced_digits() (*sage.rings.integer.Integer method*), 17
bernmm_bern_modp() (*in module sage.rings.bernmm*), 61
bernmm_bern_rat() (*in module sage.rings.bernmm*), 61
bernoulli() (*in module sage.arith.misc*), 91
bernoulli_mod_p() (*in module sage.rings.bernoulli_mod_p*), 62
bernoulli_mod_p_single() (*in module sage.rings.bernoulli_mod_p*), 63
binary() (*sage.rings.integer.Integer method*), 18
binomial() (*in module sage.arith.misc*), 91
binomial() (*sage.rings.integer.Integer method*), 18
binomial_coefficients() (*in module sage.arith.misc*), 93
bit_length() (*sage.rings.integer.Integer method*), 19
bits() (*sage.rings.integer.Integer method*), 19

C

ceil() (*sage.rings.integer.Integer method*), 20
ceil() (*sage.rings.rational.Rational method*), 161
characteristic() (*sage.rings.integer_ring.IntegerRing_class method*), 5
characteristic() (*sage.rings.rational_field.RationalField method*), 147
charpoly() (*sage.rings.rational.Rational method*), 162
class_number() (*sage.rings.integer.Integer method*), 20
class_number() (*sage.rings.rational_field.RationalField method*), 148
completion() (*sage.rings.integer_ring.IntegerRing_class method*), 5
completion() (*sage.rings.rational_field.RationalField method*), 148
complex_embedding() (*sage.rings.rational_field.RationalField method*), 148
conjugate() (*sage.rings.integer.Integer method*), 20
conjugate() (*sage.rings.rational.Rational method*), 162
construction() (*sage.rings.rational_field.RationalField method*), 148
content() (*sage.rings.rational.Rational method*), 162
continuant() (*in module sage.arith.misc*), 93
continued_fraction() (*sage.rings.rational.Rational method*), 163
continued_fraction_list() (*sage.rings.rational.Rational method*), 163

coprime_integers() (*sage.rings.integer.Integer method*), 21
CRT() (*in module sage.arith.misc*), 77
crt() (*in module sage.arith.misc*), 94
crt() (*sage.arith.multi_modular.MultiModularBasis_base method*), 74
crt() (*sage.rings.integer.Integer method*), 21
CRT_basis() (*in module sage.arith.misc*), 79
crt_basis() (*in module sage.rings.integer_ring*), 12
CRT_list() (*in module sage.arith.misc*), 79
CRT_vectors() (*in module sage.arith.misc*), 80

D

dedekind_psi() (*in module sage.arith.misc*), 96
dedekind_sum() (*in module sage.arith.misc*), 97
defining_polynomial() (*sage.rings.rational_field.RationalField method*), 148
degree() (*sage.rings.integer_ring.IntegerRing_class method*), 5
degree() (*sage.rings.rational_field.RationalField method*), 149
denom() (*sage.rings.rational.Rational method*), 164
denominator() (*sage.rings.integer.Integer method*), 21
denominator() (*sage.rings.rational.Rational method*), 164
differences() (*in module sage.arith.misc*), 98
digits() (*sage.rings.integer.Integer method*), 22
discriminant() (*sage.rings.rational_field.RationalField method*), 149
divide_knowing_divisible_by() (*sage.rings.integer.Integer method*), 23
divides() (*sage.rings.integer.Integer method*), 24
divisors() (*in module sage.arith.misc*), 99
divisors() (*sage.rings.integer.Integer method*), 24

E

embeddings() (*sage.rings.rational_field.RationalField method*), 149
eratosthenes() (*in module sage.arith.misc*), 100
euclidean_degree() (*sage.rings.integer.Integer method*), 25
Euler_Phi (*class in sage.arith.misc*), 81
exact_log() (*sage.rings.integer.Integer method*), 25
exp() (*sage.rings.integer.Integer method*), 26
extend_with_primes() (*sage.arith.multi_modular.MultiModularBasis_base method*), 74
extension() (*sage.rings.integer_ring.IntegerRing_class method*), 5
extension() (*sage.rings.rational_field.RationalField method*), 149

F

factor() (*in module sage.arith.misc*), 100
factor() (*sage.rings.integer.Integer method*), 27
factor() (*sage.rings.rational.Rational method*), 164
factor_aurifeuillian() (*in module sage.rings.factorint*), 66
factor_cunningham() (*in module sage.rings.factorint*), 66
factor_trial_division() (*in module sage.rings.factorint*), 67
factor_using_pari() (*in module sage.rings.factorint*), 67
factorial() (*in module sage.arith.misc*), 102
factorial() (*sage.rings.integer.Integer method*), 28
falling_factorial() (*in module sage.arith.misc*), 103
floor() (*sage.rings.integer.Integer method*), 29
floor() (*sage.rings.rational.Rational method*), 165
four_squares() (*in module sage.arith.misc*), 104
four_squares_pyx() (*in module sage.rings.sum_of_squares*), 69
frac() (*in module sage.rings.rational_field*), 158
fraction_field() (*sage.rings.integer_ring.IntegerRing_class method*), 6
free_integer_pool() (*in module sage.rings.integer*), 59
fundamental_discriminant() (*in module sage.arith.misc*), 105

G

gamma() (*sage.rings.integer.Integer method*), 29
gamma() (*sage.rings.rational.Rational method*), 165
gauss_sum() (*in module sage.arith.misc*), 105
GCD() (*in module sage.arith.misc*), 82
gcd() (*in module sage.arith.misc*), 106
gcd_int() (*sage.rings.fast_arith.arith_int method*), 68
GCD_list() (*in module sage.rings.integer*), 14
gcd_longlong() (*sage.rings.fast_arith.arith_llong method*), 68
gen() (*sage.rings.integer_ring.IntegerRing_class method*), 6
gen() (*sage.rings.rational_field.RationalField method*), 149
generic_power() (*in module sage.arith.power*), 73
gens() (*sage.rings.integer_ring.IntegerRing_class method*), 6
gens() (*sage.rings.rational_field.RationalField method*), 150
get_gcd() (*in module sage.arith.misc*), 107
get_inverse_mod() (*in module sage.arith.misc*), 107
global_height() (*sage.rings.integer.Integer method*), 29
global_height() (*sage.rings.rational.Rational method*), 165

global_height_arch() (*sage.rings.rational.Rational method*), 166
global_height_non_arch() (*sage.rings.rational.Rational method*), 166

H

height() (*sage.rings.rational.Rational method*), 167
hex() (*sage.rings.integer.Integer method*), 30
hilbert_conductor() (*in module sage.arith.misc*), 108
hilbert_conductor_inverse() (*in module sage.arith.misc*), 108
hilbert_symbol() (*in module sage.arith.misc*), 109
hilbert_symbol_negative_at_S() (*sage.rings.rational_field.RationalField method*), 150

I

imag() (*sage.rings.integer.Integer method*), 30
imag() (*sage.rings.rational.Rational method*), 167
int_to_Q (*class in sage.rings.rational*), 183
int_to_Z (*class in sage.rings.integer*), 59
Integer (*class in sage.rings.integer*), 15
integer_ceil() (*in module sage.arith.misc*), 110
integer_floor() (*in module sage.arith.misc*), 110
integer_rational_power() (*in module sage.rings.rational*), 183
integer_trunc() (*in module sage.arith.misc*), 111
IntegerRing() (*in module sage.rings.integer_ring*), 1
IntegerRing_class (*class in sage.rings.integer_ring*), 1
IntegerWrapper (*class in sage.rings.integer*), 59
inverse_mod() (*in module sage.arith.misc*), 111
inverse_mod() (*sage.rings.integer.Integer method*), 30
inverse_mod_int() (*sage.rings.fast_arith.arith_int method*), 68
inverse_mod_longlong() (*sage.rings.fast_arith.arith_llong method*), 68
inverse_of_unit() (*sage.rings.integer.Integer method*), 31
is_absolute() (*sage.rings.rational_field.RationalField method*), 150
is_field() (*sage.rings.integer_ring.IntegerRing_class method*), 6
is_Integer() (*in module sage.rings.integer*), 60
is_integer() (*sage.rings.integer.Integer method*), 31
is_integer() (*sage.rings.rational.Rational method*), 168
is_IntegerRing() (*in module sage.rings.integer_ring*), 12
is_integral() (*sage.rings.integer.Integer method*), 31
is_integral() (*sage.rings.rational.Rational method*), 168
is_integrally_closed() (*sage.rings.integer_ring.IntegerRing_class method*), 6
is_irreducible() (*sage.rings.integer.Integer method*), 32
is_noetherian() (*sage.rings.integer_ring.IntegerRing_class method*), 6
is_norm() (*sage.rings.integer.Integer method*), 32
is_norm() (*sage.rings.rational.Rational method*), 169
is_nth_power() (*sage.rings.rational.Rational method*), 170
is_one() (*sage.rings.integer.Integer method*), 32
is_one() (*sage.rings.rational.Rational method*), 170
is_padic_square() (*sage.rings.rational.Rational method*), 170
is_perfect_power() (*sage.rings.integer.Integer method*), 32
is_perfect_power() (*sage.rings.rational.Rational method*), 171
is_power_of() (*sage.rings.integer.Integer method*), 33
is_power_of_two() (*in module sage.arith.misc*), 111
is_prime() (*in module sage.arith.misc*), 112
is_prime() (*sage.rings.integer.Integer method*), 34
is_prime_field() (*sage.rings.rational_field.RationalField method*), 150
is_prime_power() (*in module sage.arith.misc*), 113
is_prime_power() (*sage.rings.integer.Integer method*), 35
is_pseudoprime() (*in module sage.arith.misc*), 113
is_pseudoprime() (*sage.rings.integer.Integer method*), 36
is_pseudoprime_power() (*in module sage.arith.misc*), 114
is_pseudoprime_power() (*sage.rings.integer.Integer method*), 37
is_Rational() (*in module sage.rings.rational*), 184
is_rational() (*sage.rings.integer.Integer method*), 37
is_rational() (*sage.rings.rational.Rational method*), 172
is_RationalField() (*in module sage.rings.rational_field*), 158
is_S_integral() (*sage.rings.rational.Rational method*), 167
is_S_unit() (*sage.rings.rational.Rational method*), 168
is_square() (*in module sage.arith.misc*), 115
is_square() (*sage.rings.integer.Integer method*), 37
is_square() (*sage.rings.rational.Rational method*), 172
is_squarefree() (*in module sage.arith.misc*), 116
is_squarefree() (*sage.rings.integer.Integer method*), 38
is_sum_of_two_squares_pyx() (*in module sage.rings.sum_of_squares*), 70
is_surjective() (*sage.rings.rational.Z_to_Q method*), 183

is_unit() (*sage.rings.integer.Integer method*), 38
isqrt() (*sage.rings.integer.Integer method*), 38

J

jacobi() (*sage.rings.integer.Integer method*), 38
jacobi_symbol() (*in module sage.arith.misc*), 116

K

kronecker() (*in module sage.arith.misc*), 117
kronecker() (*sage.rings.integer.Integer method*), 39
kronecker_symbol() (*in module sage.arith.misc*), 118
krull_dimension() (*sage.rings.integer_ring.IntegerRing_class method*), 7

L

lcm() (*in module sage.arith.functions*), 72
LCM_list() (*in module sage.arith.functions*), 71
legendre_symbol() (*in module sage.arith.misc*), 119
list() (*sage.arith.multi_modular.MultiModularBasis_base method*), 75
list() (*sage.rings.integer.Integer method*), 39
list() (*sage.rings.rational.Rational method*), 172
local_height() (*sage.rings.rational.Rational method*), 172
local_height_arch() (*sage.rings.rational.Rational method*), 173
log() (*sage.rings.integer.Integer method*), 39
log() (*sage.rings.rational.Rational method*), 173
long_to_Q (*class in sage.rings.rational*), 185
long_to_Z (*class in sage.rings.integer*), 60

M

make_integer() (*in module sage.rings.integer*), 60
make_rational() (*in module sage.rings.rational*), 185
maximal_order() (*sage.rings.rational_field.RationalField method*), 151
minpoly() (*sage.rings.rational.Rational method*), 174
mod_ui() (*sage.rings.rational.Rational method*), 174
module
    sage.arith.functions, 71
    sage.arith.misc, 77
    sage.arith.multi_modular, 74
    sage.arith.power, 73
    sage.rings.bernmm, 61
    sage.rings.bernoulli_mod_p, 62
    sage.rings.factorint, 65
    sage.rings.fast_arith, 68
    sage.rings.integer, 13
    sage.rings.integer_ring, 1
    sage.rings.rational, 159
    sage.rings.rational_field, 145
    sage.rings.sum_of_squares, 69
Moebius (*class in sage.arith.misc*), 83
mqrr_rational_reconstruction() (*in module sage.arith.misc*), 119
multifactorial() (*sage.rings.integer.Integer method*), 41
MultiModularBasis (*class in sage.arith.multi_modular*), 74
MultiModularBasis_base (*class in sage.arith.multi_modular*), 74
multinomial() (*in module sage.arith.misc*), 120
multinomial_coefficients() (*in module sage.arith.misc*), 120
multiplicative_order() (*sage.rings.integer.Integer method*), 41
multiplicative_order() (*sage.rings.rational.Rational method*), 174
MutableMultiModularBasis (*class in sage.arith.multi_modular*), 76

N

nbits() (*sage.rings.integer.Integer method*), 42
ndigits() (*sage.rings.integer.Integer method*), 42
next_prime() (*in module sage.arith.misc*), 121
next_prime() (*sage.arith.multi_modular.MutableMultiModularBasis method*), 76
next_prime() (*sage.rings.integer.Integer method*), 42
next_prime_power() (*in module sage.arith.misc*), 122
next_prime_power() (*sage.rings.integer.Integer method*), 43
next_probable_prime() (*in module sage.arith.misc*), 123
next_probable_prime() (*sage.rings.integer.Integer method*), 43
ngens() (*sage.rings.integer_ring.IntegerRing_class method*), 7
ngens() (*sage.rings.rational_field.RationalField method*), 151
norm() (*sage.rings.rational.Rational method*), 175
nth_prime() (*in module sage.arith.misc*), 123
nth_root() (*sage.rings.integer.Integer method*), 44
nth_root() (*sage.rings.rational.Rational method*), 175
number_field() (*sage.rings.rational_field.RationalField method*), 151
number_of_divisors() (*in module sage.arith.misc*), 123
numer() (*sage.rings.rational.Rational method*), 176
numerator() (*sage.rings.integer.Integer method*), 45
numerator() (*sage.rings.rational.Rational method*), 176

O

oct() (*sage.rings.integer.Integer method*), 45
odd_part() (*in module sage.arith.misc*), 124
odd_part() (*sage.rings.integer.Integer method*), 46
ord() (*sage.rings.integer.Integer method*), 46
ord() (*sage.rings.rational.Rational method*), 176

order() (*sage.rings.integer_ring.IntegerRing_class method*), 7
order() (*sage.rings.rational_field.RationalField method*), 151
ordinal_str() (*sage.rings.integer.Integer method*), 47

P

p_primary_part() (*sage.rings.integer.Integer method*), 47
parameter() (*sage.rings.integer_ring.IntegerRing_class method*), 7
partial_product() (*sage.arith.multi_modular.MultiModularBasis_base method*), 75
perfect_power() (*sage.rings.integer.Integer method*), 48
period() (*sage.rings.rational.Rational method*), 177
places() (*sage.rings.rational_field.RationalField method*), 151
plot() (*sage.arith.misc.Euler_Phi method*), 82
plot() (*sage.arith.misc.Moebius method*), 84
plot() (*sage.arith.misc.Sigma method*), 85
polynomial() (*sage.rings.rational_field.RationalField method*), 152
popcount() (*sage.rings.integer.Integer method*), 48
power_basis() (*sage.rings.rational_field.RationalField method*), 152
power_mod() (*in module sage.arith.misc*), 124
powermod() (*sage.rings.integer.Integer method*), 49
precomputation_list() (*sage.arith.multi_modular.MultiModularBasis_base method*), 75
previous_prime() (*in module sage.arith.misc*), 125
previous_prime() (*sage.rings.integer.Integer method*), 49
previous_prime_power() (*in module sage.arith.misc*), 125
previous_prime_power() (*sage.rings.integer.Integer method*), 50
prime_divisors() (*in module sage.arith.misc*), 126
prime_divisors() (*sage.rings.integer.Integer method*), 50
prime_factors() (*in module sage.arith.misc*), 127
prime_factors() (*sage.rings.integer.Integer method*), 51
prime_powers() (*in module sage.arith.misc*), 128
prime_range() (*in module sage.rings.fast_arith*), 68
prime_to_m_part() (*in module sage.arith.misc*), 128
prime_to_m_part() (*sage.rings.integer.Integer method*), 51
prime_to_S_part() (*sage.rings.rational.Rational method*), 177
primes() (*in module sage.arith.misc*), 129
primes_first_n() (*in module sage.arith.misc*), 130
primes_of_bounded_norm_iter() (*sage.rings.rational_field.RationalField method*), 152
primitive_root() (*in module sage.arith.misc*), 130
prod() (*sage.arith.multi_modular.MultiModularBasis_base method*), 75
Python Enhancement Proposals
    PEP 3127, 15

Q

Q (*in module sage.rings.rational_field*), 145
QQ (*class in sage.rings.rational*), 159
QQ (*in module sage.rings.rational_field*), 145
quadratic_defect() (*sage.rings.rational_field.RationalField method*), 152
quadratic_residues() (*in module sage.arith.misc*), 131
quo_rem() (*sage.rings.integer.Integer method*), 51
quotient() (*sage.rings.integer_ring.IntegerRing_class method*), 7

R

radical() (*in module sage.arith.misc*), 131
random_element() (*sage.rings.integer_ring.IntegerRing_class method*), 7
random_element() (*sage.rings.rational_field.RationalField method*), 153
random_prime() (*in module sage.arith.misc*), 132
range() (*sage.arith.misc.Moebius method*), 84
range() (*sage.rings.integer_ring.IntegerRing_class method*), 9
range_by_height() (*sage.rings.rational_field.RationalField method*), 154
Rational (*class in sage.rings.rational*), 159
rational_power_parts() (*in module sage.rings.rational*), 185
rational_recon_int() (*sage.rings.fast_arith.arith_int method*), 68
rational_recon_longlong() (*sage.rings.fast_arith.arith_llong method*), 68
rational_reconstruction() (*in module sage.arith.misc*), 133
rational_reconstruction() (*sage.rings.integer.Integer method*), 52
RationalField (*class in sage.rings.rational_field*), 145
real() (*sage.rings.integer.Integer method*), 52
real() (*sage.rings.rational.Rational method*), 178
relative_discriminant() (*sage.rings.rational_field.RationalField method*), 154
relative_norm() (*sage.rings.rational.Rational method*), 178

replace_prime() (*sage.arith.multi_modular.MutableMultiModularBasis method*), 76
residue_field() (*sage.rings.integer_ring.IntegerRing_class method*), 10
residue_field() (*sage.rings.rational_field.RationalField method*), 154
rising_factorial() (*in module sage.arith.misc*), 134
round() (*sage.rings.integer.Integer method*), 52
round() (*sage.rings.rational.Rational method*), 178

S

sage.arith.functions
    module, 71
sage.arith.misc
    module, 77
sage.arith.multi_modular
    module, 74
sage.arith.power
    module, 73
sage.rings.bernmm
    module, 61
sage.rings.bernoulli_mod_p
    module, 62
sage.rings.factorint
    module, 65
sage.rings.fast_arith
    module, 68
sage.rings.integer
    module, 13
sage.rings.integer_ring
    module, 1
sage.rings.rational
    module, 159
sage.rings.rational_field
    module, 145
sage.rings.sum_of_squares
    module, 69
section() (*sage.rings.rational.Q_to_Z method*), 159
section() (*sage.rings.rational.Z_to_Q method*), 183
selmer_generators() (*sage.rings.rational_field.RationalField method*), 155
selmer_group() (*sage.rings.rational_field.RationalField method*), 155
selmer_group_iterator() (*sage.rings.rational_field.RationalField method*), 155
selmer_space() (*sage.rings.rational_field.RationalField method*), 156
Sigma (*class in sage.arith.misc*), 85
sign() (*sage.rings.integer.Integer method*), 52
sign() (*sage.rings.rational.Rational method*), 179
signature() (*sage.rings.rational_field.RationalField method*), 157
simplify() (*sage.rings.rational_field.RationalField method*), 157
sort_complex_numbers_for_display() (*in module sage.arith.misc*), 135
sqrt() (*sage.rings.integer.Integer method*), 53
sqrt() (*sage.rings.rational.Rational method*), 179
sqrtrem() (*sage.rings.integer.Integer method*), 54
squarefree_divisors() (*in module sage.arith.misc*), 136
squarefree_part() (*sage.rings.integer.Integer method*), 54
squarefree_part() (*sage.rings.rational.Rational method*), 180
str() (*sage.rings.integer.Integer method*), 55
str() (*sage.rings.rational.Rational method*), 181
subfactorial() (*in module sage.arith.misc*), 136
sum_of_k_squares() (*in module sage.arith.misc*), 137
support() (*sage.rings.integer.Integer method*), 55
support() (*sage.rings.rational.Rational method*), 181

T

test_bit() (*sage.rings.integer.Integer method*), 55
three_squares() (*in module sage.arith.misc*), 138
three_squares_pyx() (*in module sage.rings.sum_of_squares*), 70
trace() (*sage.rings.rational.Rational method*), 181
trailing_zero_bits() (*sage.rings.integer.Integer method*), 56
trial_division() (*in module sage.arith.misc*), 139
trial_division() (*sage.rings.integer.Integer method*), 56
trunc() (*sage.rings.integer.Integer method*), 57
trunc() (*sage.rings.rational.Rational method*), 182
two_squares() (*in module sage.arith.misc*), 139
two_squares_pyx() (*in module sage.rings.sum_of_squares*), 71

V

val_unit() (*sage.rings.integer.Integer method*), 57
val_unit() (*sage.rings.rational.Rational method*), 182
valuation() (*in module sage.arith.misc*), 140
valuation() (*sage.rings.integer.Integer method*), 58
valuation() (*sage.rings.integer_ring.IntegerRing_class method*), 11
valuation() (*sage.rings.rational.Rational method*), 182
valuation() (*sage.rings.rational_field.RationalField method*), 158
verify_bernoulli_mod_p() (*in module sage.rings.bernoulli_mod_p*), 64

X

XGCD() (*in module sage.arith.misc*), 86
xgcd() (*in module sage.arith.misc*), 141

xgcd() (*sage.rings.integer.Integer method*), 58
xgcd_int() (*sage.rings.fast_arith.arith_int method*), 68
xkcd() (*in module sage.arith.misc*), 142
xlcm() (*in module sage.arith.misc*), 143

Z

Z_to_Q (*class in sage.rings.rational*), 183
zeta() (*sage.rings.integer_ring.IntegerRing_class method*), 11
zeta() (*sage.rings.rational_field.RationalField method*), 158