Standard Commutative Rings

Release 10.4

The Sage Development Team

Jul 23, 2024
| CONTENTS  |    |
|----------|--|---|
| 1 Integers |    | 1 |
| 2 Rationals |    | 249 |
| 3 Indices and Tables |    | 317 |
| Python Module Index |    | 319 |
| Index |    | 321 |
1.1 Ring $\mathbb{Z}$ of Integers

The `IntegerRing_class` represents the ring $\mathbb{Z}$ 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('94803849083985934859834583945394')
94803849083985934859834583945394
```

(continues on next page)
sage.rings.integer_ring.IntegerRing()

Return the integer ring.

EXAMPLES:

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

```python
>>> from sage.all import *
>>> IntegerRing()
Integer Ring
>>> ZZ==IntegerRing()
True
```

class sage.rings.integer_ring.IntegerRing_class

Bases: CommutativeRing

The ring of integers.

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

```python
sage: Z = IntegerRing(); Z
Integer Ring
sage: Z.characteristic()
0
sage: Z.is_field()
False
sage: Z.category()
Join of Category of Dedekind domains
    and Category of euclidean domains
    and Category of noetherian rings
    and Category of infinite enumerated sets
    and Category of metric spaces
sage: Z(2^(2^5) + 1)
4294967297
```

```python
>>> from sage.all import *
```
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
We next illustrate basic arithmetic in $\mathbb{Z}$:

```python
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
```

(continues on next page)
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'>
```

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
1234/19
```
Integers can be coerced:

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

We can create integers from several types of objects:

```python
sage: Z(17/1)
17
sage: Z(Mod(19,23))
19
sage: Z(2 + 3*5 + O(5^3))
# needs sage.rings.padics
17
```

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

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

```python
>>> from sage.all import *
>>> a + Integer(17)
1251
>>> a * Integer(374)
461516
>>> Integer(374) * a
461516
>>> a/Integer(19)
1234/19
>>> Integer(0) + Z(-Integer(64))
-64
```
The \texttt{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):

\begin{verbatim}
    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]
\end{verbatim}

The \texttt{str} method returns a string of the digits, using letters \texttt{a} to \texttt{z} to represent digits 10..36:

\begin{verbatim}
    sage: Z(928270).str(base=32)
    'sage'
\end{verbatim}

Note that \texttt{str} only works with bases 2 through 36.

\textbf{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.

\textbf{EXAMPLES}:

\begin{verbatim}
    sage: ZZ.absolute_degree()
    1
\end{verbatim}
**characteristic()**

Return the characteristic of the integers, which is 0.

**EXAMPLES:**

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

```python
>>> from sage.all import *
>>> ZZ.characteristic()
0
```

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

Return the metric completion of the integers at the prime \(p\).

**INPUT:**

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

**OUTPUT:**

- The completion of \(\mathbb{Z}\) at \(p\).

**EXAMPLES:**

```
sage: ZZ.completion(infinity, 53)
Integer Ring
sage: ZZ.completion(5, 15, {'print_mode': 'bars'})  # needs sage.rings.padics
5-adic Ring with capped relative precision 15
```

```python
>>> from sage.all import *
>>> ZZ.completion(infinity, Integer(53))
Integer Ring
>>> ZZ.completion(Integer(5), Integer(15), {'print_mode': 'bars'})  # needs sage.rings.padics
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
```

```python
>>> from sage.all import *
>>> 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
x = polygen(ZZ, 'x')
ZZ.extension(x^2 - 5, 'a')  # needs sage.rings.number_field
```

Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5

```sage
ZZ.extension([x^2 + 1, x^2 + 2], 'a,b')  # needs sage.rings.number_field
```

Relative Order generated by [-b*a - 1, -3*a + 2*b] in Number Field in a with defining polynomial x^2 + 1 over its base field

```python
from sage.all import *
>>> x = polygen(ZZ, 'x')
>>> ZZ.extension(x^Integer(2) - Integer(5), a)  # needs sage.rings.number_field
```

Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5

```python
>>> ZZ.extension([x^Integer(2) + Integer(1), x^Integer(2) + Integer(2)], a, b)  # needs sage.rings.number_field
```

Relative Order generated by [-b*a - 1, -3*a + 2*b] 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
sage: ZZ.fraction_field()
Rational Field
sage: ZZ.fraction_field() == QQ
True
```

```python
>>> from sage.all import *
>>> ZZ.fraction_field()  # needs sage.rings.number_field
```

Rational Field

```python
>>> ZZ.fraction_field() == QQ
True
```

from_bytes(input_bytes, byteorder='big', is_signed=False)

Return the integer represented by the given array of bytes.

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

INPUT:
• input_bytes – a bytes-like object or iterable producing bytes

1.1. Ring Z of Integers
• `byteorder`—str (default: "big"); determines the byte order of `input_bytes`; can only be "big" or "little"

• `is_signed`—boolean (default: `False`); determines whether to use two’s compliment to represent the integer

EXAMPLES:

```python
sage: ZZ.from_bytes(b'\x00\x10', byteorder='big')
16
sage: ZZ.from_bytes(b'\x00\x10', byteorder='little')
4096
sage: ZZ.from_bytes(b'\xfc\x00', byteorder='big', is_signed=True)
-1024
sage: ZZ.from_bytes(b'\xfc\x00', byteorder='big', is_signed=False)
65536
sage: ZZ.from_bytes([255, 0, 0], byteorder='big')
16711680
sage: type(_)
<class 'sage.rings.integer.Integer'>
```

```python
>>> from sage.all import *

sage: ZZ.from_bytes(b'\x00\x10', byteorder='big')
16
sage: ZZ.from_bytes(b'\x00\x10', byteorder='little')
4096
sage: ZZ.from_bytes(b'\xfc\x00', byteorder='big', is_signed=True)
-1024
sage: ZZ.from_bytes(b'\xfc\x00', byteorder='big', is_signed=False)
65536
sage: ZZ.from_bytes([Integer(255), Integer(0), Integer(0)], byteorder='big')
16711680
sage: type(_)
<class 'sage.rings.integer.Integer'>
```

### 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:**

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

```python
>>> from sage.all import *

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:

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

```python
>>> from sage.all import *

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

**is_field** *(proof=True)*

Return False since the integers are not a field.

EXAMPLES:

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

```python
>>> from sage.all import *

>>> ZZ.is_field()
False
```

**is_integrally_closed()**

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

**Note:** This should rather be inherited from the category of DedekindDomains.

EXAMPLES:

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

```python
>>> from sage.all import *

>>> ZZ.is_integrally_closed()
True
```

**krull_dimension()**

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

**Note:** This should rather be inherited from the category of DedekindDomains.

EXAMPLES:

```python
sage: ZZ.krull_dimension()
1
```
ngens ()
Return the number of additive generators of the ring, which is 1.

EXAMPLES:

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

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

EXAMPLES:

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

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

EXAMPLES:

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

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

EXAMPLES:

```python
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)
```

(continues on next page)
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 $2^y - 1$.

  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:

```python
sage: ZZ.random_element().parent() is ZZ
True
```

```python
>>> from sage.all import *
>>> ZZ.random_element().parent() is ZZ
True
```

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

```python
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

sage: def prob(x):
    return 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")
```

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

>>> def prob(x):
...     return Integer(1)/Integer(5)

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

Here we use the distribution '1/n':

```python
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")
```
>>> from sage.all import *
>>> def prob(n):
...     if n == Integer(0):
...         return Integer(1)/Integer(5)
...     return Integer(2)/(Integer(5)*abs(n)*(abs(n) + Integer(1)))
>>> dic = defaultdict(Integer)
>>> counter = RealNumber('0.0')
>>> add_samples(distribution="1/n")
>>> while any(abs(dic[i]/counter - prob(i)) > RealNumber('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: def prob(x):
......:     return 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: def prob(x):
......:     return 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

>>> from sage.all import *
>>> -Integer(10) <= ZZ.random_element(-Integer(10), Integer(10)) < Integer(10)
True
>>> def prob(x):
...     return Integer(1)/Integer(20)
>>> dic = defaultdict(Integer)
>>> counter = RealNumber('0.0')
>>> add_samples(-Integer(10), Integer(10))
>>> while any(abs(dic[i]/counter - prob(i)) > RealNumber('0.01') for i in dic):
...     add_samples(-Integer(10), Integer(10))
>>> Integer(0) <= ZZ.random_element(Integer(5)) < Integer(5)
True
>>> def prob(x):
...     return Integer(1)/Integer(5)
>>> dic = defaultdict(Integer)
>>> counter = RealNumber('0.0')
>>> add_samples(Integer(5))
>>> while any(abs(dic[i]/counter - prob(i)) > RealNumber('0.01') for i in dic):
.....:     pass

(continues on next page)
Notice that the right endpoint is not included:

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

We return a sample from a discrete Gaussian distribution:

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

```python
>>> from sage.all import *
>>> ZZ.random_element(RealNumber('11.0'), distribution="gaussian").parent()  # needs sage.modules
ZZ
```

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

Optimized range function for Sage integers.

AUTHORS:

- Robert Bradshaw (2007-09-20)

EXAMPLES:

```python
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]
```

```python
>>> from sage.all import *
>>> ZZ.range(Integer(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> ZZ.range(-Integer(5), Integer(5))
```
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 Issue #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. \( \mathbb{Z}/p\mathbb{Z} \).

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

>>> from sage.all import *
>>> ZZ.residue_field(ZZ.ideal(Integer(97)))
Residue field of Integers modulo 97

valuation \((p)\)
Return the discrete valuation with uniformizer \(p\).

EXAMPLES:

sage: v = ZZ.valuation(3); v
#...

(continues on next page)
3-adic valuation

```
sage: v(3)  # needs sage.rings.padics
1
```

```
>>> from sage.all import *

>>> v = ZZ.valuation(Integer(3)); v
# needs sage.rings.padics
3-adic valuation

>>> v(Integer(3))
# needs sage.rings.padics
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 \( \mathbb{Z} \)

**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
sage: ZZ.zeta(0)
Traceback (most recent call last):
...
ValueError: n must be positive in zeta()
```

```
>>> from sage.all import *

>>> ZZ.zeta()
-1

>>> ZZ.zeta(Integer(1))
1

>>> ZZ.zeta(Integer(3))
Traceback (most recent call last):
...
ValueError: no nth root of unity in integer ring

>>> ZZ.zeta(Integer(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 $E[i] = 1 \pmod{X[i]}$ and $E[i] = 0 \pmod{X[j]}$ 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:

```python
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)
```

```python
>>> from sage.all import *

>>> X = [Integer(11),Integer(20),Integer(31),Integer(51)]
>>> E = crt_basis([Integer(11),Integer(20),Integer(31),Integer(51)])
>>> E[0]%X[0], E[1]%X[0], E[2]%X[0], E[3]%X[0]
(1, 0, 0, 0)
>>> E[0]%X[1], E[1]%X[1], E[2]%X[1], E[3]%X[1]
(0, 1, 0, 0)
>>> E[0]%X[2], E[1]%X[2], E[2]%X[2], E[3]%X[2]
(0, 0, 1, 0)
>>> 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 \( \mathbb{Z} \) 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:

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

```python
>>> from sage.all import *

>>> a = Integer(Integer(3)); b = Integer(Integer(4))

>>> a + b == Integer(7)
True
```

Add an integer and a real number:

```python
sage: a + 4.0

# needs sage.rings.real_mpfr
7.00000000000000
```

```python
>>> from sage.all import *

>>> a + RealNumber('4.0')

# needs sage.rings.real_mpfr
7.00000000000000
```

Add an integer and a rational number:

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

17/5
```

```python
>>> from sage.all import *

>>> a + Rational(Integer(2))/Integer(5)

17/5
```

Add an integer and a complex number:

```python
sage: # needs sage.rings.real_mpfr
sage: b = ComplexField().0 + 1.5
sage: loads((a + b).dumps()) == a + b
True
```

```python
sage: z = 32
sage: -z

-32
```

```python
sage: z = 0; -z

0
```

```python
sage: z = -0; -z

0
```

(continues on next page)
\textbf{COERCIONS:}

Return version of this integer in the multi-precision floating real field $\mathbb{R}$:
sage: n = 9390823
sage: RR = RealField(200)  # needs sage.rings.real_mpfr
sage: RR(n)  # needs sage.rings.real_mpfr
9.3908230000000000000000000000000000000000000000000000000000e6

>>> from sage.all import *
>>> n = Integer(9390823)
>>> RR = RealField(Integer(200))  # needs sage.rings.real_mpfr
>>> RR(n)  # needs sage.rings.real_mpfr
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 Issue #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 (Issue #25983)

sage.rings.integer.GCD_list(v)
Return the greatest common divisor of a list of integers.

1.2. Elements of the ring \( \mathbb{Z} \) of integers 23
• $v$ – 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:**

```python
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'>
```

```python
>>> from sage.all import *

>>> from sage.rings.integer import GCD_list

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

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

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

```python
>>> from sage.all import *

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

The inputs are converted to Sage integers.

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

```python
>>> from sage.all import *

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

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

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

```python
>>> from sage.all import *

>>> GCD_list([])
0
```

**class** `sage.rings.integer.Integer`  
**Bases:** `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.
The constructor of \texttt{Integer} interprets strings that begin with \texttt{0o} as octal numbers, strings that begin with \texttt{0x} as hexadecimal numbers and strings that begin with \texttt{0b} as binary numbers.

The class \texttt{Integer} is implemented in Cython, as a wrapper of the GMP \texttt{mpz} integer type.

EXAMPLES:

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

```
>>> from sage.all import *

>>><br>
>>><br>
```

Sage integers support PEP 3127 literals:

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

```
>>> from sage.all import *

>>><br>
>>><br>
```

Conversion from PARI:

```
sage: Integer(pari('-1038010437159300804879946356441519384'))  # ...
-1038010437159300804879946356441519384
sage: Integer(pari('Pol([-3])'))  # ...
-3
```

```
>>> from sage.all import *

>>><br>
>>><br>
```

Conversion from gmpy2:

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

1.2. Elements of the ring \( \mathbb{Z} \) of integers
__pow__ (left, right, modulus)

Return \( (\text{left} \ ^ \ \text{right}) \ % \ \text{modulus} \).

EXAMPLES:

```
sage: 2^-6
1/64
sage: 2^6
64
sage: 2^0
1
sage: 2^-0
1
sage: (-1)^(1/3) # needs sage.symbolic
(-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)
```

(continues on next page)
We raise $2$ to various interesting exponents:

```
sage: 2^x  # symbolic x
    ...needs sage.symbolic
2^x
sage: 2^1.5  # real number
    ...needs sage.rings.real_mpfr
2.82842712474619
sage: 2^float(1.5)  # python float abs tol 3e-16
2.8284271247461903
sage: 2^I  # complex number
    ...needs sage.symbolic
2^I
```

```
>>> r = int(Integer(2)) ** Integer(10); r; type(r)
1024  
>>> r = int(Integer(3)) ** -Integer(3); r; type(r)
0.037037037037037035  
>>> r = float(RealNumber('2.5')) ** Integer(10); r; type(r)
9536.7431640625
```

```
>>> from sage.all import *
```
A symbolic sum:

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

```python
>>> from sage.all import *
>>> # needs sage.symbolic
>>> x, y, z = var('x,y,z')
>>> Integer(2)**(x + y + z)
>>> Integer(2)**(Integer(1)/Integer(2))
>>> sqrt(2)
>>> Integer(2)**(-Integer(1)/Integer(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
```

```python
>>> from sage.all import *
>>> ZZ(Integer(0)).additive_order()
1
>>> ZZ(Integer(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)
```

```python
>>> from sage.all import *
>>> x = -Integer(12)
>>> 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 $\text{positive\_shift}$.

**INPUT:**

- \text{base} – integer (default: 10); when \text{base} is 2, only the nonnegative or the nonpositive integers can be represented by $\text{balanced\_digits}$. Thus we say \text{base} must be greater than 2.

- \text{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:**

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

```python
>>> from sage.all import *
>>> Integer(8).balanced_digits(Integer(3))
[-1, 0, 1]
>>> (-Integer(15)).balanced_digits(Integer(5))
[0, 2, -1]
>>> Integer(17).balanced_digits(Integer(6))
[-1, 3]
>>> Integer(17).balanced_digits(Integer(6), positive_shift=False)
[-1, -3, 1]
>>> (-Integer(46)).balanced_digits()
[4, 5, -1]
>>> (-Integer(46)).balanced_digits(positive_shift=False)
[4, -5]
>>> (-Integer(23)).balanced_digits(Integer(12))
[1, -2]
>>> (-Integer(23)).balanced_digits(Integer(12), positive_shift=False)
```

(continues on next page)
See also:

digits

binary()
Return the binary digits of self as a string.

EXAMPLES:

```python
sage: print(Integer(15).binary())
1111
sage: print(Integer(16).binary())
10000
sage: print(Integer(16938402384092843092843098243).binary())
1101101011101100011111000111001001010011101000110101000111111000101000000000101111000010000011
```

```python
>>> from sage.all import *
>>> print(Integer(Integer(15)).binary())
1111
>>> print(Integer(Integer(16)).binary())
10000
>>> print(Integer(Integer(16938402384092843092843098243)).binary())
1101101011101100011111000111001001010011101000110101000111111000101000000000101111000010000011
```

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:

```python
sage: 10.binomial(2)
45
sage: 10.binomial(2, algorithm='pari')
# needs sage.libs pari
45
sage: 10.binomial(-2)
0
sage: (-2).binomial(3)
```

(continues on next page)
The argument $m$ or $(self - m)$ must fit into an unsigned long:

```
sage: (2**256).binomial(2**256)
1
sage: (2**256).binomial(2**256 - 1)
1157920892373161954235709850088687907853269984665640564039457584007913129639936
sage: (2**256).binomial(2**128)
Traceback (most recent call last):
... OverflowError: m must fit in an unsigned long
```

```
>>> from sage.all import *
>>> (Integer(2)**Integer(256)).binomial(Integer(2)**Integer(256))
1
>>> (Integer(2)**Integer(256)).binomial(Integer(2)**Integer(256) - Integer(1))
1157920892373161954235709850088687907853269984665640564039457584007913129639936
>>> (Integer(2)**Integer(256)).binomial(Integer(2)**Integer(128))
Traceback (most recent call last):
... OverflowError: m must fit in an unsigned long
```

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

**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

\[ x == \sum (b \times 2^e \text{ for } e, b \text{ in enumerate}(x\text{.bits}())) \]

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

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
```
>>> from sage.all import *
>>> n = Integer(6)
>>> 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. However, the results are correct for discriminants \( D \) with \(|D| \leq 2 \cdot 10^{10}\).

OUTPUT:

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

Note: For positive \( D \), 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() # needs sage.libs.pari
1
sage: (-104).class_number() # needs sage.libs.pari
6
sage: [((4*n + 1), (4*n + 1).class_number()) for n in [21..29]] # needs sage.libs.pari
[(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:

```python
sage: n = 205
sage: n.conjugate()
205
```

```python
given from sage.all import *
given n = Integer(205)
given n.conjugate()
given 205
```

coprime_integers(m)

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

EXAMPLES:

```python
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]
```

```python
given from sage.all import *
given n = Integer(8)
given n.coprime_integers(Integer(8))
[1, 3, 5, 7]
given n.coprime_integers(Integer(11))
[1, 3, 5, 7, 9]
given n = Integer(5); n.coprime_integers(Integer(10))
[1, 2, 3, 4, 6, 7, 8, 9]
given n.coprime_integers(Integer(5))
[1, 2, 3, 4]
given n = Integer(99); n.coprime_integers(Integer(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
sage: n = 17
sage: m = n.crt(5, 23, 11); m
247
sage: m%23
17
sage: m%11
5
```

```python
>>> from sage.all import *

```

```python
>>> n = Integer(17)
>>> m = n.crt(Integer(5), Integer(23), Integer(11)); m
247
>>> m=Integer(23)
17
>>> m=Integer(11)
5
```

denominator ()

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

EXAMPLES:

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

```python
>>> from sage.all import *

```

```python
>>> x = Integer(5)
>>> x.denominator()
1
>>> x = Integer(0)
>>> x.denominator()
1
```

1.2. Elements of the ring \( \mathbb{Z} \) of integers
\textbf{digits} (\texttt{base=10, digits=None, padto=0})

Return a list of digits for \texttt{self} in the given base in little endian order.

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

INPUT:

- \texttt{base} – integer (default: 10)
- \texttt{digits} – optional indexable object as source for the digits
- \texttt{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 \texttt{digits(2)}, you can use \texttt{bits()}.

Also see \texttt{ndigits()}.

\textbf{EXAMPLES}:

\begin{verbatim}
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]
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]

>>> from sage.all import *

>>> Integer(17).digits()
[7, 1]
>>> Integer(5).digits(base=Integer(2), digits=['zero','one'])
['one', 'zero', 'one']
>>> Integer(5).digits(Integer(3))
[2, 1]
>>> Integer(0).digits(base=Integer(10))  # 0 has 0 digits
[]
\end{verbatim}
We support large bases.

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

```python
>>> from sage.all import *
>>> n=Integer(2)**Integer(6000)
>>> n.digits(Integer(2)**Integer(3000))
[0, 0, 1]
```

```python
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
```

```python
>>> from sage.all import *
>>> base=Integer(3); n=Integer(25)
>>> l=n.digits(base)
>>> # the next relationship should hold for all n,base
>>> sum(base**i*l[i] for i in range(len(l)))==n
True
>>> base=Integer(3); n=-Integer(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 \texttt{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
```

```
>>> from sage.all import *
>>> x = Integer(123); ZZ(x.digits(), Integer(10))
123
>>> x == ZZ(x.digits(Integer(6)), Integer(6))
True
>>> x == ZZ(x.digits(Integer(25)), Integer(25))
True
```

Using \texttt{sum()} and \texttt{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
```

```
>>> from sage.all import *
>>> base = Integer(4)
>>> 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]
```

```
>>> from sage.all import *
>>> digits = list(range(Integer(15)))
>>> l = [ZZ(i).digits(Integer(15),digits) for i in range(Integer(100))]  
>>> l[Integer(16)]
[1, 1]
```

This function is comparable to \texttt{str()} for speed.

```
sage: n=3^100000
sage: n.digits(base=10)[-1]  # slightly slower than str
˓→needs sage.rings.real_interval_field
1
sage: n=10^100000
sage: n.digits(base=10)[-1]  # slightly faster than str
˓→needs sage.rings.real_interval_field
1
```
AUTHORS:

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

divide_knowing_divisible_by(right)

Return the integer \( \frac{\text{self}}{\text{right}} \) when \( \text{self} \) is divisible by \( \text{right} \).

If \( \text{self} \) is not divisible by \( \text{right} \), the return value is undefined, and may not even be close to \( \frac{\text{self}}{\text{right}} \) for multi-word integers.

EXAMPLES:

```python
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.

```python
sage: a = 2^70; a
1180591620717411303424
sage: a // 11  # floor divide
107326510974310118493
sage: a.divide_knowing_divisible_by(11)  # way off and possibly random
43215361478743422388970455040
```

1.2. Elements of the ring \( \mathbb{Z} \) of integers
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
```

```python
>>> from sage.all import *
>>> Z = IntegerRing()
>>> Z(Integer(5)).divides(Z(Integer(10)))
True
>>> Z(Integer(0)).divides(Z(Integer(5)))
False
>>> Z(Integer(10)).divides(Z(Integer(5)))
False
```

divisors \( \text{(method=None)} \)  
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()  
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()  
#-- needs sage.libs.pari
sage: len(v)  
#-- needs sage.libs.pari
172800
sage: prod(e + 1 for p, e in factor(a))  
172800
sage: all(t.divides(a) for t in v)  
#-- needs sage.libs.pari
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.

\textbf{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.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: ZZ(1).euclidean_degree()
sage: ZZ(Integer(1)).euclidean_degree()
1
\end{verbatim}

\textbf{exact\_log}(m)  
Return the largest integer \(k\) such that \(m^k \leq \text{self}\), 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.

\textbf{INPUT:}

\begin{itemize}
  \item \(m\) – integer \(\geq 2\)
\end{itemize}

\textbf{AUTHORS:}

\begin{itemize}
  \item David Harvey (2006-09-15)
  \item Joel B. Mohler (2009-04-08) – rewrote this to handle small cases and/or easy cases up to 100x faster.
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: Integer(125).exact_log(5)
sage: Integer(124).exact_log(5)
sage: Integer(126).exact_log(5)
sage: Integer(3).exact_log(5)
sage: Integer(1).exact_log(5)
sage: Integer(178^1700).exact_log(178)
sage: Integer(178^1700-1).exact_log(178)
sage: Integer(178^1700+1).exact_log(178)
3
2
3
0
0
1700
1699
1700
1700
1700
1700
1699
\end{verbatim}

\begin{verbatim}
sage: # we need to exercise the large base code path too
sage: Integer(178^1700-1).exact_log(1780)  # needs sage.rings.real_interval_field
1699
\end{verbatim}

\begin{verbatim}
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 = 2983579823750185701375109835; m = 32
\end{verbatim}

(continues on next page)
sage: n.exact_log(m)
18
sage: # The next is a favorite of mine. The log2 approximate is exact and
→immediately provable.
sage: n = 9015371057912709517902579010793251709257901270941709247901209742124
sage: m = 213509721309572
sage: n.exact_log(m)
4

>>> from sage.all import *
>>> Integer(Integer(125)).exact_log(Integer(5))
3
>>> Integer(Integer(124)).exact_log(Integer(5))
2
>>> Integer(Integer(126)).exact_log(Integer(5))
3
>>> Integer(Integer(3)).exact_log(Integer(5))
0
>>> Integer(Integer(1)).exact_log(Integer(5))
0
>>> Integer(Integer(178)**Integer(1700)).exact_log(Integer(178))
1700
>>> Integer(Integer(178)**Integer(1700)-Integer(1)).exact_log(Integer(178))
1699
>>> Integer(Integer(178)**Integer(1700)+Integer(1)).exact_log(Integer(178))
1700
>>> # we need to exercise the large base code path too
>>> Integer(Integer(1780)**Integer(1700)-Integer(1)).exact_log(Integer(1780))
1699
>>> # The following are very very fast.
>>> # Note that for base m a perfect power of 2, we get the exact log by
→counting bits.
>>> n = Integer(2983579823750185701375109835); m = Integer(32)
>>> n.exact_log(m)
18
>>> # The next is a favorite of mine. The log2 approximate is exact and
→immediately provable.
>>> n =
→Integer(9015371057912709517902579010793251709257901270941709247901209742124)
>>> m = Integer(213509721309572)
>>> n.exact_log(m)
4

sage: # needs sage.rings.real_mpfr
sage: x = 3^100000
sage: RR(log(RR(x), 3))
100000.000000000
sage: RR(log(RR(x + 100000), 3))
100000.000000000

>>> from sage.all import *
>>> # needs sage.rings.real_mpfr
>>> x = Integer(3)**Integer(100000)
>>> RR(log(RR(x), Integer(3)))

1.2. Elements of the ring \( Z \) of integers

43
100000.000000000
>>> RR(log(RR(x + Integer(100000)), Integer(3)))
100000.000000000

```
sage: # needs sage.rings.real_mpfr
sage: x.exact_log(3)
100000
sage: (x + 1).exact_log(3)
100000
sage: (x - 1).exact_log(3)
99999
```

```python
>>> from sage.all import *
```

```python
# needs sage.rings.real_mpfr
```

```python
x.exact_log(Integer(3))
100000
```

```python
(x + Integer(1)).exact_log(Integer(3))
100000
```

```python
(x - Integer(1)).exact_log(Integer(3))
99999
```

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

```python
>>> from sage.all import *
```

```python
# needs sage.rings.real_mpfr
```

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

```
exp(prec=None)
```

Return 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()  # needs sage.symbolic
e^8
```

```
sage: Integer(8).exp(prec=100)  # needs sage.symbolic
2980.9579870417282747435920995
```

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

```
sage: y = Integer(145^145)
sage: y.exp() # needs sage.symbolic
→ e^25024207011349079210459585279553675697932183658421565260323592409432707306554163224876110065674577230503793369785727108337766011928747055351280379806937944746847277089168867282654496776717056860661614337004721164703369140625

sage: y.exp(prec=53) # default RealField precision #
→ +infinity
```

```
>>> from sage.all import *
>>> y = Integer(145)**Integer(145)
>>> y.exp() # needs sage.symbolic
→ e^25024207011349079210459585279553675697932183658421565260323592409432707306554163224876110065674577230503793369785727108337766011928747055351280379806937944746847277089168867282654496776717056860661614337004721164703369140625

>>> y.exp(prec=Integer(53)) # default RealField precision
→ +infinity
```

\textbf{factor} \texttt{(algorithm='pari', proof=None, limit=None, int_=False, verbose=0)}

Return the prime factorization of this integer as a formal \texttt{Factorization} object.

INPUT:

- \texttt{algorithm} – \texttt{string}
  
  - 'pari' – (default) use the PARI library
  
  - 'flint' – use the FLINT library
  
  - 'kash' – use the KASH computer algebra system (requires \texttt{kash})
  
  - 'magma' – use the MAGMA computer algebra system (requires an installation of \texttt{MAGMA})
  
  - 'qsieve' – use Bill Hart’s quadratic sieve code; WARNING: this may not work as expected, see \texttt{qsieve} for more information
  
  - 'ecm' – use ECM-GMP, an implementation of Hendrik Lenstra’s elliptic curve method.

- \texttt{proof} – \texttt{bool} (default: \texttt{True}) whether or not to prove primality of each factor (only applicable for 'pari' and 'ecm').

- \texttt{limit} – \texttt{int} or \texttt{None} (default: \texttt{None}) if limit is given it must fit in a \texttt{signed int}, and the factorization is done using trial division and primes up to limit.

OUTPUT:

- a \texttt{Factorization} object containing the prime factors and their multiplicities

\textit{1.2. Elements of the ring \( \mathbb{Z} \) of integers}
EXAMPLES:

```python
sage: n = 2^100 - 1; n.factor()  # needs sage.libs.pari
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):

```python
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:

```python
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:

```python
sage: n = 920384092842390423848290348203948092384082349082
sage: n.factor(proof=False)  # needs sage.libs.pari
2 * 11 * 1531 * 4402903 * 10023679 * 6191629572170540533894518173
sage: n.factor(proof=True)  # needs sage.libs.pari
2 * 11 * 1531 * 4402903 * 10023679 * 6191629572170540533894518173
```

(continues on next page)
We factor using trial division only:

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

An example where FLINT is used:

```
sage: n = 82862385732327628428164127822
sage: n.factor(algorithm='flint')
# needs sage.libs.flint
2 * 3 * 11 * 13 * 41 * 73 * 22650083 * 1424602265462161
```

We factor using a quadratic sieve algorithm:

```
sage: # needs sage.libs.pari
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: # needs sage.libs.pari
sage: p = next_prime(10^15)
sage: q = next_prime(10^21)
sage: n = p * q
sage: n.factor(algorithm='ecm')
1000000000000037 * 1000000000000000000117

>>> from sage.all import *
>>> # needs sage.libs.pari
>>> p = next_prime(Integer(10)**Integer(15))
>>> q = next_prime(Integer(10)**Integer(21))
>>> n = p * q
>>> 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()))
0 1
1 1
2 2
3 6
4 24
5 120
6 720

>>> from sage.all import *
>>> for n in srange(Integer(7)):
...     print("{} {}".format(n, n.factorial()))
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

>>> from sage.all import *
>>> (Integer(2)**Integer(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

>>> from sage.all import *
>>> (-Integer(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

>>> from sage.all import *
>>> n = Integer(6)
>>> n.floor()
6

gamma()

The gamma function on integers is the factorial function (shifted by one) on positive integers, and ±∞ on non-positive integers.

EXAMPLES:

sage: # needs sage.symbolic
sage: gamma(5)
24
sage: gamma(0)
Infinity
sage: gamma(-1)
Infinity
sage: gamma(-2^150)
Infinity

>>> from sage.all import *
>>> # needs sage.symbolic
>>> gamma(Integer(5))
24
>>> gamma(Integer(0))
Infinity
>>> gamma(-Integer(1))
Infinity
>>> gamma(-Integer(2)**Integer(150))
Infinity

global_height (prec=None)

Return the absolute logarithmic height of this rational integer.

INPUT:

1.2. Elements of the ring \( \mathbb{Z} \) of integers
• 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(_)
2.0000000000000000000
```

```
>>> from sage.all import *
>>> ZZ(Integer(5)).global_height()
1.60943791243410
>>> ZZ(Integer(-2)).global_height(prec=Integer(100))
0.69314718055994530941723212146
>>> exp(_)
2.0000000000000000000
```

hex()

Return the hexadecimal digits of \( \text{self} \) in lowercase.

**Note:** ‘0x’ is not prepended to the result like is done by the corresponding Python function on \( \text{int} \). 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
```

```
>>> from sage.all import *
>>> print(Integer(Integer(15)).hex())
f
>>> print(Integer(Integer(16)).hex())
10
>>> print(Integer(Integer(16938402384092843092843098243)).hex())
36bb1e3929d1a8fe2802f083
```

imag()

Return the imaginary part of \( \text{self} \), which is zero.

EXAMPLES:
\texttt{sage}: \text{Integer(9).imag()}
0

\texttt{>>> from sage.all import *}
\texttt{>>> Integer(Integer(9)).imag()}
0

\textbf{inverse\_mod}\,(n)

Return the inverse of self modulo \( n \), if this inverse exists.
Otherwise, raise a \texttt{ZeroDivisionError} exception.

INPUT:
• self – Integer
• \( n \) – Integer, or ideal of integer ring

OUTPUT:
• \( x \) – Integer such that \( x \cdot \text{self} = 1 \pmod{\text{n}} \), or raises \texttt{ZeroDivisionError}.

IMPLEMENTATION:
Call the \texttt{mpz\_invert} GMP library function.

EXAMPLES:
\texttt{sage}: a = \text{Integer(189)}
\texttt{sage}: a.inverse\_mod(10000)
4709
\texttt{sage}: a.inverse\_mod(-10000)
4709
\texttt{sage}: a.inverse\_mod(1890)
Traceback (most recent call last):
  ...  
\texttt{ZeroDivisionError: inverse of Mod(189, 1890) does not exist}
\texttt{sage}: a = \text{Integer(19)}**100000 \# long time
\texttt{sage}: c = a.inverse\_mod(a*a) \# long time
Traceback (most recent call last):
  ...  
\texttt{ZeroDivisionError: inverse of Mod(\ldots, \ldots) does not exist}

We check that Issue \#10625 is fixed:
```
sage: ZZ(2).inverse_mod(ZZ.ideal(3))
2
```

```
>>> from sage.all import *
>>> ZZ(Integer(2)).inverse_mod(ZZ.ideal(Integer(3)))
2
```

We check that Issue #9955 is fixed:

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

```
>>> from sage.all import *
>>> Rational(Integer(3)) % Rational(-Integer(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
```

```
>>> from sage.all import *
>>> (Integer(1)).inverse_of_unit()
1
>>> (-Integer(1)).inverse_of_unit()
-1
>>> Integer(5).inverse_of_unit()
Traceback (most recent call last):
  ... ArithmeticError: inverse does not exist
>>> Integer(0).inverse_of_unit()
Traceback (most recent call last):
  ... ArithmeticError: inverse does not exist
```

**is_discriminant()**

Return `True` if this integer is a discriminant.

**Note:** A discriminant is an integer congruent to 0 or 1 modulo 4.

**EXAMPLES:**
is_discriminant()

Return False if this integer is a fundamental discriminant.

Note: A fundamental discriminant is a discriminant, not 0 or 1 and not a square multiple of a smaller discriminant.

is_fundamental_discriminant()

Return True if this integer is a fundamental discriminant.

is_integer()

Return True as they are integers

is Integer of the ring Z of integers
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()
Return True if self is irreducible, i.e. +/- prime

EXAMPLES:

```
sage: z = 2^31 - 1
sage: z.is_irreducible()  # needs sage.libs.pari
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: n = 7
sage: n.is_norm(QQ)
```

(continues on next page)
True
\begin{verbatim}
sage: n.is_norm(QQ, element=True)
(True, 7)

sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x^2 - 2, 'beta')
sage: n = 4
sage: n.is_norm(K)
True
sage: 5.is_norm(K)
False
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)
\end{verbatim}

\section*{is_one()}
Return True if the integer is 1, otherwise False.

\textbf{EXAMPLES:}
\begin{verbatim}
sage: Integer(1).is_one()
True
sage: Integer(0).is_one()
False
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> n = Integer(7)
>>> n.is_norm(QQ)
True
>>> n.is_norm(QQ, element=True)
(True, 7)

>>> # needs sage.rings.number_field
>>> x = polygen(ZZ, 'x')
>>> K = NumberField(x^2 - Integer(2), 'beta')
>>> n = Integer(4)
>>> n.is_norm(K)
True
>>> Integer(5).is_norm(K)
False
>>> n.is_norm(K, element=True)
(True, -4*beta + 6)
>>> n.is_norm(K, element=True)[Integer(1)].norm()
4
>>> n = Integer(5)
>>> n.is_norm(K, element=True)
(False, None)
\end{verbatim}
is_perfect_power()

Return True if self is a perfect power, i.e. 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()
False
sage: (-4).is_perfect_power()
False

>>> from sage.all import *

>>> Integer(-Integer(27)).is_perfect_power()
True
>>> Integer(Integer(12)).is_perfect_power()
False

>>> z = Integer(8)
>>> z.is_perfect_power()
True
>>> Integer(144).is_perfect_power()
True
>>> Integer(10).is_perfect_power()
False
>>> (-Integer(8)).is_perfect_power()
True
>>> (-Integer(4)).is_perfect_power()
False

is_power_of(n)

Return 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.
• \textit{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.

• \textit{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
```

```
>>> from sage.all import *
>>> Integer(Integer(64)).is_power_of(Integer(4))
True
>>> Integer(Integer(64)).is_power_of(Integer(16))
False
```

\textbf{Note:} For large integers self, \textit{is\_power\_of()} is faster than \textit{is\_perfect\_power()}. The following examples give some indication of how much faster.

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

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

```python
sage: z = 2^31 - 1
sage: z.is_prime()    # needs sage.libs.pari
True
sage: z = 2^31
sage: z.is_prime()    False
sage: z = 7
sage: z.is_prime()    True
sage: z = -7
sage: z.is_prime()    False
sage: z.is_irreducible()    True
```

```python
>>> from sage.all import *
>>> z = Integer(2)**Integer(31) - Integer(1)
>>> z.is_prime()    # needs sage.libs.pari
True
>>> z = Integer(2)**Integer(31)
>>> z.is_prime()    False
>>> z = Integer(7)
>>> z.is_prime()    True
>>> z = -Integer(7)
>>> z.is_prime()    False
>>> z.is_irreducible()    True
```

```python
sage: z = 10^80 + 129
sage: z.is_prime(proof=False)    # needs sage.libs.pari
True
sage: z.is_prime(proof=True)    # needs sage.libs.pari
True
```

```python
>>> from sage.all import *
>>> z = Integer(10)**Integer(80) + Integer(129)
>>> z.is_prime(proof=False)    # needs sage.libs.pari
True
>>> z.is_prime(proof=True)    # needs sage.libs.pari
True
```

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

```python
sage: proof.arithmetic()
True
sage: n = 10^100 + 267
```

(continues on next page)
sage: timeit("n.is_prime()")  # not tested  
← needs sage.libs.pari 
5 loops, best of 3: 163 ms per loop 
sage: proof.arithmetic(False) 
sage: proof.arithmetic() 
False  
sage: timeit("n.is_prime()")  # not tested  
← needs sage.libs.pari 
1000 loops, best of 3: 573 us per loop 

>>> from sage.all import * 
>>> proof.arithmetic() 
True 
>>> n = Integer(10)**Integer(100) + Integer(267) 
>>> timeit("n.is_prime()")  # not tested  
← needs sage.libs.pari 
5 loops, best of 3: 163 ms per loop 
>>> proof.arithmetic(False) 
>>> proof.arithmetic() 
False 
>>> timeit("n.is_prime()")  # not tested  
← needs sage.libs.pari 
1000 loops, best of 3: 573 us per loop 

ALGORITHM: 
Calls the PARI function pari:isprime.

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_pseudo-prime()). 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: # needs sage.libs.pari 
sage: 17.is_prime_power() 
True  
sage: 10.is_prime_power() 
(continues on next page)
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

>>> from sage.all import *
>>> # needs sage.libs.pari
>>> Integer(17).is_prime_power()
True
>>> Integer(10).is_prime_power()
False
>>> Integer(64).is_prime_power()
True
>>> (Integer(3)**Integer(10000)).is_prime_power()
True
>>> (Integer(10000)).is_prime_power()
False
>>> (-Integer(3)).is_prime_power()
False
>>> Integer(0).is_prime_power()
False
>>> Integer(1).is_prime_power()
False
>>> p = next_prime(Integer(10)**Integer(20)); p
10000000000000000000039
>>> p.is_prime_power()
True
>>> (p**Integer(97)).is_prime_power()
True
>>> (p + Integer(1)).is_prime_power()
False

With the get_data keyword set to True:

sage: # needs sage.libs.pari
sage: (3^100).is_prime_power(get_data=True)
(3, 100)
sage: 12.is_prime_power(get_data=True)
(12, 0)

(continues on next page)
The method works for large entries when `proof=False`:

```python
sage: proof.arithmetic(False)
sage: ((10^500 + 961)^4).is_prime_power()                      # needs sage.libs.pari
True
```

We check that Issue #4777 is fixed:

```python
sage: n = 150607571^14
sage: n.is_prime_power()                                       # needs sage.libs.pari
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:
\texttt{sage: } z = 2^{31} - 1
\texttt{sage: } z\texttt{.is_pseudoprime()}  
\texttt{# needs sage.libs.pari}
\texttt{True}
\texttt{sage: } z = 2^{31}
\texttt{sage: } z\texttt{.is_pseudoprime()}  
\texttt{# needs sage.libs.pari}
\texttt{False}

\texttt{from sage.all import *}
\texttt{z = Integer(2)**Integer(31) - Integer(1)}
\texttt{z\texttt{.is_pseudoprime()}}  
\texttt{# needs sage.libs.pari}
\texttt{True}
\texttt{z = Integer(2)**Integer(31)}
\texttt{z\texttt{.is_pseudoprime()}}  
\texttt{# needs sage.libs.pari}
\texttt{False}

\textbf{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 \texttt{is\_prime\_power()}.  

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{get\_data=} (default \texttt{False}) if \texttt{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.
\end{itemize}

\textbf{EXAMPLES:}

\texttt{sage: } # needs sage.libs.pari
\texttt{sage: x = 10^200 + 357}
\texttt{sage: x.is_pseudoprime()}
\texttt{True}
\texttt{sage: } (x^12).is_pseudoprime\_power()
\texttt{True}
\texttt{sage: } (x^12).is_pseudoprime\_power(get\_data=True)
\texttt{(1000...000357, 12)}
\texttt{sage: } (997^100).is_pseudoprime\_power()
\texttt{True}
\texttt{sage: } (998^100).is_pseudoprime\_power()
\texttt{False}
\texttt{sage: } ((10^1000 + 453)^2).is_pseudoprime\_power()
\texttt{True}

\texttt{from sage.all import *}
\texttt{# needs sage.libs.pari}
\texttt{x = Integer(10)**Integer(200) + Integer(357)}
\texttt{x.is_pseudoprime()}  
\texttt{True}
\texttt{(x^12).is_pseudoprime\_power()}  
\texttt{True}
\texttt{(x^12).is_pseudoprime\_power(get\_data=True)}
\texttt{(1000...000357, 12)}
\texttt{(Integer(997)**Integer(100)).is_pseudoprime\_power()}  
\texttt{True}
\texttt{(Integer(998)**Integer(100)).is_pseudoprime\_power()}

(continues on next page)
False

>>> ((Integer(10)**Integer(1000) + Integer(453))**Integer(2)).is_pseudoprime_power()
True

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

sage: 5.is_rational()
True

>>> from sage.all import *

sage: Integer(5).is_rational()
True

is_square()
Return True if self is a perfect square.
EXAMPLES:

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

>>> from sage.all import *

sage: Integer(Integer(4)).is_square()  # needs sage.libs.pari
True
sage: Integer(Integer(41)).is_square()  # needs sage.libs.pari
False

is_squarefree()
Return True if this integer is not divisible by the square of any prime and False otherwise.
EXAMPLES:

sage: 100.is_squarefree()  # needs sage.libs.pari
False
sage: 102.is_squarefree()  # needs sage.libs.pari
True
sage: 0.is_squarefree()  # needs sage.libs.pari
False

>>> from sage.all import *

sage: Integer(100).is_squarefree()  # needs sage.libs.pari
False
sage: Integer(102).is_squarefree()  # needs sage.libs.pari
True

(continues on next page)
is_unit()

Return True if this integer is a unit, i.e., 1 or −1.

EXAMPLES:

```python
sage: for n in srange(-2,3):
    ....:    print("{} {}\n{} \n{} \n{} \n{} \n{} \n{} \n{} \n{} \n{} \n{} \n{} \n{} \n{} \n{} \n{}").format(n, n.is_unit()))
-2 False
-1 True
0 False
1 True
2 False
```

isqrt()

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

EXAMPLES:

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

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

```python
sage: from sage.all import *
>>> from sage.all import *
>>> a = Integer(-Integer(102)).isqrt()
Traceback (most recent call last):
...
ValueError: square root of negative integer not defined
```

jacobi(b)

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

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

>>> from sage.all import *
>>> z = -Integer(1)
>>> z.jacobi(Integer(17))
1
>>> z.jacobi(Integer(19))
-1
>>> z.jacobi(Integer(17)*Integer(19))
-1
>>> (Integer(2)).jacobi(Integer(17))
1
>>> (Integer(3)).jacobi(Integer(19))
-1
>>> (Integer(6)).jacobi(Integer(17)*Integer(19))
-1
>>> (Integer(6)).jacobi(Integer(33))
0
>>> a = Integer(3); b = Integer(7)
>>> a.jacobi(b) == -b.jacobi(a)
True

kronecker(b)
Calculate the Kronecker symbol \( \left( \frac{\text{self}}{b} \right) \) with the Kronecker extension \( \left( \frac{\text{self}}{2} \right) = \left( \frac{2}{\text{self}} \right) \) when \( \text{self} \) is odd, or \( \left( \frac{\text{self}}{2} \right) = 0 \) when \( \text{self} \) 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
>>> from sage.all import *
>>> z = Integer(5)
>>> z.kronecker(Integer(41))
1
>>> z.kronecker(Integer(43))
-1
>>> z.kronecker(Integer(8))
-1
>>> z.kronecker(Integer(15))
0
>>> a = Integer(2); b = Integer(5)
>>> 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:**

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

```python
>>> from sage.all import *
>>> m = Integer(5)
>>> m.list()
[5]
```

**log**(m=\(\text{None} \), prec=\(\text{None} \))

Return symbolic log by default, unless the logarithm is exact (for an integer argument). When `prec` 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:**

```python
sage: Integer(124).log(5)  # Needs sage.symbolic
log(124)/log(5)
sage: Integer(124).log(5, 100)  # Needs sage.rings.real_mpfr
2.9950093311241087454822446806
sage: Integer(125).log(5) 3
sage: Integer(125).log(5, prec=53)  # Needs sage.rings.real_mpfr
3.0000000000000000000000000000000000000000000000000000000000000
sage: log(Integer(125))  # Needs sage.symbolic
3*log(5)
```

1.2. Elements of the ring \(Z\) of integers
>>> from sage.all import *
>>> Integer(Integer(124)).log(Integer(5))
→ # needs sage.symbolic
log(124)/log(5)
>>> Integer(Integer(124)).log(Integer(5), Integer(100))
→ # needs sage.rings.real_mpfr
2.995093311241087454822446806
>>> Integer(Integer(125)).log(Integer(5))
3
>>> Integer(Integer(125)).log(Integer(5), prec=Integer(53))
→ # needs sage.rings.real_mpfr
3.00000000000000
>>> log(Integer(Integer(125)))
→ # needs sage.symbolic
3*log(5)

For extremely large numbers, this works:

sage: x = 3**100000
sage: log(x, 3)  # needs sage.rings.real_interval_field
100000

Also \(\log(x)\), giving a symbolic output, works in a reasonable amount of time for this \(x\):

sage: x = 3**100000
sage: log(x)  # needs sage.symbolic
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)  # needs sage.rings.real_mpfr
100000.000000000

sage: (x + 1).log(3, 1000)  # needs sage.rings.real_mpfr
100000.

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

```python
sage: x.log(2.5, prec=53)  # needs sage.rings.real_mpfr
119897.784671579
```

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

```python
sage: log(-1)  # needs sage.symbolic
I*pi
```

The logarithm of zero is done likewise:

```python
sage: log(0)  # needs sage.symbolic
-\infty
```

Some rational bases yield integer logarithms (Issue #21517):

```python
sage: ZZ(8).log(1/2)
-3
```

Check that Python ints are accepted (Issue #21518):
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:
When entries are too large an \texttt{OverflowError} is raised:

\begin{verbatim}
>>> (2**64).multifactorial(2)
Traceback (most recent call last):
  ...  
OverflowError: argument too large for multifactorial

>>> from sage.all import *

>>> (Integer(2)**Integer(64)).multifactorial(Integer(2))
Traceback (most recent call last):
  ...  
OverflowError: argument too large for multifactorial
\end{verbatim}

\texttt{multiplicative\_order()}  
Return the multiplicative order of \texttt{self}.  

\textbf{EXAMPLES:}

\begin{verbatim}
>>> ZZ(1).multiplicative\_order()
1
>>> ZZ(-1).multiplicative\_order()
2
>>> ZZ(0).multiplicative\_order()
+Infinity
>>> ZZ(2).multiplicative\_order()
+Infinity

>>> from sage.all import *

>>> ZZ(Integer(1)).multiplicative\_order()
1
>>> ZZ(-Integer(1)).multiplicative\_order()
2
>>> ZZ(Integer(0)).multiplicative\_order()
+Infinity
>>> ZZ(Integer(2)).multiplicative\_order()
+Infinity
\end{verbatim}

\texttt{nbits()}

Alias for \texttt{bit\_length()}.  

\texttt{ndigits()}  
\texttt{ndigits(\texttt{base=10})}

Return the number of digits of \texttt{self} expressed in the given base.

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{base} – integer (default: 10)
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
>>> n = 52
>>> n.ndigits()
\end{verbatim}
```
2
sage: n = -10003
sage: n.ndigits()  # needs sage.rings.real_interval_field
5
sage: n = 15
sage: n.ndigits(2)  # needs sage.rings.real_interval_field
4
sage: n = 1000**1000000+1
sage: n.ndigits()  # needs sage.rings.real_interval_field
3000001
sage: n = 1000**1000000-1
sage: n.ndigits()  # needs sage.rings.real_interval_field
3000000
sage: n = 10**10000000-10**9999990
sage: n.ndigits()  # needs sage.rings.real_interval_field
10000000

>>> from sage.all import *
>>> n = Integer(52)
>>> n.ndigits()  # needs sage.libs.pari
2
>>> n = -Integer(10003)
>>> n.ndigits()  # needs sage.rings.real_interval_field
5
>>> n = Integer(15)
>>> n.ndigits(Integer(2))
4
>>> n = Integer(1000)**Integer(1000000)+Integer(1)
>>> n.ndigits()  # needs sage.rings.real_interval_field
3000001
>>> n = Integer(1000)**Integer(1000000)-Integer(1)
>>> n.ndigits()  # needs sage.rings.real_interval_field
3000000
>>> n = Integer(10)**Integer(10000000)-Integer(10)**Integer(9999990)
>>> n.ndigits()  # needs sage.rings.real_interval_field
10000000
```

**next_prime**(proof=None)

Return the next prime after self.

This method calls the PARI function pari:nextprime.

**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()  # needs sage.libs.pari
```
```
Use `proof=False`, which is way faster since it does not need a primality proof:

```python
sage: b = (2^1024).next_prime(proof=False)  #
← needs sage.libs.pari
sage: b - 2^1024  #
← needs sage.libs.pari
643
```

```python
sage: b = (Integer(2)**Integer(1024)).next_prime(proof=False)  #
← needs sage.libs.pari
sage: b - Integer(2)**Integer(1024)  #
← needs sage.libs.pari
643
```

```python
sage: Integer(0).next_prime()  #
← needs sage.libs.pari
2
sage: Integer(1001).next_prime()  #
← needs sage.libs.pari
1009
```

```python
sage: Integer(0).next_prime()  #
← needs sage.libs.pari
2
sage: Integer(1001).next_prime()  #
← needs sage.libs.pari
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 goes 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()
sage: 2
sage: 2.next_prime_power()
sage: 3
sage: 103.next_prime_power()        #→ needs sage.libs.pari
  107
sage: 107.next_prime_power()
sage: 109
sage: 2044.next_prime_power()       #→ needs sage.libs.pari
  2048
```

```python
>>> from sage.all import *
>>> (-Integer(1)).next_prime_power()
  2
>>> Integer(2).next_prime_power()
  3
>>> Integer(103).next_prime_power()    #→ needs sage.libs.pari
  107
>>> Integer(107).next_prime_power()
  109
>>> Integer(2044).next_prime_power()   #→ needs sage.libs.pari
  2048
```

`next_probable_prime()`

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

EXAMPLES:

```
sage: # needs sage.libs.pari
sage: (-37).next_probable_prime()
  2
sage: (100).next_probable_prime()
  101
sage: (2^512).next_probable_prime()
  13407807929425970995740249880084612747936582059239337723561443721764030073546976801874298253920868073843707607
sage: 0.next_probable_prime()
  2
sage: 126.next_probable_prime()
  127
sage: 144168.next_probable_prime()
  144169
```
>>> from sage.all import *

# needs sage.libs.pari

>>> (-Integer(37)).next_probable_prime()
2
>>> (Integer(100)).next_probable_prime()
101
>>> (Integer(2)**Integer(512)).next_probable_prime()
1340780792942597099574024998205846127479365820592393377723561443721764030073546976801874298
>>> Integer(0).next_probable_prime()
2
>>> Integer(126).next_probable_prime()
127
>>> Integer(144168).next_probable_prime()
144169

nth_root (n, truncate_mode=0)

Return the (possibly truncated) n-th root of self.

INPUT:

• n – integer ≥ 1 (must fit in the 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 n-th 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)

1.2. Elements of the ring \( \mathbb{Z} \) of integers
ValueError: 124 is not a 3rd power
>>> Integer(Integer(124)).nth_root(Integer(3), truncate_mode=Integer(1))
(4, False)

>>> Integer(Integer(125)).nth_root(Integer(3), truncate_mode=Integer(1))
(5, True)

>>> Integer(Integer(126)).nth_root(Integer(3), truncate_mode=Integer(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)

>>> from sage.all import *

>>> Integer(-Integer(125)).nth_root(Integer(3))
-5

>>> Integer(-Integer(125)).nth_root(Integer(3), truncate_mode=Integer(1))
(-5, True)

>>> Integer(-Integer(124)).nth_root(Integer(3), truncate_mode=Integer(1))
(-4, False)

>>> Integer(-Integer(126)).nth_root(Integer(3), truncate_mode=Integer(1))
(-5, False)

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

sage: Integer(125).nth_root(3, True)
(5, True)

>>> from sage.all import *

>>> Integer(Integer(125)).nth_root(Integer(2), True)
(11, False)

>>> Integer(Integer(125)).nth_root(Integer(3), True)
(5, True)

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

>>> from sage.all import *

>>> Integer(Integer(125)).nth_root(-Integer(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
>>> from sage.all import *
>>> Integer(-Integer(25)).nth_root(Integer(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)

>>> from sage.all import *
>>> a=Integer(9)
>>> a.nth_root(Integer(3))
Traceback (most recent call last):
...  
ValueError: 9 is not a 3rd power

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

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

>>> ZZ(Integer(2)**Integer(20)).nth_root(Integer(21), truncate_mode=Integer(1))
(1, False)

**numerator()**

Return the numerator of this integer.

**EXAMPLES:**

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

>>> from sage.all import *

(continues on next page)
>>> x = Integer(5)
>>> x.numerator()
5

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

>>> from sage.all import *

>>> x = Integer(0)
>>> x.numerator()
0

\textbf{oct} ()

Return the digits of self in base 8.

\textbf{Note:} '0' (or '0o') is not prepended to the result like is done by the corresponding Python function on \texttt{int}. 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.

\textbf{EXAMPLES:}

\begin{verbatim}
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(16938402384092843092843098243).oct())
15535436162247215217705000570203
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> print(Integer(Integer(800)).oct())
1440
>>> print(Integer(Integer(8)).oct())
10
>>> print(Integer(-Integer(50)).oct())
-62
>>> print(Integer(-Integer(899)).oct())
-1603
>>> print(Integer(Integer(16938402384092843092843098243)).oct())
15535436162247215217705000570203
\end{verbatim}

Behavior of Sage integers vs. Python integers:

\begin{verbatim}
sage: Integer(10).oct()
'12'
sage: oct(int(10))
'0o12'
sage: Integer(-23).oct()
\end{verbatim}
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 \( \text{self} \) 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 \( \text{self} \) is 0.

**EXAMPLES:**

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

Order \((p)\)

Return the \( p \)-adic valuation of \( \text{self} \).

**INPUT:**

- \( p \) – an integer at least 2.

**EXAMPLES:**

```python
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

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

ordinal_str()

Return 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'

from sage.all import *

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

ZZ(Integer(1001)).ordinal_str()
'1001st'

ZZ(Integer(113)).ordinal_str()
'113th'

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):
... Value Error: 6 is not a prime number

```python
>>> from sage.all import *
>>> n = Integer(40)
>>> n.p_primary_part(Integer(2))
8
>>> n.p_primary_part(Integer(5))
5
>>> n.p_primary_part(Integer(7))
1
>>> n.p_primary_part(Integer(6))
Traceback (most recent call last):
...
ValueError: 6 is not a prime number
```

**perfect_power()**

Return \((a, b)\), 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:**

```python
sage: 144.perfect_power()         # needs sage.libs.pari
(12, 2)
sage: 1.perfect_power()          (1, 1)
sage: 0.perfect_power()           (0, 1)
sage: (-1).perfect_power()        (-1, 1)
sage: (-8).perfect_power()        # needs sage.libs.pari
(-2, 3)
sage: (-4).perfect_power()        (-4, 1)
sage: (101^29).perfect_power()    # needs sage.libs.pari
(101, 29)
sage: (-243).perfect_power()      # needs sage.libs.pari
(-3, 5)
sage: (-64).perfect_power()       # needs sage.libs.pari
(-4, 3)
```
>>> from sage.all import *
>>> Integer(144).perfect_power()
(12, 2)
>>> Integer(1).perfect_power()
(1, 1)
>>> Integer(0).perfect_power()
(0, 1)
>>> (-Integer(1)).perfect_power()
(-1, 1)
>>> (-Integer(4)).perfect_power()
(-4, 1)
>>> (Integer(101)**Integer(29)).perfect_power()
(101, 29)
>>> (-Integer(243)).perfect_power()
(-3, 5)
>>> (-Integer(64)).perfect_power()
(-4, 3)

popcount()

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

EXAMPLES:

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

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

>>> from sage.all import *
>>> n = Integer(123)
>>> n.str(Integer(2))
'1111011'
>>> n.popcount()
6
>>> n = -Integer(17)
>>> 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

```python
>>> from sage.all import *

>>> z = Integer(2)

>>> z.powermod(Integer(31),Integer(31))
2
>>> z.powermod(Integer(0),Integer(31))
1
>>> z.powermod(-Integer(31),Integer(31)) == Integer(2)**-Integer(31) % Integer(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

```python
>>> from sage.all import *

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

`previous_prime(proof=None)`

Return the previous prime before self.

This method calls the PARI function `pari:precprime`.

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:

```python
sage: 10.previous_prime()  # needs sage.libs.pari
7
sage: 7.previous_prime()  # needs sage.libs.pari
5
sage: 14376485.previous_prime()  # needs sage.libs.pari
14376463
```

(continues on next page)
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)  # needs sage.libs.pari
sage: 2^1024 - b                                 # needs sage.libs.pari
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 goes through the odd numbers calling the method `is_prime_power()`.

**See also:**

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

EXAMPLES:

```
sage: # needs sage.libs.pari
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
```

```
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> Integer(3).previous_prime_power()
2
>>> Integer(103).previous_prime_power()
101
>>> Integer(107).previous_prime_power()
103
>>> Integer(2044).previous_prime_power()
2039
>>> Integer(2).previous_prime_power()
Traceback (most recent call last):
...
ValueError: no prime power less than 2
```

`prime_divisors(*args, **kwds)`

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.

INPUT:

• `limit` – (integer, optional keyword argument) Return only prime divisors up to this bound, and the factorization is done by checking primes up to `limit` using trial division.

Any additional arguments are passed on to the `factor()` method.

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]
```
>>> from sage.all import *
>>> a = Integer(1); a.prime_divisors()
[]
>>> a = Integer(100); a.prime_divisors()
[2, 5]
>>> a = -Integer(100); a.prime_divisors()
[2, 5]
>>> a = Integer(2004); a.prime_divisors()
[2, 3, 167]

Setting the optional limit argument works as expected:

```python
sage: a = 10^100 + 1
sage: a.prime_divisors()  # needs sage.libs.pari
[73, 137, 401, 1201, 1601, 1676321, 5964848081,
  129694419029057750551385771184564274499075700947656757821537291527196801]
sage: a.prime_divisors(limit=10^3)
[73, 137, 401]
sage: a.prime_divisors(limit=10^7)
[73, 137, 401, 1201, 1601, 1676321]
```

prime_factors(*args, **kwds)

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.

INPUT:

- limit – (integer, optional keyword argument) Return only prime divisors up to this bound, and the factorization is done by checking primes up to limit using trial division.

Any additional arguments are passed on to the factor() method.

EXAMPLES:

```python
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]
```
Setting the optional limit argument works as expected:

```python
sage: a = 10^100 + 1
sage: a.prime_divisors()
# Needs sage.libs.pari
[73, 137, 401, 1201, 1601, 1676321, 5964848081, 129694419029057750551385771184564274499075700947656757821537291527196801]
sage: a.prime_divisors(limit=10^3)
[73, 137, 401]
sage: a.prime_divisors(limit=10^7)
[73, 137, 401, 1201, 1601, 1676321]
```

**prime_to_m_part** *(m)*

Return 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:**

```python
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
```

```python
>>> from sage.all import *
>>> Integer(43434).prime_to_m_part(Integer(20))
21717
```
>>>
Integer(2048).prime_to_m_part(Integer(2))
1
>>>
Integer(2048).prime_to_m_part(Integer(3))
2048
>>>
Integer(0).prime_to_m_part(Integer(2))
Traceback (most recent call last):
...
ArithmeticError: self must be nonzero

**quo_rem**(other)

Return 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:**

```python
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)
```

(continues on next page)
\begin{verbatim}
>>> q, r = a.quo_rem(b)
>>> q*b + r == a
True

>>> Integer(3).quo_rem(ZZ['x'].gen(0))
(0, 3)
\end{verbatim}

**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 \textbf{Rational}

**EXAMPLES:**

\begin{verbatim}
sage: (3/7)%100
29
sage: (29).rational_reconstruction(100)
3/7

>>> from sage.all import *

>>> (Integer(3)/Integer(7))%Integer(100)
29
>>> (Integer(29)).rational_reconstruction(Integer(100))
3/7
\end{verbatim}

**real()**

Return the real part of \(self\), which is \(self\).

**EXAMPLES:**

\begin{verbatim}
sage: Integer(-4).real()
-4

>>> from sage.all import *

>>> Integer(-Integer(4)).real()
-4
\end{verbatim}

**round** \((mode='away')\)

Return the nearest integer to \(self\), which is \(self\) since \(self\) is an integer.

**EXAMPLES:**

This example addresses Issue #23502:

\begin{verbatim}
sage: n = 6
sage: n.round()
6
\end{verbatim}
>>> from sage.all import *
>>> n = Integer(6)
>>> n.round()
6

sign()

Return 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

>>> from sage.all import *
>>> Integer(500).sign()
1
>>> Integer(0).sign()
0
>>> (-Integer(10)**Integer(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 \texttt{ValueError} if the square is not in the base ring. Ignored if \texttt{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()
˓→ needs sage.symbolic
sqrt(102)

>>> from sage.all import *
>>> Integer(Integer(144)).sqrt()
12
>>> sqrt(Integer(Integer(144)))
12
>>> Integer(Integer(102)).sqrt()
˓→ # needs sage.symbolic
sqrt(102)

1.2. Elements of the ring $\mathbb{Z}$ of integers
sqrtrem()

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

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

>>> from sage.all import *
>>> Integer(25).sqrtrem()
(5, 0)
>>> Integer(27).sqrtrem()
(5, 2)
>>> Integer(0).sqrtrem()
(0, 0)

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

>>> from sage.all import *
>>> Integer(-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 \( x = z y^2 \), 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

>>> from sage.all import *
>>> squarefree_part(Integer(100))
1

(continues on next page)
sage: a = 8 * 5^6 * 101^2
sage: a.squarefree_part(bound=2).factor()
2 * 5^6 * 101^2
sage: a.squarefree_part(bound=5).factor()
2 * 101^2
sage: a.squarefree_part(bound=1000)
2
sage: a.squarefree_part(bound=2**14)
2
sage: a = 7^3 * next_prime(2^100)**2 * next_prime(2^200)
# needs sage.libs.pari
sage: a / a.squarefree_part(bound=1000)
49

\textbf{str} \hspace{1em} (\textit{base}=10)
\begin{itemize}
  \item Return the string representation of \texttt{self} in the given base.
\end{itemize}
\begin{itemize}
  \item \texttt{sage: Integer(2^10).str(2)}
  \texttt{\textasciitilde'10000000000'}
  \item \texttt{sage: Integer(2^10).str(17)}
  \texttt{\textasciitilde'394'}
\end{itemize}
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]
```
>>> from sage.all import *
>>> factorial(Integer(10)).support()
[2, 3, 5, 7]
>>> (-Integer(999)).support()
[3, 37]

Trying to find the support of 0 raises an ArithmeticError:

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

```python
>>> from sage.all import *
>>> Integer(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:

```python
sage: w = 6
sage: w.str(Integer(2))
'110'
sage: w.test_bit(2)
1
sage: w.test_bit(-1)
0
sage: x = -20
sage: x.str(Integer(2))
'-10100'
sage: x.test_bit(4)
0
sage: x.test_bit(5)
1
sage: x.test_bit(6)
1
```

```python
>>> from sage.all import *
>>> w = Integer(6)
>>> w.str(Integer(2))
'110'
>>> w.test_bit(Integer(2))
1
>>> w.test_bit(-Integer(1))
0
>>> x = -Integer(20)
>>> x.str(Integer(2))
'-10100'
```
Standard Commutative Rings, Release 10.4

```python
>>> x.test_bit(Integer(4))
0
>>> x.test_bit(Integer(5))
1
>>> x.test_bit(Integer(6))
1
```

**to_bytes**(length=1, byteorder='big', is_signed=False)

Return an array of bytes representing an integer.

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

**INPUT:**

- length – positive integer (default: 1); integer is represented in length bytes
- byteorder – str (default: "big"); determines the byte order of the output; can only be "big" or "little"
- is_signed – boolean (default: False); determines whether to use two's compliment to represent the integer

**Todo:** It should be possible to convert straight from the gmp type in cython. This could be significantly faster, but I am unsure of the fastest and cleanest way to do this.

**EXAMPLES:**

```python
sage: (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
sage: (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x02\x00'
sage: (-1024).to_bytes(10, byteorder='big', is_signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\x00'
sage: x = 1000
sage: x.to_bytes((x.bit_length() + 7) // 8, byteorder='little')
b'\xe8\x03'
```

```python
>>> from sage.all import *

>>> (Integer(1024)).to_bytes(Integer(2), byteorder='big')
b'\x04\x00'

>>> (Integer(1024)).to_bytes(Integer(10), byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x02\x00'

>>> (-Integer(1024)).to_bytes(Integer(10), byteorder='big', is_signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\x00'

>>> x = Integer(1000)

>>> x.to_bytes((x.bit_length() + Integer(7)) // Integer(8), byteorder='little')
b'\xe8\x03'
```

**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:**

```python
sage: 11.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:

```python
sage: # needs sage.libs.pari
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)
100000049000057
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: # needs sage.libs.pari
sage: n = next_prime(10^5) * next_prime(10^40); n.trial_division()
100003
```
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> n = next_prime(Integer(10)**Integer(6))*next_prime(Integer(10)**Integer(7)); n.trial_division()
1000003
>>> (-n).trial_division()
1000003
>>> n.trial_division(bound=Integer(100))
1000049000057
>>> n.trial_division(bound=-Integer(10))
Traceback (most recent call last):
  ... ValueError: bound must be positive
>>> n.trial_division(bound=Integer(0))
Traceback (most recent call last):
  ... ValueError: bound must be positive
>>> ZZ(Integer(0)).trial_division()
Traceback (most recent call last):
  ... ValueError: self must be nonzero

>>> # needs sage.libs.pari
>>> n = next_prime(Integer(10)**Integer(5)) * next_prime(Integer(10)**Integer(40)); n.trial_division()
100003
>>> n.trial_division(bound=Integer(10)**Integer(4))
1000030000000000000000000000000012100363
>>> (-n).trial_division(bound=Integer(10)**Integer(4))
1000030000000000000000000000000012100363
>>> (-n).trial_division()
100003

>>> n = Integer(2) * next_prime(Integer(10)**Integer(40)); n.trial_division()
2
>>> n = Integer(3) * next_prime(Integer(10)**Integer(40)); n.trial_division()
3
>>> n = Integer(5) * next_prime(Integer(10)**Integer(40)); n.trial_division()
5

1.2. Elements of the ring $\mathbb{Z}$ of integers
You can specify a starting point:

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

```python
>>> from sage.all import *

>>> n = Integer(3)*Integer(5)*Integer(101)*Integer(103)
>>> n.trial_division(start=Integer(50))
101
```

**trunc()**

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

**EXAMPLES:**

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

>>> from sage.all import *

>>> n = Integer(6)
>>> n.trunc()
6
```

**val_unit(p)**


**INPUT:**

- `p` – an integer at least 2.

**OUTPUT:**

- `v_p(self)` – the p-adic valuation of `self`
- `u_p(self) = self / p^{v_p(self)}`

**EXAMPLES:**

```sage
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)
```

(continues on next page)
valuation \( (p) \)

Return the \( p \)-adic valuation of \( \text{self} \).

INPUT:

- \( p \) – an integer at least 2.

EXAMPLES:

```python
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 an integer → larger than 1.
```

We do not require that \( p \) is a prime:

```python
sage: (2**11).valuation(4)
5
```
```python
>>> from sage.all import *
>>> (Integer(2)**Integer(11)).valuation(Integer(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 \(\text{self}\) 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 Issue #13628.

**EXAMPLES:**

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

```python
>>> from sage.all import *
>>> Integer(6).xgcd(Integer(4))
(2, 1, -1)
```

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

**Bases:** `Integer`

**Rationale for the IntegerWrapper class:**

With `Integer` objects, the allocation/deallocation function slots are hijacked with custom functions that stick already allocated `Integer` objects (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` objects 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: 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.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)
doctest:warning...
DeprecationWarning: The function is_Integer is deprecated;
use 'isinstance(..., Integer)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
sage: is_Integer(2/1)
False
sage: is_Integer(int(2))
False
sage: is_Integer('5')
False

>>> from sage.all import *
>>> from sage.rings.integer import is_Integer
>>> is_Integer(Integer(2))
doctest:warning...
DeprecationWarning: The function is_Integer is deprecated;
use 'isinstance(..., Integer)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True

>>> is_Integer(Integer(2)/Integer(1))
False

>>> is_Integer(int(Integer(2)))
False

>>> is_Integer('5')
False

sage.rings.integer.make_integer(s)

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

EXAMPLES:
sage: from sage.rings.integer import make_integer
sage: make_integer(-29)
-73
sage: make_integer(29)
Traceback (most recent call last):
... 
TypeError: expected str...Integer found

>>> from sage.all import *
>>> from sage.rings.integer import make_integer

>>> make_integer(-29)
-73
>>> make_integer(Integer(29))
Traceback (most recent call last):
... 
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)
Compute $B_k \mod p$, where $B_k$ is the $k$-th Bernoulli number.
If $B_k$ is not $p$-integral, return $-1$.

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)

(continues on next page)
sage: p = 10000019; k = 1000
sage: bernoulli(k) % p
1972762
sage: bernmm_bern_modp(p, k)
1972762

>>> from sage.all import *
>>> from sage.rings.bernmm import bernmm_bern_modp

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

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

Compute 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)
1.4 Bernoulli numbers modulo $p$

**AUTHOR:**

- David Harvey (2006-07-26): initial version
- 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

```python
globals()  # for easy access to sage functionality
sage: bernmm_bern_rat(100, 3)
1
sage: bernmm_bern_rat(100, 3)
-94598037819122125295227433069493721872702841533066936133385696204311395415197247711/
33330
```

**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 \( p = 37 \):

\[
\text{sage}: \text{bernoulli\_mod\_p}(37) \\
[1, 31, 16, 15, 16, 4, 17, 32, 22, 31, 15, 15, 17, 12, 29, 2, 0, 2] \\
\text{sage}: [\text{bernoulli}(n) \mod 37 \text{ for } n \text{ in range}(0, 36, 2)] \\
[1, 31, 16, 15, 16, 4, 17, 32, 22, 31, 15, 15, 17, 12, 29, 2, 0, 2]
\]

Boundary case:

\[
\text{sage}: \text{bernoulli\_mod\_p}(3) \\
[1] \\
\text{sage}: [\text{bernoulli}(n) \mod 3 \text{ for } n \text{ in range}(0, 36, 2)] \\
[1]
\]

AUTHOR:
- David Harvey (2006-08-06)

\[
\text{sage.rings.bernoulli\_mod\_p.bernoulli\_mod\_p\_single}(p, k) \\
\text{Return the Bernoulli number } B_k \text{ mod } p. \\
\text{If } B_k \text{ is not } p\text{-integral, an } \text{ArithmeticError} \text{ is raised.} \\
\text{INPUT:} \\
- p \text{ – integer, a prime} \\
- k \text{ – non-negative integer} \\
\text{OUTPUT:} \\
\text{The } k\text{-th Bernoulli number mod } p. \\
\text{EXAMPLES:}
\]

\[
\text{sage}: \text{bernoulli\_mod\_p\_single}(1009, 48) \\
628 \\
\text{sage}: \text{bernoulli}(48) \mod 1009 \\
628 \\
\text{sage}: \text{bernoulli\_mod\_p\_single}(1, 5) \\
\text{Traceback (most recent call last):} \\
... \\
\text{ValueError}: p (=1) must be a prime >= 3 \\
\text{sage}: \text{bernoulli\_mod\_p\_single}(100, 4) \\
\text{Traceback (most recent call last):} \\
... \\
\text{ValueError}: p (=100) must be a prime
\]
\begin{verbatim}
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

>>> from sage.all import *
>>> bernoulli_mod_p_single(Integer(1009), Integer(48))
628
>>> bernoulli(Integer(48)) % Integer(1009)
628

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

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

>>> bernoulli_mod_p_single(Integer(19), Integer(5))
0

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

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

Check results against \texttt{bernoulli_mod_p}:

\begin{verbatim}
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)
\end{verbatim}
\end{verbatim}
[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]
```

```
>>> from sage.all import *

>>> bernoulli_mod_p(Integer(37))
[1, 31, 16, 15, 16, 4, 17, 32, 22, 31, 15, 15, 17, 12, 29, 2, 0, 2]

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

>>> bernoulli_mod_p(Integer(31))
[1, 26, 1, 17, 1, 9, 11, 27, 14, 23, 13, 22, 14, 8, 14]

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

>>> bernoulli_mod_p(Integer(3))
[1]

>>> [bernoulli_mod_p_single(Integer(3), n) % Integer(3) for n in range(Integer(0), Integer(2), Integer(2))]
[1]

>>> bernoulli_mod_p(Integer(5))
[1, 1]

>>> [bernoulli_mod_p_single(Integer(5), n) % Integer(5) for n in range(Integer(0), Integer(4), Integer(2))]
[1, 1]

>>> bernoulli_mod_p(Integer(7))
[1, 6, 3]

>>> [bernoulli_mod_p_single(Integer(7), n) % Integer(7) for n in range(Integer(0), Integer(6), Integer(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:

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

This one should test that long longs are working:

```python
sage: verify_bernoulli_mod_p(bernoulli_mod_p(next_prime(Integer(3))))
True
sage: verify_bernoulli_mod_p(bernoulli_mod_p(next_prime(Integer(1000))))
True
sage: verify_bernoulli_mod_p([Integer(1), Integer(2), Integer(4), Integer(5), ... Integer(4)])
True
sage: verify_bernoulli_mod_p([Integer(1), Integer(2), Integer(3), Integer(4), ... Integer(5)])
False
```

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 \( F_{\pm}^n(m^2n) \).

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: # needs sage.libs.pari sage.rings.real_interval_field
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: # needs sage.libs.pari
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)
Traceback (most recent call last):
...
ValueError: m has to be positive
```

(continues on next page)
ValueError: n has to be greater than 1
>>> aurifeuillian(Integer(2), Integer(0))
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 \texttt{factor\_aurifeuillian()}. 

\begin{verbatim}
sage.rings.factorint.factor_aurifeuillian (n, check=True)

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

INPUT:

\begin{itemize}
\item \( n \) – integer
\end{itemize}

OUTPUT:

List of factors of \( n \) found by Aurifeuillian factorization.

EXAMPLES:

\begin{verbatim}
sage: # needs sage.libs.pari sage.rings.real_interval_field
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]
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> # needs sage.libs.pari sage.rings.real_interval_field
>>> from sage.rings.factorint import factor_aurifeuillian as fa
>>> fa(Integer(2)**Integer(6) + Integer(1))
[5, 13]
>>> fa(Integer(2)**Integer(58) + Integer(1))
[536838145, 536903681]
>>> fa(Integer(3)**Integer(3) + Integer(1))
[4, 1, 7]
>>> fa(Integer(5)**Integer(5) - Integer(1))
[4, 11, 71]
>>> prod(_) == Integer(5)**Integer(5) - Integer(1)
True
>>> fa(Integer(2)**Integer(4) + Integer(1))
[17]
>>> fa((Integer(6)**Integer(2)*Integer(3)**Integer(3) + Integer(1))
[109, 91, 127]
\end{verbatim}

REFERENCES:

112 Chapter 1. Integers
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:**

```python
sage: from sage.rings.factorint import factor_cunningham
sage: factor_cunningham(2^257-1)  # optional - cunningham_tables
535006138814359 * 1155685395246619182673033 * 374550598501810936581776630096313181393
sage: factor_cunningham((3^101+1)*(2^60).next_prime(), proof=False)  # optional - cunningham_tables
2^2 * 379963 * 1152921504606847009 * 101729152719872329220830935465875077827527
```

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:**

```python
>>> from sage.all import *
>>> from sage.rings.factorint import factor_trial_division
>>> factor_trial_division(Integer(2)**Integer(257)-Integer(1))  # optional - cunningham_tables
2 * 11 * 41835640583745019265831379463815822381094652231
>>> factor_trial_division((Integer(3)**Integer(101)+Integer(1))*(Integer(2)**Integer(60)).next_prime(), proof=False)  # optional - cunningham_tables
2^2 * 379963 * 1152921504606847009 * 101729152719872329220830935465875077827527
```

1.5. Integer factorization functions
1.6 Integer factorization using FLINT

AUTHORS:

- Michael Orlitzky (2023)

sage.rings.factorint_flint.factor_using_flint(n)

Factor the nonzero integer \( n \) using FLINT.

This function returns a list of (factor, exponent) pairs. The factors will be of type Integer, and the exponents will be of type int.

INPUT:

- \( n \) – a nonzero sage Integer; the number to factor.

OUTPUT:

A list of (Integer, int) pairs representing the factors and their exponents.

EXAMPLES:

```python
sage: from sage.rings.factorint_flint import factor_using_flint
sage: n = ZZ(9962572652930382)
sage: factors = factor_using_flint(n)
sage: factors
[(2, 1), (3, 1), (1660428775488397, 1)]
sage: prod( f^e for (f,e) in factors ) == n
True
```

Negative numbers will have a leading factor of \((-1)^1\):

```python
sage: n = ZZ(-1 * 2 * 3)
sage: factor_using_flint(n)
[(-1, 1), (2, 1), (3, 1)]
```

The factorization of unity is empty:

```python
sage: factor_using_flint(ZZ.one())
[]
```

While zero has a single factor, of... zero:

```python
sage: factor_using_flint(ZZ.zero())
[(0, 1)]
```
1.7 Integer factorization using PARI

AUTHORS:

• Jeroen Demeyer (2015)

sage.rings.factorint_pari.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

>>> from sage.all import *

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

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

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

>>> from sage.all import *

>>> alarm(RealNumber('0.5')); factor(Integer(2)**Integer(1000) - Integer(1), ...
˓→verbose=Integer(3))
Traceback (most recent call last):
... 
AlarmInterrupt

>>> pari.get_debug_level()
0
1.8 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 > 43627309.
        - "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] (continues on next page)
sage: prime_range(2,2)
[]
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)
1.9 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 (Issue #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)
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: ...
```

```python
>>> from sage.all import *
>>> from sage.rings.sum_of_squares import four_squares_pyx
>>> four_squares_pyx(Integer(15447))
(2, 5, 17, 123)
>>> Integer(2)**Integer(2) + Integer(5)**Integer(2) + Integer(17)**Integer(2) +
   → Integer(123)**Integer(2)
15447
>>> four_squares_pyx(Integer(523439))
(3, 5, 26, 723)
>>> Integer(3)**Integer(2) + Integer(5)**Integer(2) + Integer(26)**Integer(2) +
   → Integer(723)**Integer(2)
523439
>>> four_squares_pyx(Integer(2)**Integer(32))
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:

```python
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]
```

```python
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:

```python
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):
...
ValueError: 7 is not a sum of 3 squares
```

(continues on next page)
OverflowError: ...

```python
>>> from sage.all import *
>>> from sage.rings.sum_of_squares import three_squares_pyx
>>> three_squares_pyx(Integer(0))
(0, 0, 0)
>>> three_squares_pyx(Integer(1))
(0, 0, 1)
>>> three_squares_pyx(Integer(2))
(0, 1, 1)
>>> three_squares_pyx(Integer(3))
(1, 1, 1)
>>> three_squares_pyx(Integer(4))
(0, 0, 2)
>>> three_squares_pyx(Integer(5))
(0, 1, 2)
>>> three_squares_pyx(Integer(6))
(1, 1, 2)
>>> three_squares_pyx(Integer(7))
Traceback (most recent call last):
...  
ValueError: 7 is not a sum of 3 squares
>>> three_squares_pyx(Integer(107))
(5, 9)
>>> three_squares_pyx(Integer(2)**Integer(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:**

```python
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: ...

>>> from sage.all import *
>>> from sage.rings.sum_of_squares import two_squares_pyx
>>> two_squares_pyx(Integer(1))
(0, 0)
>>> two_squares_pyx(Integer(1))
(0, 1)
>>> two_squares_pyx(Integer(2))
(1, 1)
>>> two_squares_pyx(Integer(3))
Traceback (most recent call last):
... 
ValueError: 3 is not a sum of 2 squares
>>> two_squares_pyx(Integer(106))
(5, 9)
>>> two_squares_pyx(Integer(2)**Integer(32))
Traceback (most recent call last):
... 
OverflowError: ...

1.10 Fast Arithmetic Functions

```python
sage: from sage.arith.functions import LCM_list
sage: LCM_list([3, 9, 30])
90
sage: type(LCM_list([3, 9, 30]))
<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'>

>>> from sage.all import *
>>> w = LCM_list([int(Integer(3)), int(Integer(9)), int(Integer(30))]); w
90
>>> 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, tuple or iterable 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

>>> from sage.all import *
>>> lcm(Integer(97), Integer(100))
9700
>>> LCM(Integer(97), Integer(100))
9700
>>> LCM(Integer(0), Integer(2))
0
>>> LCM(-Integer(3), -Integer(5))
15
>>> LCM([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)])
60
>>> v = LCM(range(Integer(1), Integer(10000)))  # *very* fast!
>>> len(str(v))
4349
1.11 Generic implementation of powering

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

\[
\text{sage.arith.power.generic_power}(a, n)
\]

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
dsage: 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
```

(continues on next page)
>>> a.parent() is F
True
>>> a = generic_power(F(Integer(1)), Integer(2))
>>> a.parent() is F
True

>>> generic_power(Integer(5), Integer(0))
1

>>> generic_power(Integer(2), Integer(5)/Integer(4))
Traceback (most recent call last):
...
NotImplementedError: non-integral exponents not supported

```
sage: class SymbolicMul(str):
    ...
    def __mul__(self, other):
        ...
        return type(self)(s)

sage: x = SymbolicMul("x")

sage: print(generic_power(x, Integer(7)))
(((x*x)*(x*x))*((x*x)*x))
```

```python
>>> from sage.all import *

>>> class SymbolicMul(str):
...    def __mul__(self, other):
...        s = "((**))".format(self, other)
...        return type(self)(s)

>>> x = SymbolicMul("x")

>>> print(generic_power(x, Integer(7)))
(((x*x)*(x*x))*((x*x)*x))
```

### 1.12 Utility classes for multi-modular algorithms

**class** sage.arith.multi_modular.MultiModularBasis

Bases: 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 $\prod_{j=i}^{j-1} m_j^{-1} (\text{mod } m_i)$.

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:

```python
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 [3, 5, 7]

sage: mm.prod() >= 2*height
True

>>> from sage.all import *
>>> from sage.arith.multi_modular import MultiModularBasis_base
>>> mm = MultiModularBasis_base([Integer(3), Integer(5), Integer(7)]); mm
MultiModularBasis with moduli [3, 5, 7]

>>> height = Integer(52348798724)
>>> mm = MultiModularBasis_base(height); mm
MultiModularBasis with moduli [3, 5, 7]

>>> mm.prod() >= Integer(2)*height
True

\textbf{crt} (b)

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

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

\textbf{INPUT}:

\begin{itemize}
  \item \( b \) – a list of length at most self.n
\end{itemize}

\textbf{OUTPUT}:

Integer \( z \) where \( z = b[i] \text{mod} m_i \) for \( 0 \leq i < \text{len}(b) \)

\textbf{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

\begin{verbatim}
>>> from sage.all import *
>>> from sage.arith.multi_modular import MultiModularBasis_base
>>> mm = MultiModularBasis_base([Integer(10007), Integer(10009), Integer(10037), Integer(10039), Integer(17351)])
>>> res = mm.crt([Integer(3), Integer(5), Integer(7), Integer(9)]); res
8474803647063985
>>> res % Integer(10007)
3
>>> res % Integer(10009)
5
\end{verbatim}
extend_with_primes \( (\text{plist}, \text{partial\_products}=`\text{None}, \text{check}=`\text{True}) \)

Extend the stored list of moduli with the given primes in \text{plist}.

\textbf{EXAMPLES:}

```python
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]
```

\textbf{list()}

Return a list with the prime moduli.

\textbf{EXAMPLES:}

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

\textbf{partial\_product} \( (n) \)

Return a list containing precomputed partial products.

\textbf{EXAMPLES:}

```python
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
```
```python
>>> from sage.all import *
>>> from sage.arith.multi_modular import MultiModularBasis_base
>>>
>>> mm = MultiModularBasis_base([Integer(46307), Integer(10007)]);
>>> mm
MultiModularBasis with moduli [46307, 10007]
>>> mm.partial_product(Integer(0))
46307
>>> mm.partial_product(Integer(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:**

```python
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]
```

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

**prod()**

Return the product of the prime moduli.

**EXAMPLES:**

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

```python
>>> from sage.all import *
>>> from sage.arith.multi_modular import MultiModularBasis_base
>>>
>>> mm = MultiModularBasis_base([Integer(46307)]);
>>> mm.prod()
46307
>>> mm = MultiModularBasis_base([Integer(46307), Integer(10007)]); mm
>>> mm.prod()
463394149
```

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

**Bases:** 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 pre-computed data, and return the new prime modulus.

EXAMPLES:

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

```python
>>> from sage.all import *
>>> from sage.arith.multi_modular import MutableMultiModularBasis
>>> mm = MutableMultiModularBasis([Integer(10007)])
>>> p = mm.next_prime()
>>> Integer(1024) < p < Integer(32768)
True
>>> p != Integer(10007)
True
>>> mm.list() == [Integer(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 modulus.

INPUT:

- `ix` – index into list of moduli

OUTPUT: the new prime modulus

EXAMPLES:

```python
sage: from sage.arith.multi_modular import MutableMultiModularBasis
sage: mm = MutableMultiModularBasis([10007, 10009, 10037, 10039])
| MultiModularBasis with moduli [10007, 10009, 10037, 10039]
|.prev_prod = mm.prod(); prev_prod
1009227247850909
| mm.precomputation_list()
[1, 5004, 6536, 6060]
| mm.partial_product(2)
1005306552331
| p = mm.replace_prime(1)
| mm.list() == [10007, p, 10037, 10039]
True
| mm.prod()*10009 == prev_prod*p
True
| precomputed = mm.precomputation_list()
| precomputed == [prod(Integers(mm[i])(1 / mm[j])
.....:     for j in range(i))
.....:     for i in range(4)]
True
```
1.13 Miscellaneous arithmetic functions

AUTHORS:

• Kevin Stueve (2010-01-17): in is_prime(n), delegated calculation to n.is_prime()

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 \( x \equiv a \mod m \) and \( x \equiv b \mod n \), if one exists. By the Chinese Remainder Theorem, a solution to the simultaneous congruences exists if and only if \( a \equiv b \mod \gcd(m, n) \). The solution \( x \) is only well-defined modulo \( \lcm(m, n) \).

If \( a \) and \( b \) are lists, returns a simultaneous solution to the congruences \( x \equiv a_i \mod b_i \), if one exists.

See also:

• CRT_list()
EXAMPLES:

Using \texttt{crt} by giving it pairs of residues and moduli:

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> crt(Integer(2), Integer(1), Integer(3), Integer(5))
11

>>> crt(Integer(13), Integer(20), Integer(100), Integer(301))
28013

>>> crt([Integer(2), Integer(1)], [Integer(3), Integer(5)])
11

>>> crt([Integer(13), Integer(20)], [Integer(100), Integer(301)])
28013
\end{verbatim}

You can also use upper case:

\begin{verbatim}
sage: c = CRT(2,3, 3, 5); c
8
sage: c % 3 == 2
True
sage: c % 5 == 3
True
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> c = CRT(Integer(2),Integer(3), Integer(3), Integer(5)); c
8

>>> c % Integer(3) == Integer(2)
True

>>> c % Integer(5) == Integer(3)
True
\end{verbatim}

Note that this also works for polynomial rings:

\begin{verbatim}
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
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
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> # needs sage.rings.number_field

>>> x = polygen(ZZ, 'x')

>>> K = NumberField(x**Integer(3) - Integer(7), names=('a',)); (a,) = K._first_
\end{verbatim}
You can also do this for any number of moduli:

```python
sage: # needs sage.rings.number_field
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:

```python
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
```

```python
>>> from sage.all import *
```

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

```python
>>> x = polygen(QQ)
>>> crt(Integer(2), Integer(3), x - Integer(1), x + Integer(1))
-1/2*x + 5/2
```

```python
>>> crt(Integer(2), x, x**Integer(2) - Integer(1), x**Integer(2) + Integer(1))
-1/2*x^3 + x^2 + 1/2*x + 1
```

```python
>>> x = polygen(QQ)
>>> crt(Integer(2), Integer(3), x - Integer(1), x + Integer(1))
-1/2*x + 5/2
```

```python
>>> from sage.all import *
```

```python
>>> from sage.all import *
>>> from gmpy2 import mpz
```

Crt also work with numpy and gmpy2 numbers:

```python
sage: import numpy
# needs numpy
sage: import gmpy2
# needs gmpy2
```

(continues on next page)
sage: crt(mpz(2), mpz(3), mpz(7), mpz(11))
58
sage: crt(mpz(2), 3, mpz(7), numpy.int8(11)) # needs numpy
58

>>> from sage.all import *
>>> import numpy
# needs numpy
58
>>>
crt(numpy.int8(Integer(2)), numpy.int8(Integer(3)), numpy.int8(Integer(7)),
numpy.int8(Integer(11))) # needs numpy
58
>>> from gmpy2 import mpz
>>> crt(mpz(Integer(2)), mpz(Integer(3)), mpz(Integer(7)), mpz(Integer(11)))
58
>>> crt(mpz(Integer(2)), Integer(3), mpz(Integer(7)), numpy.int8(Integer(11))) # needs numpy
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 $j \neq i$.

EXAMPLES:

sage: a1 = ZZ(mod(42,5))
sage: a2 = ZZ(mod(Integer(42),Integer(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]]
Given a list `values` of elements and a list of corresponding moduli, find a single element that reduces to each element of `values` modulo the corresponding moduli.

See also:

• `crt()`

**EXAMPLES:**

```python
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:

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

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

```python
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
```

(continues on next page)
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
```

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

```
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.
```

1.13. Miscellaneous arithmetic functions 135
INPUT:

- \( X \) – list or tuple, consisting of lists/tuples/vectors/etc. of integers of the same length
- \( \text{moduli} \) – list of \( \text{len}(X) \) moduli

OUTPUT:

- list – application of CRT componentwise.

EXAMPLES:

```python
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])  # needs sage.modules
[10, 43, 17]

>>> from sage.all import *
>>> CRT_vectors([[Integer(3), Integer(5), Integer(7)], [Integer(3), Integer(5), Integer(11)]], [Integer(2), Integer(3)])
[3, 5, 5]

>>> CRT_vectors([vector(ZZ, [Integer(2), Integer(3), Integer(1)]), Sequence([Integer(1), Integer(7), Integer(8)], ZZ)], [Integer(8), Integer(9)])  # needs sage.modules
[10, 43, 17]
```

```python
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 \( \leq n \) that are relatively prime to \( n \). Thus if \( n \leq 0 \) then \( \text{euler}_\phi(n) \) is defined and equals 0.

INPUT:

- \( n \) – an integer

EXAMPLES:

```python
sage: euler_phi(1)
1
sage: euler_phi(2)
1
sage: euler_phi(3)  # needs sage.libs.pari
2
sage: euler_phi(12)  # needs sage.libs.pari
4
sage: euler_phi(37)  # needs sage.libs.pari
36

>>> from sage.all import *
>>> euler_phi(Integer(1))
1
>>> euler_phi(Integer(2))
1
```
Notice that euler_phi is defined to be 0 on negative numbers and 0.

\[
\text{sage: euler_phi(-1)}
\]
0

\[
\text{sage: euler_phi(0)}
\]
0

\[
\text{sage: type(euler_phi(0))}
\]
<class 'sage.rings.integer.Integer'>

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

\[
\text{sage: euler_phi(21)}
\]
12

\[
\text{sage: [i for i in range(21) if gcd(21,i) == 1]}
\]
[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).

\[
\text{sage: len([i for i in range(21) if gcd(21,i) == 1]) == euler_phi(21)}
\]
True

The phi function also has a special plotting method.
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:**

```python
sage: from sage.arith.misc import Euler_Phi
sage: p = Euler_Phi().plot()
# needs sage.libs.pari sage.plot
sage: p.ymax()
# needs sage.libs.pari sage.plot
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
sage: GCD(2/3, 4/5)
2/15
sage: GCD([2,4,6,8])
2
sage: GCD(srange(0,10000,10))
# fast !
10
```

>>> from sage.all import *

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

Note that to take the gcd of n elements for n ≠ 2 you must put the elements into a list by enclosing them in [ .. ]. Before Issue #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 ...
```

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

```sage
sage: gcd(3)
```

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

```sage
>>> from sage.all import *

>>> gcd(Integer(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
sage: gcd([])
0
```

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

```sage
>>> from sage.all import *

>>> gcd([])
0
```

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

class sage.arith.misc.Moebius

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
# needs sage.libs pari
sage: moebius(-5)
```
The moebius function even makes sense for non-integer inputs.

Tests with numpy and gmpy2 numbers:

1.13. Miscellaneous arithmetic functions
sage: moebius(mpz(-5))  # needs sage.libs.pari
-1

>>> from sage.all import *
>>> from numpy import int8  # needs numpy

>> moebius(int8(-Integer(5)))  # needs numpy sage.libs.pari
-1

>>> from gmpy2 import mpz

>>> moebius(mpz(-Integer(5)))  # needs sage.libs.pari
-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:

142 Chapter 1. Integers

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: # needs sage.libs.pari
sage: v = moebius.range(-10, 10); v
[1, 0, 0, -1, 1, -1, 0, -1, -1, 1, 0, 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

>>> from sage.all import *

>>> # needs sage.libs.pari

>>> v = moebius.range(-Integer(10), Integer(10)); v
[1, 0, 0, -1, 1, -1, 0, -1, -1, 1, 0, 1, -1, 0, -1, 1, -1, 0, 0]
>>> v == [moebius(n) for n in range(-Integer(10), Integer(10))]
True

>>> v = moebius.range(-Integer(1000), Integer(2000), Integer(4))
>>> v == [moebius(n) for n in range(-Integer(1000), Integer(2000), Integer(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

    >>> from sage.all import *
    >>> # needs sage.plot
    >>> sigma(Integer(5))
    6
    >>> sigma(Integer(5), Integer(2))
    26

    The sigma function also has a special plotting method.

    sage: P = plot(sigma, 1, 100)  # needs sage.plot

    >>> from sage.all import *
    >>> # needs sage.plot
    >>> P = plot(sigma, Integer(1), Integer(100))
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()  # needs sage.libs.pari sage.plot
sage: p.ymax()  # needs sage.libs.pari sage.plot
124.0
```

```
from sage.all import *
from sage.arith.misc import Sigma

p = Sigma().plot()  # needs sage.libs.pari sage.plot
p.ymax()  # needs sage.libs.pari sage.plot
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 \(\text{xgcd}\) 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)      # needs sage.rings.number_field
sage: g.xgcd(g + 2)                       # needs sage.rings.number_field
(1, 1/3*g, 0)

sage: # needs sage.rings.number_field
sage: R.<a,b> = K[

sage: S.<y> = R.fraction_field()

sage: xgcd(y^2, a*y + b)
(1, a^2 + (2*g)*a + 3)/(b^3 + g*b^2), ((-a + (-g))/b^2)*y + 1/b)

>>> from sage.all import *

>>> xgcd(Integer(56), Integer(44))
(4, 4, -5)
>>> Integer(4)*Integer(56) + (-Integer(5))*Integer(44)
4

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

>>> x = polygen(QQ)
>>> xgcd(x**Integer(3) - Integer(1), x**Integer(2) - Integer(1))
(x - 1, 1, -x)

>>> K = NumberField(x**Integer(2) - Integer(3), names=('g',)); (g,) = K._first_
    _ngens(1) # needs sage.rings.number_field
>>> g.xgcd(g + Integer(2))
```
Here is an example of a \texttt{xgcd} for two polynomials over the integers, where the linear combination is not the gcd but the gcd multiplied by the resultant:

\begin{Verbatim}
\texttt{sage: R,\langle x\rangle = ZZ[]} \\texttt{[}\]
\texttt{sage: gcd(2*x*(x-1), x^2)} \\texttt{x}
\texttt{sage: xgcd(2*x*(x-1), x^2)} \\texttt{(2*x, -1, 2)}
\texttt{sage: (2*(x-1)).resultant(x)} \\texttt{#...}
\end{Verbatim}

Tests with numpy and gmpy2 types:

\begin{Verbatim}
\texttt{sage: from numpy import int8} \\texttt{#...}
\texttt{\rightarrow needs numpy}
\texttt{sage: xgcd(4, int8(8))} \\texttt{#...}
\texttt{\rightarrow needs numpy}
\texttt{(4, 1, 0)}
\texttt{sage: xgcd(int8(4), int8(8))} \\texttt{#...}
\texttt{\rightarrow needs numpy}
\texttt{(4, 1, 0)}
\texttt{sage: from gmpy2 import mpz}
\texttt{sage: xgcd(mpz(4), mpz(8))}
\texttt{(4, 1, 0)}
\texttt{sage: xgcd(4, mpz(8))}
\texttt{(4, 1, 0)}
\end{Verbatim}

\begin{Verbatim}
\texttt{>>> from sage.all import *}
\texttt{>>> from numpy import int8} \\texttt{#...}
\texttt{\rightarrow needs numpy}
\texttt{\rightarrow needs numpy}
\texttt{\rightarrow needs numpy}
\texttt{\rightarrow needs numpy}
\end{Verbatim}
sage.arith.misc.algdep($z$, $\text{degree}$, 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 $\text{degree}$ 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
- $\text{degree}$ – an integer
- $\text{height\_bound}$ – an integer (default: None) specifying the maximum coefficient size for the returned polynomial
- $\text{proof}$ – a boolean (default: False), requires height_bound to be set

EXAMPLES:

```python
sage: algdep(1.888888888888888, 1)  # needs sage.libs.pari
9*x - 17
sage: algdep(0.12121212121212, 1)  # needs sage.libs.pari
33*x - 4
sage: algdep(sqrt(Integer(2)), Integer(2))  # needs sage.libs.pari sage.symbolic
x^2 - 2

>>> from sage.all import *
>>> algdep(RealNumber('1.8888888888888888888'), Integer(1))    # needs sage.libs.pari
9*x - 17
>>> algdep(RealNumber('0.1212121212121212'), Integer(1))     # needs sage.libs.pari
33*x - 4
>>> algdep(sqrt(Integer(2)), Integer(2))                        # needs sage.libs.pari sage.symbolic
x^2 - 2
```

1.13. Miscellaneous arithmetic functions 147
This example involves a complex number:

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

This example involves a \( p \)-adic number:

```python
sage: K = Qp(3, print_mode='series')
# needs sage.rings.padics
sage: a = K(7/19); a
# needs sage.rings.padics
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)
# needs sage.rings.padics
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:

```python
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 -
21187195088
sage: factor(p)
227004321085*x^4 - 216947902586*x^3 - 99411220986*x^2 + 82234881648*x -
21187195088
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
```

(continues on next page)
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:

```python
sage: algdep(pi.n(), 5, height_bound=10, proof=True) is None
# needs sage.libs.pari sage.symbolic
True
```

For stronger results, we need more precision:

```python
sage: algdep(pi.n(), 5, height_bound=100, proof=True) is None
# needs sage.libs.pari sage.symbolic
True
```

(continues on next page)
ValueError: insufficient precision for non-existence proof

```python
>>> algdep(pi.n(Integer(200)), Integer(5), height_bound=Integer(100), proof=True)
   is None
True
```

Traceback (most recent call last):
...
ValueError: insufficient precision for non-existence proof

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

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

```python
sage: # needs sage.libs.pari sage.symbolic
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(), 8, height_bound=1000, proof=True); f
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
sage: f(a).expand()
0
```

```python
>>> from sage.all import *
>>> # needs sage.libs.pari sage.symbolic
>>> a = sqrt(Integer(2)) + sqrt(Integer(3)) + sqrt(Integer(5))

>>> algdep(a.n(Integer(1000)), Integer(8), height_bound=Integer(1000), proof=True)
```

```python
>>> f = algdep(a.n(Integer(1000)), Integer(8), height_bound=Integer(1000), proof=True); f
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
```

```python
>>> 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:
Standard Commutative Rings, Release 10.4

- $z$ – 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)  # needs sage.libs.pari
9*x - 17

sage: algdep(0.12121212121212, 1)  # needs sage.libs.pari
33*x - 4

sage: algdep(sqrt(2), 2)  # needs sage.libs.pari sage.symbolic
x^2 - 2
```

```
>>> from sage.all import *

>>> algdep(RealNumber('1.888888888888888'), Integer(1))  # needs sage.libs.pari
9*x - 17

>>> algdep(RealNumber('0.12121212121212'), Integer(1))  # needs sage.libs.pari
33*x - 4

>>> algdep(sqrt(Integer(2)), Integer(2))  # needs sage.libs.pari sage.symbolic
x^2 - 2
```

This example involves a complex number:

```
sage: z = (1/2) * (1 + RDF(sqrt(3)) * CC.0); z  # needs sage.symbolic
0.500000000000000 + 0.866025403784439*I

sage: algdep(z, 6)  # needs sage.libs.pari sage.symbolic
x^2 - x + 1
```

```
>>> from sage.all import *

>>> z = (Integer(1)/Integer(2)) * (Integer(1) + RDF(sqrt(Integer(3))) * CC.gen(0)); z  # needs sage.symbolic
0.500000000000000 + 0.866025403784439*I

>>> algdep(z, Integer(6))  # needs sage.symbolic
x^2 - x + 1
```

This example involves a $p$-adic number:

```
sage: K = Qp(3, print_mode='series')  # needs sage.rings.padics

sage: a = K(7/19); a  # needs sage.rings.padics
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)  # needs sage.libs.pari sage.symbolic
```

(continues on next page)
These examples show the importance of proper precision control. We compute a 200-bit approximation to $\sqrt{2}$ which is wrong in the 33'th bit:

```python
>>> from sage.all import *
>>> K = Qp(Integer(3), print_mode='series')
→ # needs sage.rings.padics
>>> a = K(Integer(7)/Integer(19)); a
→ # needs sage.rings.padics
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)
>>> algdep(a, Integer(1))
→ # needs sage.rings.padics
19*x - 7
```

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:

```python
sage: # needs sage.libs.pari sage.rings.real_mpfr
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:
For stronger results, we need more precision:

```
sage: algdep(sqrt(2) + sqrt(3) + sqrt(5).n(), 8, height_bound=100, proof=True) is None  Traceback (most recent call last):
  ...  ValueError: insufficient precision for uniqueness proof
sage: f = algdep(sqrt(2) + sqrt(3) + sqrt(5).n(1000), 8, height_bound=1000, proof=True); f
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
sage: f(sqrt(2) + sqrt(3) + sqrt(5)).expand() 0
```

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
```

We can also use proof=True to get positive results:
```python
>>> # needs sage.libs.pari sage.symbolic
>>> a = sqrt(Integer(2)) + sqrt(Integer(3)) + sqrt(Integer(5))
>>> algdep(a.n(), Integer(8), height_bound=Integer(1000), proof=True)
Traceback (most recent call last):
... ValueError: insufficient precision for uniqueness proof
>>> f = algdep(a.n(Integer(1000)), Integer(8), height_bound=Integer(1000),
→ proof=True); f
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
>>> 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 <= 30000 and 'bernmmp' for larger values (this is just a heuristic, and not guaranteed to be optimal on all hardware)
  - 'arb' – use the bernoulli_fmpq_ui function (formerly part of Arb) of the FLINT library
  - 'flint' – use the arith_bernoulli_number function of the FLINT library
  - 'pari' – use the PARI C library
  - 'gap' – use GAP
  - 'gp' – use PARI/GP interpreter
  - 'magma' – use MAGMA (optional)
  - 'bernmmp' – use bernmm package (a multimodular algorithm)
- num_threads – positive integer, number of threads to use (only used for bernmm algorithm)

**EXAMPLES:**

```python
sage: bernoulli(12)  # needs sage.libs.flint
-691/2730
sage: bernoulli(50)  # needs sage.libs.flint
495057205241079648212477525/66
```

```python
>>> from sage.all import *
>>> bernoulli(Integer(12))  # needs sage.libs.flint
-691/2730
>>> bernoulli(Integer(50))  # needs sage.libs.flint
495057205241079648212477525/66
```

We demonstrate each of the alternative algorithms:
AUTHOR:

• David Joyner and William Stein

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

Return the binomial coefficient

\[ \binom{x}{m} = \frac{x(x-1)\cdots(x-m+1)}{m!} \]
which is defined for \( m \in \mathbb{Z} \) 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)
#needs sage.libs.pari
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)
#needs sage.rings.real_mpfr
1.87500000000000
sage: binomial(Zp(5)(99),50)
3 + 4*5^3 + 2*5^4 + 4*5^5 + 4*5^6 + 4*5^7 + 5^9 + 2*5^10 + 3*5^11 +
4*5^12 + 4*5^13 + 2*5^14 + 3*5^15 + 3*5^16 + 4*5^17 + 4*5^18 + 2*5^19 + O(5^20)
sage: binomial(Qp(3)(2/3),2)
2*3^-2 + 2*3^-1 + 2 + 2*3 + 2*3^2 + 2*3^3 + 2*3^4 + 2*3^5 + 2*3^6 + 2*3^7 +
2*3^8 + 2*3^9 + 2*3^10 + 2*3^11 + 2*3^12 + 2*3^13 + 2*3^14 + 2*3^15 + 2*3^16 +...

sage: n = var('n'); binomial(n, 2)
#needs sage.symbolic
1/2*(n - 1)*n
sage: n = var('n'); binomial(n, n)
#needs sage.symbolic
1
sage: n = var('n'); binomial(n, n - 1)
#needs sage.symbolic
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
```
>>> from sage.all import *
>>> from sage.arith.misc import binomial
>>> binomial(Integer(5), Integer(2))
10
>>> binomial(Integer(2), Integer(0))
1
>>> binomial(Integer(1)/Integer(2), Integer(0))  # needs sage.libs.pari
1
>>> binomial(Integer(3), -Integer(1))
0
>>> binomial(Integer(20), Integer(10))
184756
>>> binomial(-Integer(2), Integer(5))
-6
>>> binomial(-Integer(5), -Integer(2))
0
>>> binomial(RealField()('2.5'), Integer(2))  # needs sage.rings.real_mpfr
1.87500000000000
>>> binomial(Zp(Integer(5))(Integer(99)), Integer(50))
3 + 4*5^3 + 2*5^4 + 4*5^5 + 4*5^6 + 4*5^7 + 4*5^8 + 5^9 + 2*5^10 + 3*5^11 + 4*5^12 + 4*5^13 + 2*5^14 + 3*5^15 + 3*5^16 + 4*5^17 + 4*5^18 + 2*5^19 + O(5^20)
>>> binomial(Qp(Integer(3))(Integer(2)/Integer(3)), Integer(2))  # needs sage.libs.pari
2*3^-2 + 2*3^-1 + 2 + 2*3 + 2*3^2 + 2*3^3 + 2*3^4 + 2*3^5 + 2*3^6 + 2*3^7 + 2*3^8 + 2*3^9 + 2*3^10 + 2*3^11 + 2*3^12 + 2*3^13 + 2*3^14 + 2*3^15 + O(3^18)
>>> n = var('n'); binomial(n, Integer(2))  # needs sage.symbolic
1/2*(n - 1)*n
>>> n = var('n'); binomial(n, n)  # needs sage.symbolic
1
>>> n = var('n'); binomial(n, n - Integer(1))  # needs sage.symbolic
n
>>> binomial(Integer(2)**Integer(100), Integer(2)**Integer(100))
1

>>> x = polygen(ZZ)
>>> binomial(x, Integer(3))
1/6*x^3 - 1/2*x^2 + 1/3*x
>>> binomial(x, x - Integer(3))
1/6*x^3 - 1/2*x^2 + 1/3*x

If $x \in \mathbb{Z}$, 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')  # needs sage.libs.pari
sage: a == b  # needs sage.libs.pari
True
```
sage.arith.misc.binomial_coefficients(n)

Return a dictionary containing pairs \{(k_1, k_2) : C_{k_1,k_2} \} where \( C_{k_1,k_2} \) 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)]

>>> from sage.all import *

>>> sorted(binomial_coefficients(Integer(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

>>> from sage.all import *

>>> R = QQ['x', 'y']; (x, y,) = R._first_ngens(2)

>>> (x+y)**Integer(3)
x^3 + 3*x^2*y + 3*x*y^2 + y^3

Tests with numpy and gmpy2 numbers:

sage: from numpy import int8

# needs numpy
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.carmichael_lambda(n)`

Return the Carmichael function of a positive integer \( n \).

The Carmichael function of \( n \), denoted \( \lambda(n) \), is the smallest positive integer \( k \) such that \( a^k \equiv 1 \mod n \) for all \( a \in \mathbb{Z}/n\mathbb{Z} \) satisfying \( \gcd(a, n) = 1 \). Thus, \( \lambda(n) = k \) is the exponent of the multiplicative group \( (\mathbb{Z}/n\mathbb{Z})^* \).

**INPUT:**
- \( n \) – a positive integer.

**OUTPUT:**
- The Carmichael function of \( n \).

**ALGORITHM:**
- If \( n = 2, 4 \) then \( \lambda(n) = \varphi(n) \). Let \( p \geq 3 \) be an odd prime and let \( k \) be a positive integer. Then \( \lambda(p^k) = p^{k-1}(p-1) = \varphi(p^k) \). If \( k \geq 3 \), then \( \lambda(2^k) = 2^{k-2} \). Now consider the case where \( n > 3 \) is composite and let \( n = p_1^{k_1}p_2^{k_2} \cdots p_t^{k_t} \) be the prime factorization of \( n \). Then

\[
\lambda(n) = \lambda(p_1^{k_1}, p_2^{k_2}, \ldots, p_t^{k_t}) = \text{lcm}(\lambda(p_1^{k_1}), \lambda(p_2^{k_2}), \ldots, \lambda(p_t^{k_t}))
\]

**EXAMPLES:**

The Carmichael function of all positive integers up to and including 10:

```python
sage: from sage.arith.misc import carmichael_lambda
sage: list(map(carmichael_lambda, [1..10]))
[1, 1, 2, 2, 4, 2, 6, 2, 6, 4]
```

The Carmichael function of the first ten primes:

```python
sage: list(map(carmichael_lambda, primes_first_n(10)))
[1, 2, 4, 6, 10, 12, 16, 18, 22, 28]
```

Cases where the Carmichael function is equivalent to the Euler phi function:

```python
sage: carmichael_lambda(2) == euler_phi(2)
True
sage: carmichael_lambda(4) == euler_phi(4)
True
```

(continues on next page)
A case where $\lambda(n) \neq \varphi(n)$:

```python
sage: k = randint(3, 1000)
sage: c = carmichael_lambda(2^k)
```

Verifying the current implementation of the Carmichael function using another implementation. The other implementation that we use for verification is an exhaustive search for the exponent of the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\ast$. 

```python
sage: from sage.arith.misc import carmichael_lambda
sage: n = randint(1, 500)
sage: c = carmichael_lambda(n)
sage: def coprime(n):
...:     return [i for i in range(n) if gcd(i, n) == 1]
sage: def znpower(n, k):
...:     L = coprime(n)
...:     return list(map(power_mod, L, [k]*len(L), [n]*len(L)))
```

True

```python
>>> from sage.all import *
>>> from sage.arith.misc import carmichael_lambda
>>> n = randint(Integer(1), Integer(500))
>>> c = carmichael_lambda(n)
>>> def coprime(n):
...     return [i for i in range(n) if gcd(i, n) == Integer(1)]
>>> def znpower(n, k):
...     L = coprime(n)
...     return list(map(power_mod, L, [k]*len(L), [n]*len(L)))
>>> def my_carmichael(n):
...     if n == Integer(1):
...         return Integer(1)
...     for k in range(Integer(1), n):
...         L = znpower(n, k)
...         ones = [Integer(1)] * len(L)
...         T = [L[i] == ones[i] for i in range(len(L))]
...         if all(T):
...             return k

>>> c == my_carmichael(n)
True
```

Carmichael's theorem states that $a^{\lambda(n)} \equiv 1 \pmod{n}$ for all elements $a$ of the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$. Here, we verify Carmichael's theorem.

```python
sage: from sage.arith.misc import carmichael_lambda
sage: n = randint(2, 1000)
sage: c = carmichael_lambda(n)
sage: ZnZ = IntegerModRing(n)
sage: M = ZnZ.list_of_elements_of_multiplicative_group()
sage: ones = [Integer(1)] * len(M)
sage: P = [power_mod(a, c, n) for a in M]
sage: P == ones
True
```

REFERENCES:

- Wikipedia article Carmichael function

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

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

If $n = \text{None}$ or $n > \text{len}(v)$ the default $n = \text{len}(v)$ is used.

**INPUT:**

• $v$ – list or tuple of elements of a ring
• $n$ – optional integer

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

**EXAMPLES:**

```python
sage: continuant([1, 2, 3])
10
sage: p = continuant([2, 1, 2, 1, 4, 1, 6, 1, 8, 1, 1, 10])
517656/190435
sage: q = continuant([1, 2, 1, 4, 1, 6, 1, 8, 1, 1, 10])
517656/190435
sage: F = continued_fraction([2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10])
1
sage: F.convergent(14)
517656/190435
```

We verify the identity

$$K_n(z, z, \ldots, z) = \sum_{k=0}^{n} \binom{n-k}{k} z^{n-2k}$$
for \( n = 6 \) using polynomial arithmetic:

```python
sage: z = QQ['z'].0
sage: continuant((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()
```

```python
>>> from sage.all import *
>>> z = QQ['z'].gen(0)
>>> continuant((z,z,z,z,z,z,z,z,z,z,z,z), Integer(6))
z^6 + 5*z^4 + 6*z^2 + 1
>>> continuant(Integer(9))
Traceback (most recent call last):
 ...  
TypeError: object of type 'sage.rings.integer.Integer' has no len()
```

Tests with numpy and gmpy2 numbers:

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

```python
>>> from sage.all import *
>>> from numpy import int8
# needs numpy
>>> continuant([int8(Integer(1)), int8(Integer(2)), int8(Integer(3))])
 ...  
# needs numpy
10
>>> from gmpy2 import mpz
>>> continuant([mpz(Integer(1)), mpz(Integer(2)), mpz(Integer(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 \( x \equiv a \mod m \) and \( x \equiv b \mod n \), 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:

- \( \text{CRT\_list()} \)

**EXAMPLES:**

Using \( \text{crt} \) by giving it pairs of residues and moduli:

\[
\begin{align*}
\text{sage: } & \text{crt}(2, 1, 3, 5) \\
& 11 \\
\text{sage: } & \text{crt}(13, 20, 100, 301) \\
& 28013 \\
\text{sage: } & \text{crt}([2, 1], [3, 5]) \\
& 11 \\
\text{sage: } & \text{crt}([13, 20], [100, 301]) \\
& 28013
\end{align*}
\]

```python
>>> from sage.all import *
>>> c = CRT(Integer(2), Integer(3), Integer(3), Integer(5)); c
8
>>> c % Integer(3) == Integer(2)
True
>>> c % Integer(5) == Integer(3)
True
```

Note that this also works for polynomial rings:

\[
\begin{align*}
\text{sage: } & \text{# needs sage.rings.number_field} \\
\text{sage: } & x = \text{polygen}(\mathbb{Z}[x]) \\
\text{sage: } & K.<a> = \text{NumberField}(x^3 - 7) \\
\text{sage: } & R.<y> = K[] \\
\text{sage: } & f = y^2 + 3 \\
\text{sage: } & g = y^3 - 5 \\
\text{sage: } & \text{CRT}(1, 3, f, g) \\
& -3/26*y^4 + 5/26*y^3 + 15/26*y + 53/26
\end{align*}
\]

(continues on next page)
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: # needs sage.rings.number_field
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

You can also do this for any number of moduli:

sage: # needs sage.rings.number_field
sage: R = K[x]; (x,) = R._first_ngens(1)
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([Integer(1), a, Integer(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

(continues on next page)
+ (-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

```
>>> k.mod(f)
1
>>> k.mod(g)
a
>>> 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: from sage.all import *
>>> from sage.all import *
>>> crt(Integer(4), Integer(8), Integer(8), Integer(12))
20
>>> crt(Integer(4), Integer(6), Integer(8), Integer(12))
Traceback (most recent call last):
  ... ValueError: no solution to crt problem since gcd(8,12) does not divide 4-6
>>> x = polygen(QQ)
>>> crt(Integer(2), Integer(3), x - Integer(1), x + Integer(1))
-1/2*x + 5/2
>>> crt(Integer(2), x, x**Integer(2) - Integer(1), x**Integer(2) + Integer(1))
-1/2*x^3 + x^2 + 1/2*x + 1
>>> crt(Integer(2), x, x**Integer(2) - Integer(1), x**Integer(3) - Integer(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
>>> crt(int(Integer(2)), int(Integer(3)), int(Integer(7)), int(Integer(11)))
58
```

crt also work with numpy and gmpy2 numbers:
sage: import numpy  # needs numpy
sage: crt(numpy.int8(2), numpy.int8(3), numpy.int8(7), numpy.int8(11))  # needs numpy
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))  # needs numpy
58

>>> from sage.all import *
>>> import numpy  # needs numpy

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

\[ \psi(n) = n \prod_{p|n, p \text{ prime}} (1 + 1/p). \]

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

\[ s(p, q) = \sum_{i=0}^{q-1} \left( \left\lfloor \frac{ip}{q} \right\rfloor - \left\lfloor \frac{pi}{q} \right\rfloor \right). \]
where

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

**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:

```python
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, 0, -1/14, -5/14, 0]
[0, 7/16, 1/8, 0, -1/16, -1/8, -7/16, 0]
[0, 14/27, 4/27, 1/18, -4/27, 4/27, -1/18, -14/27, 0]
```

Check relations for restricted arguments:

```python
sage: q = 23; dedekind_sum(1, q); (q-1)*(q-2)/(12*q)
77/46
```

(continues on next page)
We check that evaluation works with large input:

\[
\text{sage: dedekind_sum}(3^{54} - 1, 2^{93} + 1) \quad \text{\# needs \text{sage.libs.flint}
}\]

\[
\text{sage: dedekind_sum}(3^{54} - 1, 2^{93} + 1, \text{algorithm}=\text{'pari'}) \quad \text{\# needs \text{sage.libs.pari}
}\]

We check consistency of the results:

\[
\text{sage: dedekind_sum}(5, 7, \text{algorithm}=\text{'default'}) \quad \text{\# needs \text{sage.libs.flint}
}\]

\[
\text{sage: dedekind_sum}(5, 7, \text{algorithm}=\text{'flint'}) \quad \text{\# needs \text{sage.libs.flint}
}\]

\[
\text{sage: dedekind_sum}(5, 7, \text{algorithm}=\text{'pari'}) \quad \text{\# needs \text{sage.libs.pari}
}\]

(continues on next page)
Tests with numpy and gmpy2 numbers:

```python
dedekind_sum(int8(Integer(5)), int8(Integer(7)), algorithm='default')  # needs numpy sage.libs.flint
```

REFERENCES:

- [Ap1997]
- Wikipedia article Dedekind_sum

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

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

EXAMPLES:
Tests with numpy and gmpy2 numbers:

sage: from numpy import int8  # needs numpy
sage: differences([int8(1), int8(4), int8(6), int8(19)]) # needs numpy
[3, 2, 13]

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

>>> from sage.all import *
>>> from numpy import int8  # needs numpy
>>> differences([int8(Integer(1)), int8(Integer(4)), int8(Integer(6)), int8(Integer(19))]) # needs numpy
[3, 2, 13]

>>> from gmpy2 import mpz
>>> differences([mpz(Integer(1)), mpz(Integer(4)), mpz(Integer(6)), mpz(Integer(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:

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

This function works whenever one has unique factorization:

\[
\begin{align*}
\text{>>> from sage.all import *} \\
\text{>>> divisors(-Integer(3))} \\
& [1, 3] \\
\text{>>> divisors(Integer(6))} \\
& [1, 2, 3, 6] \\
\text{>>> divisors(Integer(28))} \\
& [1, 2, 4, 7, 14, 28] \\
\text{>>> divisors(Integer(2)^2 \times Integer(5))} \\
& [1, 2, 4, 8, 16, 32] \\
\text{>>> divisors(Integer(100))} \\
& [1, 2, 4, 5, 10, 20, 25, 50, 100] \\
\text{>>> divisors(Integer(1))} \\
& [1] \\
\text{>>> divisors(Integer(0))} \\
& \text{Traceback (most recent call last):} \\
& ... \\
& \text{ValueError: } n \text{ must be nonzero} \\
\text{>>> divisors(Integer(2)^2 \times Integer(3) \times Integer(3)^2 \times Integer(2) \times Integer(17))} \\
& [1, 2, 3, 4, 6, 8, 9, 12, 17, 18, 24, 34, 36, 51, 68, 72, 102, 136, 153, 204, 306, 408, 612, 1224]
\end{align*}
\]
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:

```python
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)  # needs sage.libs.pari
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 \texttt{Factorization}.

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

**Warning:** This means that applying \texttt{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:**

```python
sage: f(n) = n^2
# needs sage.symbolic
sage: is_prime(f(3))
# needs sage.symbolic
False
sage: factor(f(3))
# needs sage.symbolic
9
```

```python
>>> from sage.all import *
>>> __tmp__=var("n"); f = symbolic_expression(n**Integer(2)).function(n)# needs sage.symbolic
>>> is_prime(f(Integer(3)))
# needs sage.symbolic
False
>>> factor(f(Integer(3)))
# needs sage.symbolic
9
```

**INPUT:**

- \( n \) – a nonzero integer
- \texttt{proof} – bool or None (default: None)
- \( \texttt{int\_} \) – bool (default: False) whether to return answers as Python ints
- \texttt{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)
- \texttt{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 \texttt{qsieve} and \texttt{ecm} commands give access to highly optimized implementations of algorithms for doing certain integer factorization problems. These implementations are not used by the generic \texttt{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 \texttt{Factorization}; use \texttt{Factorization??} to see more details, and examples below for usage. A \texttt{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 \((\text{prime}, \text{exponent})\) pairs and the unit, to recover the number from its factorization, and even to multiply two factorizations. See examples below.

**EXAMPLES:**

```python
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)]
```

```python
sage: f.unit()
-1
sage: f.value()
-20
```

```python
sage: factor(-next_prime(10^2) * next_prime(10^7))
# needs sage.libs.pari
-1 * 101 * 10000019
```

```python
>>> from sage.all import *
>>> factor(Integer(500))
2^2 * 5^3
>>> factor(-Integer(20))
-1 * 2^2 * 5
>>> f=factor(-Integer(20))
>>> list(f)
[(2, 2), (5, 1)]
```

```python
>>> f.unit()
-1
>>> f.value()
-20
>>> factor(-next_prime(Integer(10)**Integer(2)) * next_prime(Integer(10)**Integer(7)))
# needs sage.libs.pari
-1 * 101 * 10000019
```

```python
sage: factor(293292629867846432923017396246429, algorithm='flint')
# needs sage.libs.flint
3 * 4852301647696687 * 20148007492971089
```

```python
>>> from sage.all import *
```
>>> from sage.all import *
>>> factor(-Integer(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
sage: factor(-1)
-1
sage: factor(2^(2^7)+1)
˓→needs sage.libs.pari
59649589127497217 * 5704689200685129054721

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

sage: factor(3^89 - 1, proof=False)  #...
˓→needs sage.libs.pari
2 * 179 * 1611479891519807 * 5042939439565996049162197

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

sage: factor(Integer(3)**Integer(89) - Integer(1), proof=False)  #...
˓→needs sage.libs.pari
2 * 179 * 1611479891519807 * 5042939439565996049162197

sage: factor(2**197 + 1)  # long time (2s)
˓→needs sage.libs.pari
3 * 197002597249 * 1348959352853811313 * 251951573867253012259144010843

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

sage: K.<i> = QuadraticField(-1)  #...
˓→needs sage.rings.number_field

(continues on next page)
sage: factor(122 - 454*i)  # needs sage.rings.number_field
(-i) * (-i - 2)^3 * (i + 1)^3 * (-2*i + 3) * (i + 4)

>>> from sage.all import *
>>> K = QuadraticField(-Integer(1), names=('i',)); (i,) = K._first_ngens(1)  # needs sage.rings.number_field
>>> factor(Integer(122) - Integer(454)*i)  # needs sage.rings.number_field
(-i) * (-i - 2)^3 * (i + 1)^3 * (-2*i + 3) * (i + 4)

To access the data in a factorization:

sage: # needs sage.libs.pari
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]

>>> from sage.all import *
>>> factor(math.pi)
3.141592653589793

>>> import numpy  # needs numpy
sage: factor(numpy.int8(30))  # needs numpy sage.libs.pari
2 * 3 * 5

We can factor Python, numpy and gmpy2 numbers:

sage: factor(math.pi)
3.141592653589793

sage: import numpy  # needs numpy
sage: factor(numpy.int8(30))  # needs numpy sage.libs.pari
2 * 3 * 5

sage: import gmpy2
sage: factor(gmpy2.mpz(30))
2 * 3 * 5

>>> from sage.all import *
>>> factor(math.pi)  # needs numpy
3.141592653589793

>>> import numpy  # needs numpy
sage.arith.misc.factorial\((n, \text{algorithm}=\text{\textsc{gmp}})\)

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

INPUT:

- \(n\) – an integer
- \text{algorithm} – string (default: \text{\textsc{gmp}}):
  - \text{\textsc{gmp}} – use the GMP C-library factorial function
  - \text{\textsc{pari}} – use PARI’s factorial function

OUTPUT: an integer

EXAMPLES:

```python
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:

```python
sage: from numpy import int8
# needs numpy
sage: factorial(int8(4))  # ...
24
sage: from gmpy2 import mpz
sage: factorial(mpz(4))
24
```

```python
>>> from sage.all import *
>>> from numpy import int8
# needs numpy
>>> factorial(int8(Integer(4))) # needs numpy
24
>>> from gmpy2 import mpz
```

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 $\ast \{ 1 \ldots n \}$. 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, 10)
3628800
sage: factorial(10)
3628800

sage: # needs sage.symbolic
sage: falling_factorial(10, RR('3.0'))
720.000000000000
sage: falling_factorial(10, RR('3.3'))
1310.11633396601
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

>>> from sage.all import *
>>> falling_factorial(Integer(10), Integer(3))
720
>>> falling_factorial(Integer(10), Integer(10))
3628800
>>> factorial(Integer(10))
3628800

>>> # needs sage.symbolic
>>> falling_factorial(Integer(10), RR('3.0'))
720.000000000000
>>> falling_factorial(Integer(10), RR('3.3'))
1310.11633396601
```python
>>> a = falling_factorial(Integer(1) + I, I); a
gamma(I + 2)
>>> CC(a)
0.652965496420167 + 0.343065839816545*I
>>> falling_factorial(Integer(1) + I, Integer(4))
4*I + 2
>>> falling_factorial(I, Integer(4))
-10

>>> M = MatrixSpace(ZZ, Integer(4), Integer(4)) # needs sage.modules
>>> A = M([[Integer(1), Integer(0), Integer(1), Integer(0)],
          [Integer(1), Integer(0), Integer(1), Integer(0)],
          [Integer(1), Integer(0), Integer(1), Integer(0)],
          [Integer(1), Integer(0), Integer(1), Integer(1)]]);
# needs sage.modules
>>> falling_factorial(A, Integer(2)) # A(A - I)
[ 1 0 10 10]
[ 1 0 10 10]
[ 20 0 101 100]
[ 2 0 11 10]

>>> x = ZZ['x'].gen(0)
>>> falling_factorial(x, Integer(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:

```
```python
>>> from sage.all import *
>>> four_squares(Integer(3))
(0, 1, 1, 1)
>>> four_squares(Integer(13))
(0, 0, 2, 3)
>>> four_squares(Integer(130))
(0, 0, 3, 11)
>>> four_squares(Integer(1101011011004))
(90, 102, 1220, 1049290)
>>> four_squares(Integer(10)**Integer(100) - Integer(1))
(155024616290, 2612183768627, 14142135623730950488016887,
  99999999999999999999999999999999999999999999999999)
```

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:

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

Tests with numpy and gmpy2 numbers:

```python
sage: from numpy import int8
# needs numpy
sage: fundamental_discriminant(int8(102))
408
sage: from gmpy2 import mpz
# needs gmpy2
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 $\chi$ (with values in a
ring $K$) is defined as

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

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 $F_p$.

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.all import *
sage: from numpy import int8
    # needs numpy
>>> fundamental_discriminant(int8(Integer(102)))
    # needs numpy
408
>>> from gmpy2 import mpz
>>> fundamental_discriminant(mpz(Integer(102)))
408
```
>>> from sage.all import *
>>> # needs sage.libs.pari sage.rings.number_field
>>> from sage.arith.misc import gauss_sum
>>> F = GF(Integer(5)); q = Integer(5)
>>> zq = UniversalCyclotomicField().zeta(q - Integer(1))
>>> L = [gauss_sum(zq**i, F) for i in range(Integer(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]
>>> [g*g.conjugate() for g in L]
[1, 5, 5, 5, 1]

# needs sage.libs.pari sage.rings.number_field

>>> F = GF(Integer(11)**Integer(2)); q = Integer(11)**Integer(2)
>>> zq = UniversalCyclotomicField().zeta(q - Integer(1))
>>> g = gauss_sum(zq**Integer(4), F)
>>> 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 \([\ldots]\). Before Issue #4988 the following wrongly returned 3 since the third parameter was just ignored:

```python
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:

```python
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:

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

```python
sage: from sage.all import *
```
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 ...>
```

```python
>>> from sage.all import *
```

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

```python
>>> from sage.all import *
```

```
sage: sage.arith.misc.get_inverse_mod(Integer(6000))
<built-in method inverse_mod_int of sage.rings.fast_arith.arith_int object at ...>
sage: sage.arith.misc.get_inverse_mod(Integer(600000))
<built-in method inverse_mod_longlong of sage.rings.fast_arith.arith_llong object at ...>
sage: sage.arith.misc.get_inverse_mod(Integer(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 \(\mathbb{Q}\).

INPUT:

- \(a, b\) – integers

OUTPUT:

squarefree positive integer

EXAMPLES:

```
sage: # needs sage.libs.pari
sage: hilbert_conductor(-1, -1)
```

(continues on next page)
sage: hilbert_conductor(-1, -11)
11
sage: hilbert_conductor(-2, -5)
5
sage: hilbert_conductor(-3, -17)
17

>>> from sage.all import *
>>> # needs sage.libs.pari
>>> hilbert_conductor(-Integer(1), -Integer(1))
2
>>> hilbert_conductor(-Integer(1), -Integer(11))
11
>>> hilbert_conductor(-Integer(2), -Integer(5))
5
>>> hilbert_conductor(-Integer(3), -Integer(17))
17

Tests with numpy and gmpy2 numbers:

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

>>> from sage.all import *
>>> from numpy import int8
# needs numpy
>>> hilbert_conductor(int8(-Integer(3)), int8(-Integer(17))) # needs numpy sage.libs.pari
17
>>> from gmpy2 import mpz
>>> hilbert_conductor(mpz(-Integer(3)), mpz(-Integer(17))) # needs sage.libs.pari
17

AUTHOR:

- Gonzalo Tornaria (2009-03-02)

sage.arith.misc.hilbert_conductor_inverse(d)
Finds a pair of integers \((a, b)\) such that \(\text{hilbert_conductor}(a, b) \equiv d\).

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

INPUT:

- \(d\) – square-free positive integer

OUTPUT: pair of integers

EXAMPLES:
sage: # needs sage.libs.pari
sage: hilbert_conductor_inverse(2)
(-1, -1)
sage: hilbert_conductor_inverse(3)
(-1, -3)
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

>>> from sage.all import *
>>> # needs sage.libs.pari
>>> hilbert_conductor_inverse(Integer(2))
(-1, -1)
>>> hilbert_conductor_inverse(Integer(3))
(-1, -3)
>>> hilbert_conductor_inverse(Integer(6))
(-1, 3)
>>> hilbert_conductor_inverse(Integer(30))
(-3, -10)
>>> hilbert_conductor_inverse(Integer(4))
Traceback (most recent call last):
  ... 
ValueError: d needs to be squarefree
>>> hilbert_conductor_inverse(-Integer(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:
1.13. Miscellaneous arithmetic functions

Tests with numpy and gmpy2 numbers:

```python
sage: from numpy import int8
        # needs numpy
sage: hilbert_symbol(int8(2), int8(3), int8(5), algorithm='all')  # needs numpy sage.libs.pari
1
sage: from gmpy2 import mpz
sage: hilbert_symbol(mpz(2), mpz(3), mpz(5), algorithm='all')  # needs sage.libs.pari
1
```
AUTHORS:

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

sage.arith.misc.integer_ceil(x)

Return the ceiling of x.

EXAMPLES:

```python
>>> import *
>>> from numpy import int8

# needs numpy
>>> hilbert_symbol(int8(Integer(2)), int8(Integer(3)), int8(Integer(5)),
algorithm='all')
1

# needs numpy sage.libs.pari
>>> from gmpy2 import mpz

>>> hilbert_symbol(mpz(Integer(2)), mpz(Integer(3)), mpz(Integer(5)), algorithm='all')

# needs sage.libs.pari
```

Tests with numpy and gmpy2 numbers:

```python
>>> from numpy import float32
# needs numpy
>>> integer_ceil(float32(5.4))
6

# needs sage.symbolic
Traceback (most recent call last):
...
NotImplementedError: computation of ceiling of x not implemented
```

```python
>>> from numpy import float32
# needs numpy
>>> integer_ceil(RealNumber('5.4'))
6

# needs sage.symbolic
Traceback (most recent call last):
...
NotImplementedError: computation of ceiling of x not implemented
```
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:

```python
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
```

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

Tests with numpy and gmpy2 numbers:

```python
sage: from numpy import float32 #...
sage: integer_floor(float32(5.4))
5
```

```python
sage: from gmpy2 import mpfr
sage: integer_floor(mpfr(5.4))
5
```
>>> from sage.all import *
>>> from numpy import float32
# needs numpy
>>> integer_floor(float32(RealNumber('5.4')))  # needs numpy
5

>>> from gmpy2 import mpfr
>>> integer_floor(mpfr(RealNumber('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

>>> from sage.all import *
>>> from sage.arith.misc import integer_trunc as trunc
>>> trunc(-Integer(3)/Integer(2)), trunc(-Integer(1)), trunc(-Integer(1)/
→Integer(2)), trunc(Integer(0)), trunc(Integer(1)/Integer(2)), trunc(Integer(1)),
→trunc(Integer(3)/Integer(2))
(-1, -1, 0, 0, 0, 1, 1)
>>> isinstance(trunc(Integer(3)/Integer(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

>>> from sage.all import *
>>> inverse_mod(Integer(7), Integer(1))
0
>>> inverse_mod(Integer(5), Integer(14))
3
>>> inverse_mod(Integer(3), -Integer(5))
2

Tests with numpy and mpz numbers:

sage: from numpy import int8
# needs numpy
(continues on next page)
sage: inverse_mod(int8(5), int8(14))  # needs numpy
3
sage: from gmpy2 import mpz
sage: inverse_mod(mpz(5), mpz(14))
3

>>> from sage.all import *
>>> from numpy import int8
# needs numpy
>>> inverse_mod(int8(Integer(5)), int8(Integer(14)))  
˓→ # needs numpy
3
>>> from gmpy2 import mpz
>>> inverse_mod(mpz(Integer(5)), mpz(Integer(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
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
˓→ needs numpy

(continues on next page)
sage.arith.misc.is_prime(n)

Determine whether $n$ is a prime element of its parent ring.

INPUT:

• $n$ – the object for which to determine primality

Exceptional special cases:

• For integers, determine whether $n$ is a positive prime.

• For number fields except $\mathbb{Q}$, determine whether $n$ is a prime element of the maximal order.

ALGORITHM:

For integers, this function uses a provable primality test or a strong pseudo-primality test depending on the global arithmetic proof flag.

See also:

• is_pseudoprime()

• sage.rings.integer.Integer.is_prime()

EXAMPLES:

sage: is_prime(389)
True
sage: is_prime(2000)
False
sage: is_prime(2)
True
The function `is_prime` checks if a given integer is a prime number.

```python
sage: is_prime(-1)
False
sage: is_prime(1)
False
sage: is_prime(-2)
False
>>> from sage.all import *
>>> is_prime(Integer(389))
True
>>> is_prime(Integer(2000))
False
>>> is_prime(Integer(2))
True
>>> is_prime(-Integer(1))
False
>>> is_prime(Integer(1))
False
>>> is_prime(-Integer(2))
False
```

The `is_prime_power` function checks if a given number is a positive power of a prime number.

```python
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)
```

The function `is_prime_power` tests whether a given number `n` is a positive power of a prime number. It 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:**

```python
sage: # needs sage.libs.pari
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)

>>> from sage.all import *
>>>

# needs sage.libs.pari

>>> is_prime_power(Integer(389))
True
>>> is_prime_power(Integer(2000))
False
>>> is_prime_power(Integer(2))
True
>>> is_prime_power(Integer(1024))
True
>>> is_prime_power(Integer(1024), get_data=True)
(2, 10)

The same results can be obtained with:

sage: # needs sage.libs.pari
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)

>>> from sage.all import *
>>> # needs sage.libs.pari

>>> Integer(389).is_prime_power()
True
>>> Integer(2000).is_prime_power()
False
>>> Integer(2).is_prime_power()
True
>>> Integer(1024).is_prime_power()
True
>>> Integer(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: # needs sage.libs.pari
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
```

```python
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> is_pseudoprime(Integer(389))
True
>>> is_pseudoprime(Integer(2000))
False
>>> is_pseudoprime(Integer(2))
True
>>> is_pseudoprime(-Integer(1))
False
>>> factor(-Integer(6))
-1 * 2 * 3
>>> is_pseudoprime(Integer(1))
False
>>> is_pseudoprime(-Integer(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
- \( \text{get\_data} \) – (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: # needs sage.libs.pari
sage: is_pseudoprime_power(389)
True
sage: is_pseudoprime_power(2000)
False
```

(continues on next page)
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  
# needs numpy  
sage: is_pseudoprime_power(int16(1024))  
#...  

(continues on next page)
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:

```python
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))
# needs sage.rings.complex_double
True
sage: is_square((x-1)^2)
# needs sage.symbolic
Traceback (most recent call last):
...
NotImplementedError: is_square() not implemented for non-constant or relational elements of Symbolic Ring
```
>>> is_square(RealNumber('2.2'))
True
>>> is_square(-RealNumber('2.2'))
False
>>> is_square(CDF(-RealNumber('2.2')))  # needs sage.rings.complex_double
True
>>> is_square((x-Integer(1))**Integer(2))  # needs sage.symbolic
Traceback (most recent call last):
  ... Not Implemented Error: is_square() not implemented for non-constant or relational elements of Symbolic Ring

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

>>> from sage.all import *
>>> is_square(Integer(4), True)
(True, 2)

Tests with numpy and gmpy2 numbers:

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

>>> from sage.all import *
>>> from numpy import int8  # needs numpy
>>> is_square(int8(Integer(4)))  # needs numpy
True
>>> from gmpy2 import mpz
>>> is_square(mpz(Integer(4)))
True

Tests with Polynomial:

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

>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, 'v', names=('v',)); (v,) = R._first_ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2))  # needs sage.combinat sage.modules
sage.arith.misc.is_squarefree(n)

Test whether \( n \) is square free.

EXAMPLES:

```python
sage: is_squarefree(100) # needs sage.libs.pari
False
sage: is_squarefree(101) # needs sage.libs.pari
True
sage: R = ZZ['x']
sage: x = R.gen()
sage: is_squarefree((x^2+x+1) * (x-2)) # needs sage.libs.pari
True
sage: is_squarefree((x-1)**2 * (x-3)) # needs sage.libs.pari
False
sage: # needs sage.rings.number_field sage.symbolic
sage: O = ZZ[sqrt(-1)]
sage: I = O.gen(1)
sage: is_squarefree(I + 1) # needs sage.libs.pari
True
sage: is_squarefree(O(2)) False
sage: O(2).factor()
(-I) * (I + 1)^2
```

```python
>>> from sage.all import *
>>> is_squarefree(Integer(100)) # needs sage.libs.pari
False
>>> is_squarefree(Integer(101)) # needs sage.libs.pari
True
>>> R = ZZ['x']
>>> x = R.gen()
>>> is_squarefree((x**Integer(2)+x+Integer(1)) * (x-Integer(2))) # needs sage.libs.pari
True
>>> is_squarefree((x-Integer(1))**2 * (x-Integer(3))) # needs sage.libs.pari
False
>>> # needs sage.rings.number_field sage.symbolic
>>> O = ZZ[sqrt(-Integer(1))]
(continues on next page)"
This method fails on domains which are not Unique Factorization Domains:

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

Tests with numpy and gmpy2 numbers:

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

(continues on next page)
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
\[ b = p_1^{e_1} \times \ldots \times p_r^{e_r} \]
then
\[ (a|b) = (a|p_1)^{e_1}(a|p_2)^{e_2} \ldots (a|p_r)^{e_r} \]
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

>>> from sage.all import *
>>> jacobi_symbol(Integer(10), Integer(777))
-1
>>> jacobi_symbol(Integer(10), Integer(5))
0
>>> jacobi_symbol(Integer(10), Integer(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

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

>>> from sage.all import *
>>> from numpy import int16  # needs numpy

>>> jacobi_symbol(int16(Integer(10)), int16(Integer(777)))  # needs numpy
-1

>>> from gmpy2 import mpz

>>> jacobi_symbol(mpz(Integer(10)),mpz(Integer(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

>>> from sage.all import *

>>> kronecker(Integer(13), Integer(21))
-1

>>> kronecker(Integer(101), Integer(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

>> from sage.all import *

>> kronecker(Integer(3), Integer(5))
-1

>> kronecker(Integer(3), Integer(15))
Tests with numpy and gmpy2 numbers:

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

```python
>>> from sage.all import *
>>> from numpy import int8
# → needs numpy
>>> kronecker_symbol(Integer(13), Integer(21))
# → needs numpy
-1
>>> from gmpy2 import mpz
>>> kronecker_symbol(mpz(Integer(13)), mpz(Integer(21)))
-1
```

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

The Kronecker symbol $(x|y)$.

**INPUT:**

- $x$ – integer
- $y$ – integer

**OUTPUT:**

- an integer

**EXAMPLES:**

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

```python
>>> from sage.all import *
>>> kronecker_symbol(Integer(13), Integer(21))
-1
>>> kronecker_symbol(Integer(101), Integer(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

>>> from sage.all import *
>>> kronecker(Integer(3),Integer(5))
-1
>>> kronecker(Integer(3),Integer(15))
0
>>> kronecker(Integer(2),Integer(15))
1
>>> kronecker(-Integer(2),Integer(15))
-1
>>> kronecker(Integer(2)/Integer(3),Integer(5))
1

Tests with numpy and gmpy2 numbers:

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

>>> from sage.all import *
>>> from numpy import int8
# needs numpy
>>> kronecker_symbol(int8(Integer(13)),int8(Integer(21)))
# needs numpy
-1
>>> from gmpy2 import mpz
>>> kronecker_symbol(mpz(Integer(13)),mpz(Integer(21)))
-1

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

Note: The \texttt{kronecker_symbol()} command extends the Legendre symbol to composite moduli and \(p = 2\).

INPUT:

- \(x\) – integer
- \(p\) – an odd prime number

EXAMPLES:
Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8  # needs numpy
sage: legendre_symbol(int8(2), int8(3))  # needs numpy
-1
```

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

```
>>> from sage.all import *
>>> from numpy import int8  # needs numpy
>>> legendre_symbol(int8(Integer(2)), int8(Integer(3)))  # needs numpy
-1
```

```
>>> from gmpy2 import mpz
>>> legendre_symbol(mpz(Integer(2)), mpz(Integer(3)))
-1
```

```
sage.arith.misc.mqrr_rational_reconstruction(u, m, T)
```

1.13. Miscellaneous arithmetic functions 207
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 \( d > 0, \gcd(n, d) = 1, n/d = u \mod m \), 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:
```python
sage: mqrr_rational_reconstruction(21, 3100, 13)
(21, 1)
```
```
>>> from sage.all import *
>>> mqrr_rational_reconstruction(Integer(21), Integer(3100), Integer(13))
(21, 1)
```
Tests with numpy and gmpy2 numbers:
```python
sage: from numpy import int16
sage: mqrr_rational_reconstruction(int16(21), int16(3100), int16(13))
(21, 1)
```
```python
>>> from gmpy2 import mpz
>>> mqrr_rational_reconstruction(mpz(21), mpz(3100), mpz(13))
(21, 1)
```
```
>>> from sage.all import *
>>> from numpy import int16
>>> mqrr_rational_reconstruction(int16(Integer(21)), int16(Integer(3100)), int16(Integer(13)))
# needs numpy
(21, 1)
```
```python
>>> from gmpy2 import mpz
>>> mqrr_rational_reconstruction(mpz(Integer(21)), mpz(Integer(3100)), mpz(Integer(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:

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

EXAMPLES:

```python
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]))
# needs sage.combinat
```

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

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

INPUT:

- \(m\) – integer
- \(n\) – integer

OUTPUT: dict

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

\[
\begin{align*}
\text{sage: } & \text{sorted(multinomial_coefficients(2, 5).items())} \\
& \text{[((0, 5), 1), ((1, 4), 5), ((2, 3), 10), ((3, 2), 10), ((4, 1), 5), ((5, 0), 1)]}
\end{align*}
\]

```
from sage.all import *
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
```

Noticethatthesearethecoefficientsof\((x + y)^5\):
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: # needs sage.libs.pari
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
```

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
>>> from sage.all import *
```  

```
```
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 \texttt{Integer.next_prime_power()} of \texttt{Integers}.

See also:

• \texttt{is_prime_power()} (and \texttt{Integer.is_prime_power()})

• \texttt{previous_prime_power()} (and \texttt{Integer.previous_prime_power()})

EXAMPLES:

```
sage: # needs sage.libs.pari
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
```

```
>>> from sage.all import * 
>>> # needs sage.libs.pari
>>> next_prime_power(Integer(1))  
2
>>> next_prime_power(Integer(2))  
3
>>> next_prime_power(Integer(10)) 
11
>>> next_prime_power(Integer(7))  
8
>>> next_prime_power(Integer(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
```

```
>>> from sage.all import * 
>>> Integer(1).next_prime_power()  
2
>>> Integer(2).next_prime_power()  
```
Note that 2 is the smallest prime power:

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

```python
>>> from sage.all import *

```

```python
>>>
```

sage.arith.misc.next_probable_prime(n)

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

INPUT:

- n – an integer

EXAMPLES:

```python
sage: # needs sage.libs.pari
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)
155251809230070935148979488462502555256860171166966111390520380260509526863768863087840882864
```

```python
>>> from sage.all import *

```

```python
>>> # needs sage.libs.pari

```

```python
>>> next_probable_prime(-Integer(100))
2
>>> next_probable_prime(Integer(19))
23
>>> next_probable_prime(int(Integer(999999999)))
1000000007
>>> next_probable_prime(Integer(2)**Integer(768))
155251809230070935148979488462502555256860171166966111390520380260509526863768863087840882864
```

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)  # needs sage.libs.pari
5
sage: nth_prime(10)  # needs sage.libs.pari
29
sage: nth_prime(10^7)  # needs sage.libs.pari
179424673

>>> from sage.all import *
>>> nth_prime(Integer(3))  # needs sage.libs.pari
5
>>> nth_prime(Integer(10))  # needs sage.libs.pari
29
>>> nth_prime(Integer(10)**Integer(7))  # needs sage.libs.pari
179424673

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

>>> from sage.all import *
>>> nth_prime(Integer(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)  # needs sage.libs.pari
9
sage: number_of_divisors(-720)  # needs sage.libs.pari
30

>>> from sage.all import *
>>> number_of_divisors(Integer(100))  # needs sage.libs.pari
9
>>> number_of_divisors(-Integer(720))  # needs sage.libs.pari
(continues on next page)
Tests with numpy and gmpy2 numbers:

```python
sage: from numpy import int8
      # needs numpy
sage: number_of_divisors(int8(100))
      # needs numpy sage.libs.pari
9
sage: from gmpy2 import mpz
sage: number_of_divisors(mpz(100))
      # needs sage.libs.pari
9
```

```python
>>> from sage.all import *

>>> from numpy import int8
      # needs numpy
>>> number_of_divisors(int8(Integer(100)))
      # needs numpy sage.libs.pari
9
```

```python
>>> from gmpy2 import mpz

>>> number_of_divisors(mpz(Integer(100)))
      # needs sage.libs.pari
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:

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

```python
>>> from sage.all import *

>>> odd_part(Integer(5))
5
```

```python
>>> odd_part(Integer(4))
1
```

```python
>>> odd_part(factorial(Integer(31)))
122529844256906551386796875
```

Tests with numpy and gmpy2 numbers:

```python
sage: from numpy import int8
      # needs numpy
sage: odd_part(int8(5))
      # needs numpy
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:

\[
\begin{align*}
\text{sage: } & \quad \text{power\_mod}(2, 388, 389) \\
& 1 \\
\text{sage: } & \quad \text{power\_mod}(2, 390, 391) \\
& 285 \\
\text{sage: } & \quad \text{power\_mod}(2, -1, 7) \\
& 4 \\
\text{sage: } & \quad \text{power\_mod}(11, 1, 7) \\
& 4 \\
\end{align*}
\]

This function works for fairly general rings:

\[
\begin{align*}
\text{sage: } & \quad R.<x> = \ZZ[] \\
\text{sage: } & \quad \text{power\_mod}(3*x, 10, 7) \\
& 4*x^10 \\
\text{sage: } & \quad \text{power\_mod}(-3*x^2 + 4, 7, 2*x^3 - 5) \\
& x^14 + x^8 + x^6 + x^3 + 962509*x^2 - 791910*x - 698281 \\
\end{align*}
\]

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

The largest prime < \(n\). The result is provably correct. If \(n \leq 1\), this function raises a ValueError.

EXAMPLES:
sage: # needs sage.libs.pari
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

>>> from sage.all import *

>>> # needs sage.libs.pari

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

1.13. Miscellaneous arithmetic functions 217
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:

```python
sage: # needs sage.libs.pari
sage: previous_prime_power(3)
2
sage: previous_prime_power(10)
9
sage: previous_prime_power(7)
5
sage: previous_prime_power(127)
125
```

```python
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> previous_prime_power(Integer(3))
2
>>> previous_prime_power(Integer(10))
9
>>> previous_prime_power(Integer(7))
5
>>> previous_prime_power(Integer(127))
125
```

The same results can be obtained with:

```python
sage: # needs sage.libs.pari
sage: 3.previous_prime_power()
2
sage: 10.previous_prime_power()
9
sage: 7.previous_prime_power()
5
sage: 127.previous_prime_power()
125
```

```python
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> Integer(3).previous_prime_power()
2
>>> Integer(10).previous_prime_power()
9
>>> Integer(7).previous_prime_power()
5
>>> Integer(127).previous_prime_power()
125
```

Input less than or equal to 2 raises errors:

```python
sage: previous_prime_power(2)
Traceback (most recent call last):
```

(continues on next page)
ValueError: no prime power less than 2
\[
\text{sage: previous_prime_power(-10)}
\]
Traceback (most recent call last):
...
ValueError: no prime power less than 2

>>> from sage.all import *

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

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

sage: n = previous_prime_power(2^16 - 1)  # needs sage.libs.pari

sage: while is_prime(n):
    # needs sage.libs.pari
    ....: n = previous_prime_power(n)

sage: factor(n)  # needs sage.libs.pari
251^2

>>> from sage.all import *

>>> n = previous_prime_power(Integer(2)**Integer(16) - Integer(1))  # needs sage.libs.pari

>>> while is_prime(n):
    # needs sage.libs.pari
    ....: n = previous_prime_power(n)

>>> factor(n)  # needs sage.libs.pari
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:

\[
\text{sage: prime_divisors(1)}
\]
\[
[1]
\]
\[
\text{sage: prime_divisors(100)}
\]
\[
[2, 5]
\]
\[
\text{sage: prime_divisors(2004)}
\]
\[
[2, 3, 167]
\]
If \( n \) is negative, we do not include \(-1\) among the prime divisors, since \(-1\) is not a prime number:

```
>> prime_divisors(Integer(-100))
[2, 5]
```

For polynomials we get all irreducible factors:

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

Tests with numpy and gmpy2 numbers:

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

\[\text{sage.arith.misc.prime_factors}(n)\]

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

```python
>>> from sage.all import *
>>> prime_divisors(Integer(1))
[]
>>> prime_divisors(Integer(100))
[2, 5]
>>> prime_divisors(Integer(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
sage: prime_divisors(-100)
[2, 5]
```

```python
>>> from sage.all import *
>>> prime_divisors(-Integer(100))
[2, 5]
```

For polynomials we get all irreducible factors:

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

```python
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> prime_divisors(x**Integer(12) - Integer(1))  # needs sage.libs.pari
[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
sage: from numpy import int8
# needs numpy
sage: prime_divisors(int8(-100))
[2, 5]
```

```sage
sage: from gmpy2 import mpz
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:

```python
sage: prime_powers(20)  # needs sage.libs.pari
[2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17, 19]
sage: len(prime_powers(1000))  # needs sage.libs.pari
193
sage: len(prime_range(1000))  # needs sage.libs.pari
168
sage: # needs sage.libs.pari
sage: a = [z for z in range(95, 1234) if is_prime_power(z)]
sage: b = prime_powers(95, 1234)
sage: len(a)
194
sage: len(b)
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)]  # needs sage.libs.pari
True
sage: prime_powers(10, 7)
[]
```
sage: prime_powers(-5)
[]
sage: prime_powers(-1, 3)  # needs sage.libs.pari
[2]

>>> from sage.all import *

>>> prime_powers(Integer(20))  # needs sage.libs.pari
[2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17, 19]

>>> len(prime_powers(Integer(1000)))  # needs sage.libs.pari
193

>>> len(prime_range(Integer(1000)))  # needs sage.libs.pari
168

>>> len(prime_powers(Integer(100))) == [i for i in range(Integer(100)) if is_prime_power(i)]  # needs sage.libs.pari
True

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

>>> from sage.all import *

>>> prime_to_m_part(Integer(240),Integer(2))
15
>>> prime_to_m_part(Integer(240),Integer(3))
80
>>> prime_to_m_part(Integer(240),Integer(5))
48
>>> prime_to_m_part(Integer(43434),Integer(20))
21717

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

sage: 240.prime_to_m_part(2)
15

>>> from sage.all import *

>>> Integer(240).prime_to_m_part(Integer(2))
15

Tests with numpy and gmpy2 numbers:

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

>>> from sage.all import *

>>> from numpy import int16  # needs numpy
>>> prime_to_m_part(int16(Integer(240)), int16(Integer(2)))  # needs numpy
15
>>> from gmpy2 import mpz
>>> prime_to_m_part(mpz(Integer(240)), mpz(Integer(2)))
15

sage.arith.misc.primes (start=2, 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 (default: 2) lower bound for the primes
- `stop` – an integer (or infinity) 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:

```python
code
```
```python
code
```
```python
code
```
```python
code
```
```python
code
```
```python
code
```
```python
>>> from sage.all import *

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)  # needs sage.libs.pari
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
sage: len(primes_first_n(1000))  # needs sage.libs.pari
1000
sage: primes_first_n(0)
[]

>>> from sage.all import *
>>> primes_first_n(Integer(10))   # needs sage.libs.pari
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
>>> len(primes_first_n(Integer(1000)))   # needs sage.libs.pari
1000
>>> primes_first_n(Integer(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: # needs sage.libs.pari
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)])
[1, 2, 2, 3, 2, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 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

>>> from sage.all import *
>>> # needs sage.libs.pari
>>> primitive_root(Integer(23))
5
>>> primitive_root(-Integer(46))
5
>>> primitive_root(Integer(25))
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:

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

```
>>> from sage.all import *

>>> n = Integer(10)**Integer(50) + Integer(151)  # a prime
>>> primitive_root(n)  # needs sage.libs.pari
11
>>> primitive_root(n, check=False)  # needs sage.libs.pari
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:**

```python
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
```

```python
>>> from sage.all import *

>>> quadratic_residues(Integer(11))
[0, 1, 3, 4, 5, 9]
>>> quadratic_residues(Integer(1))
[0]
>>> quadratic_residues(Integer(2))
[0, 1]
```


```python
>>> quadratic_residues(Integer(8))
[0, 1, 4]
>>> quadratic_residues(-Integer(10))
[0, 1, 4, 5, 6, 9]
>>> v = quadratic_residues(Integer(1000)); len(v)
159
```

Tests with numpy and gmpy2 numbers:

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

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

```python
>>> from sage.all import *
>>> from numpy import int8  # needs numpy  
>>> quadratic_residues(int8(Integer(11)))  # needs numpy  
[0, 1, 3, 4, 5, 9]
```

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

Return the product of the prime divisors of \( n \).

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

EXAMPLES:

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

```python
sage: radical(0)
Traceback (most recent call last):
  ...  
ArithmeticError: radical of 0 is not defined
sage: K.<i> = QuadraticField(-1)  # needs sage.rings.number_field
```

```python
sage: radical(K(2))  # needs sage.rings.number_field
i + 1
```

```python
>>> from sage.all import *
>>> radical(Integer(2) * Integer(3)**Integer(2) * Integer(5)**Integer(5))  
30
```

```python
>>> radical(Integer(0))
Traceback (most recent call last):
  ...  
ArithmeticError: radical of 0 is not defined
```

```python
>>> K = QuadraticField(-Integer(1), names=('i',)); (i,) = K._first_ngens(1)  # needs sage.rings.number_field
```

(continues on next page)
standard commutative rings, release 10.4

(continued from previous page)

```python
>>> radical(K(Integer(2)))
˓→ # needs sage.rings.number_field
i + 1
```

Tests with numpy and gmpy2 numbers:

```python
sage: from numpy import int8 #...
˓→ # needs numpy
sage: radical(int8(50)) #...
˓→ # needs numpy
10
sage: from gmpy2 import mpz
sage: radical(mpz(50))
10
```

```python
>>> from sage.all import *
>>> from numpy import int8 #...
˓→ # needs numpy
```

```python
>>> from gmpy2 import mpz
>>> radical(mpz(Integer(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 \( \geq 2 \).
- \( proof \) – 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 \( \geq 2 \), lower bound for the chosen primes

**EXAMPLES:**

```python
sage: # needs sage.libs.pari
sage: p = random_prime(100000)
sage: p.is_prime()
True
sage: p <= 100000
True
sage: random_prime(2)
2
```

```python
>>> from sage.all import *
>>> # needs sage.libs.pari
```

```python
>>> p = random_prime(Integer(100000))
>>> p.is_prime()
True
```

(continues on next page)
Here we generate a random prime between 100 and 200:

```python
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`

```python
sage: p = random_prime(200, proof=False, lbound=100)  # needs sage.libs.pari
sage: p.is_pseudoprime()  # needs sage.libs.pari
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 $v^3 \leq \sqrt{m}/2$, the algorithm terminates. The pair $(x, y) = (|v^2|, v^3 \ast \text{sign}(v^2))$ 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:

```python
sage: m = 100000
sage: (119*inverse_mod(53,m))%m
11323
sage: rational_reconstruction(11323,m)
119/53
```

```python
>>> from sage.all import *

>>> m = Integer(100000)

>>> (Integer(119)*inverse_mod(Integer(53),m))%m
11323

>>> rational_reconstruction(Integer(11323),m)
119/53
```

```python
sage: rational_reconstruction(400,1000)
Traceback (most recent call last):
...
ArithmeticError: rational reconstruction of 400 (mod 1000) does not exist
```

```python
>>> from sage.all import *

>>> rational_reconstruction(Integer(400),Integer(1000))
Traceback (most recent call last):
...
ArithmeticError: rational reconstruction of 400 (mod 1000) does not exist
```

```python
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)
```

(continues on next page)
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'

>>> from sage.all import *
>>> rational_reconstruction(Integer(3), Integer(292393))
3
>>> a = Integers(Integer(292393))(Integer(45)/Integer(97)); a
204977
>>> rational_reconstruction(a, Integer(292393), algorithm='fast')
45/97
>>> rational_reconstruction(Integer(293048), Integer(292393))
Traceback (most recent call last):
  ...
ArithmeticError: rational reconstruction of 655 (mod 292393) does not exist
>>> rational_reconstruction(Integer(0), Integer(0))
Traceback (most recent call last):
  ...
ZeroDivisionError: rational reconstruction with zero modulus
>>> rational_reconstruction(Integer(0), Integer(1), algorithm='foobar')
Traceback (most recent call last):
  ...
ValueError: unknown algorithm 'foobar'

Tests with numpy and gmpy2 numbers:

sage: from numpy import int32
   # needs numpy
sage: rational_reconstruction(int32(3), int32(292393))
   # needs numpy
3
sage: from gmpy2 import mpz
sage: rational_reconstruction(mpz(3), mpz(292393))
3

>>> from sage.all import *
>>> from numpy import int32
   # needs numpy
>>> rational_reconstruction(int32(Integer(3)), int32(Integer(292393)))
   # needs numpy
3
>>> from gmpy2 import mpz
>>> rational_reconstruction(mpz(Integer(3)), mpz(Integer(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 \(x_\uparrow^a\).

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:**

```python
sage: rising_factorial(10, 3)
1320

sage: # needs sage.symbolic
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
```

```python
>>> from sage.all import *

>>> rising_factorial(Integer(10),Integer(3))
1320

>>> # needs sage.symbolic
>>> rising_factorial(Integer(10), RR('3.0'))
1320.00000000000

>>> rising_factorial(Integer(10), RR('3.3'))
2826.38895824964

>>> a = rising_factorial(Integer(1)+I, I); a
gamma(2*I + 1)/gamma(I + 1)

>>> CC(a)
0.266816390637832 + 0.122783354006372*I

>>> a = rising_factorial(I, Integer(4)); a
-10

>>> x = polygen(ZZ)

>>> rising_factorial(x, Integer(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:

```python
sage: # needs sage.rings.complex_double
sage: import sage.arith.misc
sage: sort_c = sort_complex_numbers_for_display
sage: nums = [CDF(i) for i in range(3)]
for i in range(3):
    nums.append(CDF(i + RDF.random_element(-3e-11, 3e-11),
    RDF.random_element()))
for i in range(len(nums)):
    if nums[i].imag():
        first_non_real = i
        break
    else:
        first_non_real = len(nums)
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()
```

```python
>>> from sage.all import *
>>> # needs sage.rings.complex_double
>>> import sage.arith.misc
>>> sort_c = sort_complex_numbers_for_display
>>> nums = [CDF(i) for i in range(Integer(3))]
>>> for i in range(Integer(3)):
...    nums.append(CDF(i + RDF.random_element(-RealNumber('3e-11'), RealNumber('3e-11')),
...    RDF.random_element()))
...    nums.append(CDF(i + RDF.random_element(-RealNumber('3e-11'), RealNumber('3e-11')),
...    RDF.random_element()))
```

(continues on next page)
>>> shuffle(nums)
>>> nums = sort_c(nums)
>>> for i in range(len(nums)):
...   if nums[i].imag():
...     first_non_real = i
...     break
... else:
...   first_non_real = len(nums)
>>> assert first_non_real >= Integer(3)
>>> for i in range(first_non_real - Integer(1)):
...   assert nums[i].real() <= nums[i + Integer(1)].real()

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
totaltime = 0.000003s
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]
```

```sage
>>> from sage.all import *
```

```sage
>>> list(squarefree_divisors(Integer(7)))
[1, 7]
```

```sage
>>> list(squarefree_divisors(Integer(6)))
[1, 2, 3, 6]
```

```sage
>>> list(squarefree_divisors(Integer(12)))
[1, 2, 3, 6]
```

Squarefree divisors are not yielded in increasing order:

```sage
totaltime = 0.000003s
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:**

```python
sage: subfactorial(0)
1
sage: subfactorial(1)
0
sage: subfactorial(8)
14833
```

Tests with numpy and gmpy2 numbers:

```python
sage: from numpy import int8
sage: subfactorial(int8(8))  # needs numpy
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 \texttt{ValueError}. 

236 Chapter 1. Integers
INPUT:
- $k$ – a non-negative integer
- $n$ – an integer

OUTPUT: a tuple $(x_1, \ldots, x_k)$ 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)
# needs sage.libs.pari
(19215400822645944253860920437586326284, 37204645194585992174252915693267578306, 3473654819477394665857484221256136567800161086815834297092488779216863122, 586019179961767363354757261035179796721850737768032876360978911074629287841061578270832330322, 2045742329455818249400191981237902399253880220373079101972854343976534785131636653709469689666, 311628095411678158492377386194583964975346960435809122253342693716118369103459303207008166496)
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, 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
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
```

1.13. Miscellaneous arithmetic functions
```python
>>> from sage.all import *

>>> sum_of_k_squares(Integer(2), Integer(9634))
(15, 97)

>>> sum_of_k_squares(Integer(3), Integer(9634))
(0, 15, 97)

>>> sum_of_k_squares(Integer(4), Integer(9634))
(1, 2, 5, 98)

>>> sum_of_k_squares(Integer(5), Integer(9634))
(0, 1, 2, 5, 98)

>>> sum_of_k_squares(Integer(6), Integer(11)**Integer(1111) - Integer(1))
(19215400822645944253860920437586326284, 37204645194585992174252915693267578306,
3473654819477394665857484221256136567800161086815834297092488779216863122,
...
5860191799617673633547572610351797996721850737768032876360978911074629287841061598270832330322...
...
204574232945581824940019198123790239925388022037307910197285434397653478513163665370946968966...
...
3116280954116781598492377386194583964975346960435809122253342693716118369103459303207008166496...

>>> sum_of_k_squares(Integer(7), Integer(0))
(0, 0, 0, 0, 0, 0, 0)

>>> sum_of_k_squares(Integer(30), Integer(999999))
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 3, 7, 44, 999)

>>> sum_of_k_squares(Integer(1), Integer(9))
(3,)

>>> sum_of_k_squares(Integer(1), Integer(10))
Traceback (most recent call last):
  ... ValueError: 10 is not a sum of 1 square

>>> sum_of_k_squares(Integer(1), -Integer(10))
Traceback (most recent call last):
  ... ValueError: -10 is not a sum of 1 square

>>> sum_of_k_squares(Integer(0), Integer(9))
Traceback (most recent call last):
  ... ValueError: 9 is not a sum of 0 squares

>>> sum_of_k_squares(Integer(0), Integer(0))
()

>>> sum_of_k_squares(Integer(7), -Integer(1))
Traceback (most recent call last):
  ... ValueError: -1 is not a sum of 7 squares

>>> sum_of_k_squares(-Integer(1), Integer(0))
Traceback (most recent call last):
  ... ValueError: k = -1 must be non-negative
```

Tests with numpy and gmpy2 numbers:

```python
sage: from numpy import int16
  # needs numpy

sage: sum_of_k_squares(int16(2), int16(9634))
  # needs numpy
```

(continues on next page)
sage: from gmpy2 import mpz
sage: sum_of_k_squares(mpz(2), mpz(9634))

>>> from sage.all import *
>>> from numpy import int16  # needs numpy

>>> sum_of_k_squares(int16(Integer(2)), int16(Integer(9634))) → # needs numpy

>>> from gmpy2 import mpz

>>> sum_of_k_squares(mpz(Integer(2)), mpz(Integer(9634)))

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, 17984650426474121466620280340569649349251249)

sage: three_squares(11^111 - 1)  
# needs sage.libs.pari
(616274160655975340150706642680, 90158293838573514329506746161, 62703823876357414039400136306531196796409981788593947233)

sage: three_squares(7 * 2^41)  
# needs sage.libs.pari
(1048576, 2097152, 3145728)

sage: three_squares(7 * 2^42)
Traceback (most recent call last):
...
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

>>> from sage.all import *

>>> three_squares(Integer(389))  
(1, 8, 18)

>>> three_squares(Integer(946))  
(9, 9, 28)
```python
>>> three_squares(Integer(2986))
(3, 24, 49)
>>> three_squares(Integer(7)**Integer(100))
(0, 0, 1798465042647146620280340569649349251249)
>>> three_squares(Integer(11)**Integer(111) - Integer(1))
  # needs sage.libs.pari
(61627416065597534150706442680, 901582938385735143295060746161,
   62703823876357441403940136306531196796409981788593947233)
>>> three_squares(Integer(7) * Integer(2)**Integer(41))
  # needs sage.libs.pari
(61627416065597534150706442680, 901582938385735143295060746161,
   62703823876357441403940136306531196796409981788593947233)
Traceback (most recent call last):
  ... ValueError: 30786325577728 is not a sum of 3 squares
>>> three_squares(Integer(0))
(0, 0, 0)
>>> three_squares(-Integer(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:

```
Tests with numpy and gmpy2 numbers:

```python
sage: from numpy import int8
# needs numpy
sage: trial_division(int8(91))  #...
4
sage: from gmpy2 import mpz
sage: trial_division(mpz(91))  
7
```
needs sage.libs.pari
(253801659504708621991421712450521, 2583712713213354898490304645018692)
sage: two_squares(0)
(0, 0)
sage: two_squares(-1)
Traceback (most recent call last):
  ...
ValueError: -1 is not a sum of 2 squares

>>> from sage.all import *

>>> two_squares(Integer(389))
(10, 17)

>>> two_squares(Integer(21))
Traceback (most recent call last):
  ...
ValueError: 21 is not a sum of 2 squares

>>> two_squares(Integer(21)**Integer(2))
(0, 21)

>>> a, b = two_squares(Integer(100000000000000000129)); a, b
# needs sage.libs.pari
(4418521500, 8970878873)

>>> a**Integer(2) + b**Integer(2)
# needs sage.libs.pari
100000000000000000129

>>> two_squares(Integer(2)**Integer(222) + Integer(1))
# needs sage.libs.pari
(253801659504708621991421712450521, 2583712713213354898490304645018692)

>>> two_squares(Integer(0))
(0, 0)

>>> two_squares(-Integer(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.valueation

EXAMPLES:
sage: valuation(512,2)
9
sage: valuation(1,2)
0

(continues on next page)
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):
  ... 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: valuation(int16(512), int16(2))
#...needs numpy
9

sage: from gmpy2 import mpz
sage: valuation(mpz(512), mpz(2))
9

>>> from sage.all import *
>>>
>>> valuation(Integer(100),Integer(10))
2
>>> valuation(Integer(200),Integer(10))
2
>>> valuation(Integer(243),Integer(3))
5
>>> valuation(Integer(243)*Integer(10007),Integer(3))
5
>>> valuation(Integer(243)*Integer(10007),Integer(10007))
1
>>> y = QQ['y'].gen()
>>> valuation(y**Integer(3), y)
3
>>> x = QQ[['x']].gen()
>>> valuation((x**Integer(3)-x**Integer(2))/(x-Integer(4)))
2
>>> valuation(4,2)
2
>>> valuation(1,1)
Traceback (most recent call last):
...
ValueError: You can only compute the valuation with respect to a integer larger than 1.

>>> from numpy import int16
#...needs numpy
>>> valuation(int16(Integer(512)), int16(Integer(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:**

```python
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)
# needs sage.rings.number_field
sage: g.xgcd(g + 2)
(1, 1/3*g, 0)
```

(continues on next page)
Here is an example of a \texttt{xgcd} for two polynomials over the integers, where the linear combination is not the gcd but the gcd multiplied by the resultant:

\begin{verbatim}
\begin{verbatim}
sage: R.<x> = ZZ[]
x
sage: gcd(Integer(2)*x*(x-Integer(1)), x**Integer(2))
x
sage: xgcd(Integer(2)*x*(x-Integer(1)), x**Integer(2))
(2*x, -1, 2)
\end{verbatim}
\end{verbatim}
\begin{verbatim}
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> gcd(Integer(2)*x*(x-Integer(1)), x**Integer(2))
x
>>> xgcd(Integer(2)*x*(x-Integer(1)), x**Integer(2))
(2*x, -1, 2)
\end{verbatim}
\end{verbatim}
\begin{verbatim}
>>> (Integer(2)*x*(x-Integer(1))).resultant(x) # needs sage.libs.pari
2
\end{verbatim}
\end{verbatim}

Tests with \texttt{numpy} and \texttt{gmpy2} types:

\begin{verbatim}
\begin{verbatim}
sage: from numpy import int8
\end{verbatim}
\end{verbatim}
\begin{verbatim}
\begin{verbatim}
sage: xgcd(4, int8(Integer(8)))
\end{verbatim}
\end{verbatim}
\begin{verbatim}
\begin{verbatim}
sage: xgcd(int8(4), int8(Integer(8)))
\end{verbatim}
\end{verbatim}
\begin{verbatim}
\begin{verbatim}
sage: from gmpy2 import mpz
\end{verbatim}
\end{verbatim}
\begin{verbatim}
\begin{verbatim}
sage: xgcd(mpz(4), mpz(Integer(8)))
\end{verbatim}
\end{verbatim}
\begin{verbatim}
\begin{verbatim}
sage: xgcd(mpz(Integer(4)), mpz(int8(Integer(8))))
\end{verbatim}
\end{verbatim}
\begin{verbatim}
\begin{verbatim}
sage: xgcd(int8(Integer(4)), mpz(Integer(8)))
\end{verbatim}
\end{verbatim}
\begin{verbatim}
\begin{verbatim}
sage: xgcd(mpz(int8(Integer(4))), mpz(Integer(8)))
\end{verbatim}
\end{verbatim}
\begin{verbatim}
\begin{verbatim}
(continues on next page)
\end{verbatim}
\end{verbatim}

(continues from previous page)
sage.arith.misc.xgcd(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
xgcd(Integer(4), mpz(Integer(8)))
```

(4, 1, 0)

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 \(l = \text{lcm}(m, n) = m_1 \cdot n_1\) where \(m_1 | m, n_1 | n\) and \(\text{gcd}(m_1, n_1) = 1\), 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 $\texttt{RationalField}$ represents the field $\mathbb{Q}$ of (arbitrary precision) rational numbers. Each rational number is an instance of the class $\texttt{Rational}$.

Interactively, an instance of $\texttt{RationalField}$ is available as $\texttt{QQ}$:

```
sage: QQ
Rational Field
```

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

```
sage: RealField(9).pi()  # needs sage.rings.real_mpfr
3.1
sage: QQ(RealField(9).pi())  # needs sage.rings.real_interval_field sage.rings.real_mpfr
22/7
sage: QQ(RealField().pi())  # needs sage.rings.real_interval_field sage.rings.real_mpfr
245850922/78256779
sage: QQ(35)
35
sage: QQ('12/347')
12/347
sage: QQ(exp(pi*I))  # needs sage.symbolic
-1
sage: x = polygen(ZZ)

sage: QQ((3*x)/(4*x))
3/4
```

```
>> from sage.all import *

>> QQ
Rational Field

>>> from sage.all import *

>>> RealField(Integer(9)).pi()  # needs sage.rings.real_mpfr
3.1

>>> QQ(RealField(Integer(9)).pi())

(continues on next page)
```
AUTHORS:

- Niles Johnson (2010-08): Issue #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()

class sage.rings.rational_field.RationalField

Bases: Singleton, NumberField

The class RationalField represents the field \( \mathbb{Q} \) of rational numbers.

EXAMPLES:

```sage
sage: a = 901824309821093821093812093810928309183091832091
sage: b = QQ(a); b
901824309821093821093812093810928309183091832091
sage: QQ(b)
901824309821093821093812093810928309183091832091
sage: QQ(int(93820984323))
93820984323
sage: QQ(ZZ(901824309821093821093812093810928309183091832091))
901824309821093821093812093810928309183091832091
sage: QQ('930482/9320842317')
-930482/9320842317
sage: QQ((-930482, 9320842317))
-930482/9320842317
sage: QQ([9320842317])
9320842317
sage: QQ(pari(3902938402384092830948242098430284398243982394))
3902938402384092830948242098430284398243982394
```

```python
>>> from sage.all import *
```
Conversion from the reals to the rationals is done by default using continued fractions.

If you specify the optional second argument base, then the string representation of the float is used.

(continues on next page)
Here's a nice example involving elliptic curves:

```
sage: # needs sage.rings.real_mpfr sage.schemes
sage: E = EllipticCurve('11a')

sage: L = E.lseries().at1(300)[0]; L
0.2538418608559106843377589233...

sage: O = E.period_lattice().omega(); O
1.26920930427955

sage: t = L/O; t
0.200000000000000

sage: QQ(RealField(45)(t))
1/5
```

```python
>>> from sage.all import *
```
absolute_polynomial()

Return a defining polynomial of \( \mathbb{Q} \), as for other number fields.

This is also aliased to \texttt{defining_polynomial()} and \texttt{absolute_polynomial()}.

EXAMPLES:

```
sage: QQ.polynomial()
x
```

algebraic_closure()

Return the algebraic closure of \texttt{self} (which is \( \overline{\mathbb{Q}} \)).

EXAMPLES:

```
sage: QQ.algebraic_closure()  # needs sage.rings.number_field
Algebraic Field
```

automorphisms()

Return all Galois automorphisms of \texttt{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
```

2.1. Field \( \mathbb{Q} \) of Rational Numbers

#### absolute_polynomial()
Return a defining polynomial of \( \mathbb{Q} \), as for other number fields.

This is also aliased to `defining_polynomial()` and `absolute_polynomial()`.

**EXAMPLES:**

```
sage: QQ.polynomial()
x
```

#### algebraic_closure()
Return the algebraic closure of `self` (which is \( \overline{\mathbb{Q}} \)).

**EXAMPLES:**

```
sage: QQ.algebraic_closure()  # needs sage.rings.number_field
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
```
>>> from sage.all import *
>>> c = QQ.characteristic(); c
0
>>> 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

>>> from sage.all import *
>>> QQ.class_number()
1

completion (p, prec, extras={})

Return the completion of \( \mathbb{Q} \) at \( p \).

EXAMPLES:

sage: QQ.completion(infinity, 53)  # needs sage.rings.real_mpfr
Real Field with 53 bits of precision
sage: QQ.completion(5, 15, {'print_mode': 'bars'})  # needs sage.rings.padics
5-adic Field with capped relative precision 15

>>> from sage.all import *
>>> QQ.completion(infinity, Integer(53))  
# needs sage.rings.real_mpfr
>>> QQ.completion(Integer(5), Integer(15), {'print_mode': 'bars'})  # needs sage.rings.padics
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()  # needs sage.rings.real_mpfr
Ring morphism:
  From: Rational Field
  To:   Complex Field with 53 bits of precision
  Defn: 1 |---> 1.00000000000000
sage: QQ.complex_embedding(20)  # needs sage.rings.real_mpfr
Ring morphism:
  From: Rational Field
  To:   Complex Field with 20 bits of precision
  Defn: 1 |---> 1.0000
>>> from sage.all import *
>>> QQ.complex_embedding() # needs sage.rings.real_mpfr
Ring morphism:
  From: Rational Field
  To:   Complex Field with 53 bits of precision
  Defn: 1 |-- 1.00000000000000

>>> QQ.complex_embedding(Integer(20)) # needs sage.rings.real_mpfr
Ring morphism:
  From: Rational Field
  To:   Complex Field with 20 bits of precision
  Defn: 1 |-- 1.0000

construction()
Return a pair (functor, parent) such that functor(parent) returns self.
This is the construction of \( \mathbb{Q} \) as the fraction field of \( \mathbb{Z} \).

EXAMPLES:

sage: QQ.construction()
(FractionField, Integer Ring)


gdel0> from sage.all import *
gdel0> QQ.construction()
(FractionField, Integer Ring)

defining_polynomial()
Return a defining polynomial of \( \mathbb{Q} \), as for other number fields.
This is also aliased to \texttt{defining_polynomial()} and \texttt{absolute_polynomial()}.

EXAMPLES:

sage: QQ.polynomial()
x


gdel0> from sage.all import *
gdel0> QQ.polynomial()
x

degree()
Return the degree of \( \mathbb{Q} \), which is 1.

EXAMPLES:

sage: QQ.degree()
1


gdel0> from sage.all import *
gdel0> QQ.degree()
1

discriminant()
Return the discriminant of the field of rational numbers, which is 1.

EXAMPLES:
**embeddings \((K)\)**

Return list of the one embedding of \(\mathbb{Q}\) into \(K\), if it exists.

**EXAMPLES:**

```python
sage: QQ.embeddings(QQ)
[Identity endomorphism of Rational Field]
sage: QQ.embeddings(CyclotomicField(5))
# needs sage.rings.number_field
[Coercion map:
  From: Rational Field
  To:  Cyclotomic Field of order 5 and degree 4]
```

\(K\) must have characteristic 0:

```python
sage: QQ.embeddings(GF(3))
Traceback (most recent call last):
  ...  
ValueError: no embeddings of the rational field into K.
```

```python
sage: QQ.embeddings(GF(Integer(3)))
Traceback (most recent call last):
  ...  
ValueError: no embeddings of the rational field into K.
```

**extension \((poly, \text{names}, **kwds)\)**

Create a field extension of \(\mathbb{Q}\).

**EXAMPLES:**

We make a single absolute extension:

```python
sage: x = polygen(QQ, 'x')
sage: K.<a> = QQ.extension(x^3 + 5); K
# needs sage.rings.number_field
Number Field in a with defining polynomial x^3 + 5
```

```python
>>> from sage.all import *
>>> x = polygen(QQ, 'x')
```

(continues on next page)
We make an extension generated by roots of two polynomials:

```python
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 $\mathbb{Q}$.

There is only the 0-th generator, which is 1.

**EXAMPLES:**

```python
sage: QQ.gen()
1
```

**gens ()**

Return a tuple of generators of $\mathbb{Q}$, which is only $(1,)$.  

**EXAMPLES:**

```python
sage: QQ.gens()
(1,)
```
hilbert_symbol_negative_at_S (S, b, check=True)

Return 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 \(\mathbb{Q}\) or as \(-1\)
- \(b\) – a non-zero rational number which is a non-square locally at every prime in \(S\).
- \(\text{check}\) – 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:**

```python
sage: QQ.hilbert_symbol_negative_at_S([-1, 5, 3, 2, 7, 11, 13, 23], -10/7)  # needs sage.rings.padics
-9867
sage: QQ.hilbert_symbol_negative_at_S([3, 5, QQ.places()[0], 11], -15)  # needs sage.rings.padics
-33
sage: QQ.hilbert_symbol_negative_at_S([3, 5], 2)  # needs sage.rings.padics
15
```

```python
>>> from sage.all import *
>>> QQ.hilbert_symbol_negative_at_S([-Integer(1), Integer(5), Integer(3),
... Integer(2), Integer(7), Integer(11), Integer(13), Integer(23)],
... -Integer(10)/7)  # needs sage.rings.padics
-9867
>>> QQ.hilbert_symbol_negative_at_S([Integer(3), Integer(5), QQ.
... places()[Integer(0)], Integer(11)], -Integer(15))  # needs sage.rings.
... padics
-33
>>> QQ.hilbert_symbol_negative_at_S([Integer(3), Integer(5)], Integer(2))  # needs sage.rings.padics
15
```

**AUTHORS:**

is_absolute() 

\(\mathbb{Q}\) is an absolute extension of \(\mathbb{Q}\).

**EXAMPLES:**

```python
sage: QQ.is_absolute()
True
```

```python
>>> from sage.all import *
>>> QQ.is_absolute()
True
```
is_prime_field()

Return True since $\mathbb{Q}$ is a prime field.

EXAMPLES:

```
sage: QQ.is_prime_field()
True
```

```
>>> from sage.all import *
>>> QQ.is_prime_field()
True
```

maximal_order()

Return the maximal order of the rational numbers, i.e., the ring $\mathbb{Z}$ of integers.

EXAMPLES:

```
sage: QQ.maximal_order()
Integer Ring
sage: QQ.ring_of_integers ()
Integer Ring
```

```
>>> from sage.all import *
>>> QQ.maximal_order()
Integer Ring
>>> QQ.ring_of_integers ()
Integer Ring
```

ngens()

Return the number of generators of $\mathbb{Q}$, which is 1.

EXAMPLES:

```
sage: QQ.ngens()
1
```

```
>>> from sage.all import *
>>> QQ.ngens()
1
```

number_field()

Return the number field associated to $\mathbb{Q}$. Since $\mathbb{Q}$ is a number field, this just returns $\mathbb{Q}$ again.

EXAMPLES:

```
sage: QQ.number_field() is QQ
True
```

```
>>> from sage.all import *
>>> QQ.number_field() is QQ
True
```

order()

Return the order of $\mathbb{Q}$, which is $\infty$.

EXAMPLES:
\texttt{sage}: \texttt{QQ.order()}

\texttt{+Infinity}

\begin{verbatim}
>>> from sage.all import *

>>> QQ.order()

+Infinity
\end{verbatim}

\textbf{places} \texttt{(all\_complex=False, prec=None)}

Return the collection of all infinite places of \texttt{self}, which in this case is just the embedding of \texttt{self} into \texttt{R}.

By default, this returns homomorphisms into \texttt{RR}. If \texttt{prec} is not None, we simply return homomorphisms into \texttt{RealField(prec)} (or \texttt{RDF} if \texttt{prec=53}).

There is an optional flag \texttt{all\_complex}, which defaults to \texttt{False}. If \texttt{all\_complex} is \texttt{True}, then the real embeddings are returned as embeddings into the corresponding complex field.

For consistency with non-trivial number fields.

\textbf{EXAMPLES}:

\begin{verbatim}
\texttt{sage}: \texttt{QQ.places()}

[Ring morphism:
 From: Rational Field
 To: Real Field with 53 bits of precision
 Defn: 1 \mapsto 1.0000000000000000000000000000000000000000000000000000000000]

\texttt{sage}: \texttt{QQ.places(prec=53)}

[Ring morphism:
 From: Rational Field
 To: Real Double Field
 Defn: 1 \mapsto 1.0]

\texttt{sage}: \texttt{QQ.places(prec=200, all\_complex=True)}

[Ring morphism:
 From: Rational Field
 To: Complex Field with 200 bits of precision
 Defn: 1 \mapsto 1.0000000000000000000000000000000000000000000000000000000000]
\end{verbatim}

\begin{verbatim}
\texttt{>>> from sage.all import *}

\texttt{>>> QQ.places()}

[Ring morphism:
 From: Rational Field
 To: Real Field with 53 bits of precision
 Defn: 1 \mapsto 1.0000000000000000000000000000000000000000000000000000000000]

\texttt{>>> QQ.places(prec=Integer(53))}

[Ring morphism:
 From: Rational Field
 To: Real Double Field
 Defn: 1 \mapsto 1.0]

\texttt{>>> QQ.places(prec=Integer(200), all\_complex=True)}

[Ring morphism:
 From: Rational Field
 To: Complex Field with 200 bits of precision
 Defn: 1 \mapsto 1.0000000000000000000000000000000000000000000000000000000000]
\end{verbatim}

\textbf{polynomial}()
Return a defining polynomial of \( \mathbb{Q} \), as for other number fields.

This is also aliased to \( \textit{defining\_polynomial()} \) and \( \textit{absolute\_polynomial()} \).

**EXAMPLES:**

```python
sage: QQ.polynomial()
```

```python
>>> from sage.all import *

sage: QQ.polynomial()
```

**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:**

```python
sage: QQ.power_basis()
```

```python
>>> from sage.all import *

sage: QQ.power_basis()
```

**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:**

```python
sage: it = QQ.primes_of_bounded_norm_iter(10)
sage: list(it)  # needs sage.libs.pari
[2, 3, 5, 7]
```

```python
>>> from sage.all import *

>>> it = QQ.primes_of_bounded_norm_iter(Integer(10))

>>> list(it)  # needs sage.libs.pari
[2, 3, 5, 7]
```

(continues on next page)
>>> list(QQ.primes_of_bounded_norm_iter(Integer(1)))
[]

\texttt{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:

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> QQ.quadratic_defect(Integer(0), Integer(7))
+Infinity
>>> QQ.quadratic_defect(Integer(5), Integer(7))
0
>>> QQ.quadratic_defect(Integer(5), Integer(2))
2
>>> QQ.quadratic_defect(Integer(5), Integer(5))
1
\end{verbatim}

\texttt{random\_element} (num\_bound=None, den\_bound=None, *args, **kwds)

Return a random element of $\mathbb{Q}$.

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 \texttt{sage.rings.integer\_ring.IntegerRing\_class.random\_element()}.

EXAMPLES:
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:

```python
sage: q = QQ.random_element(5, 10)
sage: -5/1 <= q <= 5/1
True
sage: q.denominator() <= 10
True
sage: q.numerator() <= 5
True
```
All rational numbers with height strictly less than 4:

```python
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, 3, 3, 3, 3, 3, 3, 3, 3]
```

All rational numbers with height 2:

```python
sage: list(QQ.range_by_height(2, 3))
[1/2, -1/2, 2, -2]
```

Nonsensical integer arguments will return an empty generator:

```python
sage: list(QQ.range_by_height(3, 3))
[]
sage: list(QQ.range_by_height(10, 1))
[]
```

There are no rational numbers with height \( \leq 0 \):

```python
sage: list(QQ.range_by_height(-10, 1))
[]
```

**relative_discriminant ()**

Return the relative discriminant, which is 1.

**EXAMPLES:**

```python
sage: QQ.relative_discriminant()
1
```

```python
>>> from sage.all import *
```

```python
>>> QQ.relative_discriminant()
1
```
residue_field \((p, \text{check}=\text{True})\)

Return the residue field of \(\mathbb{Q}\) at the prime \(p\), for consistency with other number fields.

**INPUT:**

- \(p\) – a prime integer.
- \(\text{check}\) (default \(\text{True}\)) – if \(\text{True}\), check the primality of \(p\), else do not.

**OUTPUT:** The residue field at this prime.

**EXAMPLES:**

```python
sage: QQ.residue_field(5)
Residue field of Integers modulo 5
sage: QQ.residue_field(next_prime(10^9))  #needs sage.rings.finite_rings
Residue field of Integers modulo 1000000007
```

selmer_generators \((S, m, \text{proof}=\text{True}, \text{orders}=\text{False})\)

Return generators of the group \(\mathbb{Q}(S, m)\).

**INPUT:**

- \(S\) – a set of primes
- \(m\) – a positive integer
- \(\text{proof}\) – ignored
- \(\text{orders}\) (default \(\text{False}\)) – if \(\text{True}\), output two lists, the generators and their orders

**OUTPUT:**

A list of generators of \(\mathbb{Q}(S, m)\) (and, optionally, their orders in \(\mathbb{Q}^*/(\mathbb{Q}^*)^m\)). This is the subgroup of \(\mathbb{Q}^*/(\mathbb{Q}^*)^m\) 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 \(\mathbb{Q}(S, m)\) contains the subgroup of those \(a\) such that \(\mathbb{Q}(\sqrt[m]{a})/\mathbb{Q}\) is unramified at all primes of \(\mathbb{Q}\) outside of \(S\), but may contain it properly when not all primes dividing \(m\) are in \(S\).

**See also:**

\texttt{RationalField.selmer\_space()}, which gives additional output when \(m = p\) is prime: as well as generators, it gives an abstract vector space over \(\mathbb{F}_p\) isomorphic to \(\mathbb{Q}(S, p)\) and maps implementing the isomorphism between this space and \(\mathbb{Q}(S, p)\) as a subgroup of \(\mathbb{Q}^*/(\mathbb{Q}^*)^p\).

**EXAMPLES:**

```python
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 Issue #31345 for details.

### selmer_group_iterator($S$, $m$, proof=True)

Return an iterator through elements of the finite group $\mathbb{Q}(S, m)$.

**INPUT:**
- $S$ – a set of primes
- $m$ – a positive integer
- `proof` - ignored

**OUTPUT:**
An iterator yielding the distinct elements of $\mathbb{Q}(S, m)$. 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]
```

(continues on next page)
list(QQ.selmer_group_iterator((Integer(2),), Integer(2)))
[1, 2, -1, -2]
list(QQ.selmer_group_iterator((Integer(2),Integer(3)), Integer(2)))
[1, 3, 2, 6, -1, -3, -2, -6]
list(QQ.selmer_group_iterator((Integer(5),), Integer(2)))
[1, 5, -1, -5]

selmer_space \( (S, p, \text{proof}=\text{None}) \)
Compute the group \( \mathbb{Q}(S, p) \) as a vector space with maps to and from \( \mathbb{Q}^* \).

**INPUT:**
- \( S \) – a list of prime numbers
- \( p \) – a prime number

**OUTPUT:**
(tuple) \( QSp, QSp\_gens, \text{from\_QSp}, \text{to\_QSp} \) where
- \( QSp \) is an abstract vector space over \( \mathbb{F}_p \) isomorphic to \( \mathbb{Q}(S, p) \);
- \( QSp\_gens \) is a list of elements of \( \mathbb{Q}^* \) generating \( \mathbb{Q}(S, p) \);
- \( \text{from\_QSp} \) is a function from \( QSp \) to \( \mathbb{Q}^* \) implementing the isomorphism from the abstract \( \mathbb{Q}(S, p) \) to \( \mathbb{Q}(S, p) \) as a subgroup of \( \mathbb{Q}^*/(\mathbb{Q}^*)^p \);
- \( \text{to\_QSp} \) is a partial function from \( \mathbb{Q}^* \) to \( QSp \), defined on elements \( a \) whose image in \( \mathbb{Q}^*/(\mathbb{Q}^*)^p \) lies in \( \mathbb{Q}(S, p) \), mapping them via the inverse isomorphism to the abstract vector space \( QSp \).

The group \( \mathbb{Q}(S, p) \) is the finite subgroup of \( \mathbb{Q}^*/(\mathbb{Q}^*)^p \) consisting of elements whose valuation at all primes not in \( S \) is a multiple of \( p \). It contains the subgroup of those \( a \in \mathbb{Q}^* \) such that \( \mathbb{Q}(\sqrt[p]{a})/\mathbb{Q} \) is unramified at all primes of \( \mathbb{Q} \) outside of \( S \), but may contain it properly when \( p \) is not in \( S \).

**EXAMPLES:**

When \( S \) is empty, \( \mathbb{Q}(S, p) \) is only nontrivial for \( p = 2 \):

```sage
sage: Q2, Q2gens, fromQ2, toQ2 = QQ.selmer_space([], 2) #
˓→ needs sage.rings.number_field
sage: Q2  #˓→ needs sage.rings.number_field
Vector space of dimension 1 over Finite Field of size 2
sage: Q2gens  #˓→ needs sage.rings.number_field
[[-1]]
```

```sage
all(QQ.selmer_space([], p)[0].dimension() == 0 #˓→ needs sage.libs.pari sage.rings.number_field
....:    for p in primes(3, 10))
```

True

```sage
from sage.all import *

sage: Q2, Q2gens, fromQ2, toQ2 = QQ.selmer_space([], Integer(2)) # ˓→ needs sage.rings.number_field
sage: Q2  #˓→ needs sage.rings.number_field
Vector space of dimension 1 over Finite Field of size 2
sage: Q2gens  #˓→ needs sage.rings.number_field
```
In general there is one generator for each \( p \in S \), and an additional generator of \(-1\) when \( p = 2\):

\[
\begin{align*}
\text{sage:} & \quad \text{# needs sage.modules sage.rings.number_field} \\
\text{sage:} & \quad QS2, QS2gens, fromQS2, toQS2 = QQ.selmer_space([5, 7], 2) \\
\text{sage:} & \quad QS2 \\
& \text{Vector space of dimension 3 over Finite Field of size 2} \\
\text{sage:} & \quad QS2gens \\
& \quad [5, 7, -1] \\
\text{sage:} & \quad toQS2(-7) \\
& \quad (0, 1, 1) \\
\text{sage:} & \quad fromQS2((0, 1, 1)) \\
& \quad -7
\end{align*}
\]

The map \( \text{fromQS2} \) is only well-defined modulo \( p \)'th powers (in this case, modulo squares):

\[
\begin{align*}
\text{sage:} & \quad \text{toQS2}(-5/7) \\
& \quad \rightarrow \text{needs sage.modules sage.rings.number_field} \\
& \quad (1, 1, 1) \\
\text{sage:} & \quad \text{fromQS2}((1, 1, 1)) \\
& \quad \rightarrow \text{needs sage.modules sage.rings.number_field} \\
& \quad -35 \\
\text{sage:} & \quad \{(-5/7)/(-35)\}.is_square() \\
& \quad True
\end{align*}
\]

The map \( \text{toQS2} \) is not defined on all of \( \mathbb{Q}^* \), only on those numbers which are squares away from 5 and 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 $\mathbb{Q}$.

See TestSuite() for a typical use case.

OUTPUT: An iterator over 100 elements of $\mathbb{Q}$.

EXAMPLES:

```
sage: tuple(QQ.some_elements())
```

(continues on next page)
valuation \( (p) \)
Return the discrete valuation with uniformizer \( p \).

EXAMPLES:

```python
sage: v = QQ.valuation(3); v
# needs sage.rings.padics
3-adic valuation
sage: v(1/3)
# needs sage.rings.padics
-1
```

```python
>>> from sage.all import *
>>> v = QQ.valuation(Integer(3)); v
# needs sage.rings.padics
3-adic valuation
>>> v(Integer(1)/Integer(3))
# needs sage.rings.padics
-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:

```python
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
```
>>> from sage.all import *
>>> QQ.zeta()
-1
>>> QQ.zeta(Integer(2))
-1
>>> QQ.zeta(Integer(1))
1
>>> QQ.zeta(Integer(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 \( \frac{n}{d} \).

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)
doctest:warning... DeprecationWarning: The function is_RationalField is deprecated; use 'isinstance(..., RationalField)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
sage: is_RF(ZZ)
False
```

```
sage: from sage.all import *
>>> from sage.rings.rational_field import is_RationalField as is_RF
>>> is_RF(QQ)
doctest:warning... DeprecationWarning: The function is_RationalField is deprecated; use 'isinstance(..., RationalField)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
>>> is_RF(ZZ)
False
```
2.2 Rational Numbers

AUTHORS:

- William Stein (2005): first version
- Gonzalo Tornaria and William Stein (2006-03-02): greatly improved python/GMP conversion; hashing
- 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
- 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.

```python
class sage.rings.rational.Q_to_Z
    Bases: Map
    
    A morphism from \mathbb{Q} to \mathbb{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

    >>> from sage.all import *
    >>> sage.rings.rational.Q_to_Z(QQ, ZZ).section()
    Natural morphism:
    From: Integer Ring
    To:   Rational Field
```

```python
class sage.rings.rational.Rational
    Bases: 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))  # needs sage.rings.number_field
5/2
sage: Rational(AA(209735/343 - 17910/49*golden_ratio).nth_root(3))  # needs sage.rings.number_field sage.symbolic
53/7
sage: QQ(float(1.5))
3/2
sage: QQ(RDF(1.2))
6/5

>>> from sage.all import *
>>> a = ~Integer(2)/Integer(3)
>>> type(a)
<class 'sage.rings.rational.Rational'>
>>> parent(a)
Rational Field
>>> Rational('1/0')
Traceback (most recent call last):
  ...
TypeError: unable to convert '1/0' to a rational
>>> Rational(RealNumber('1.5'))
3/2
>>> Rational((Integer(2)**Integer(99),Integer(2)**Integer(100)))
1/2
>>> Rational("2", "10"), Integer(16)
1/8
>>> Rational(QQbar(Integer(125)/Integer(8)).nth_root(Integer(3)))  # needs sage.rings.number_field
5/2
>>> Rational(AA(Integer(209735)/Integer(343) - Integer(17910)/Integer(49)*golden_ratio).nth_root(Integer(3))  # needs sage.rings.number_field sage.symbolic
53/7
>>> QQ(float(RealNumber('1.5')))
3/2

(continues on next page)
Conversion from fractions:

```python
sage: import fractions
sage: f = fractions.Fraction(1r, 2r)
sage: Rational(f)
1/2
```

Conversion from PARI:

```python
sage: Rational(pari('-939082/3992923'))  # needs sage.libs.pari
-939082/3992923
sage: Rational(pari('Pol([-1/2])'))  # needs sage.libs.pari
-1/2
```

Conversions from numpy:

```python
sage: # needs numpy
sage: import numpy as np
sage: QQ(np.int8('-15'))
-15
sage: QQ(np.int16('-32'))
-32
sage: QQ(np.int32('-19'))
-19
sage: QQ(np.uint32('1412'))
1412
sage: QQ(np.float16('12'))  # needs numpy
12
```

```python
>>> from sage.all import *

>>> import fractions

>>> f = fractions.Fraction(1, 2)

>>> Rational(f)
1/2
```

>> from sage.all import *
>> import fractions

>>> f = fractions.Fraction(1, 2)

>>> Rational(f)
1/2

Conversion from PARI:

```python
sage: Rational(pari('-939082/3992923'))  # needs sage.libs.pari
-939082/3992923
sage: Rational(pari('Pol([-1/2])'))  # needs sage.libs.pari
-1/2
```

Conversions from numpy:

```python
sage: # needs numpy
sage: import numpy as np
sage: QQ(np.int8('-15'))
-15
sage: QQ(np.int16('-32'))
-32
sage: QQ(np.int32('-19'))
-19
sage: QQ(np.uint32('1412'))
1412
sage: QQ(np.float16('12'))  # needs numpy
12
```

```python
>>> from sage.all import *

>>> import fractions

>>> f = fractions.Fraction(1, 2)

>>> Rational(f)
1/2
```
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
```

```
>>> from sage.all import *
>>> from gmpy2 import *

>>> QQ(mpq(3/4))
3/4
>>> QQ(mpz(Integer(42)))
42
>>> Rational(mpq(Integer(2)/Integer(3)))
2/3
>>> Rational(mpz(Integer(5)))
5
```

**absolute_norm()**

Return the norm from \(\mathbb{Q}\) to \(\mathbb{Q}\) of \(x\) (which is just \(x\)). This was added for compatibility with NumberFields

```
sage: (6/5).absolute_norm()
6/5
sage: QQ(7/5).absolute_norm()
7/5
```

```
>>> from sage.all import *

>>> (Integer(6)/Integer(5)).absolute_norm()
6/5
>>> QQ(Integer(7)/Integer(5)).absolute_norm()
7/5
```

**additive_order()**

Return the additive order of \(\text{self}\).

OUTPUT: integer or infinity
EXAMPLES:

```python
sage: QQ(0).additive_order()
1
sage: QQ(1).additive_order()
+Infinity
```

```python
>>> from sage.all import *

>>> QQ(Integer(0)).additive_order()
1
>>> QQ(Integer(1)).additive_order()
+Infinity
```

**as_integer_ratio()**

Return the pair `(self.numerator(), self.denominator())`.

EXAMPLES:

```python
sage: x = -12/29
sage: x.as_integer_ratio()
(-12, 29)
```

```python
>>> from sage.all import *

>>> x = -Integer(12)/Integer(29)

>>> 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:

```python
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
```

```python
>>> from sage.all import *

>>> n = Integer(5)/Integer(3); n.ceil()
2

>>> n = -Integer(17)/Integer(19); n.ceil()
0

>>> n = -Integer(7)/Integer(2); n.ceil()
-3

>>> n = Integer(7)/Integer(2); n.ceil()
4

>>> n = Integer(10)/Integer(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
```

```
>>> from sage.all import *

>>> (Integer(1)/Integer(3)).charpoly('x')
x - 1/3
```

The default is var='x'. (Issue #20967):

```
sage: a = QQ(2); a.charpoly('x')
x - 2
```

```
>>> from sage.all import *

>>> a = QQ(Integer(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
```

```
>>> from sage.all import *

>>> n = Integer(23)/Integer(11)
>>> 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)
sage: a.content([2/5, 4/9])
2/45

```python
from sage.all import *

a = Integer(2)/Integer(3)
a.content(Integer(2)/Integer(3))
2/3

a.content(Integer(1)/Integer(5))
1/15

a.content([Integer(2)/Integer(5), Integer(4)/Integer(9)])
2/45
```

continued_fraction()

Return the continued fraction of that rational.

EXAMPLES:

```python
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)  # needs sage.rings.real_mpfr
3.141592920
sage: pi.n(digits=10)  # needs sage.rings.real_mpfr sage.symbolic
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:

```python
sage: (13/9).continued_fraction_list()
[1, 2]
sage: 1 + 1/(2 + 1/4)
```

(continues on next page)
13/9

\texttt{sage: (225/157).continued_fraction_list()}
\[1, 2, 3, 4, 5\]
\texttt{sage: 1 + 1/(2 + 1/(3 + 1/(4 + 1/5)))}
\frac{225}{157}

\texttt{sage: (fibonacci(20)/fibonacci(19)).continued_fraction_list()}
\# needs sage.libs pari
\[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2\]

\texttt{sage: (-1/3).continued_fraction_list()}
\[-1, 1, 2\]

\begin{verbatim}
>>> from sage.all import *

>>> (Integer(13)/Integer(9)).continued_fraction_list()
[1, 2, 4]

>>> Integer(1) + Integer(1)/(Integer(2) + Integer(1)/Integer(4))
13/9

>>> (Integer(225)/Integer(157)).continued_fraction_list()
[1, 2, 3, 4, 5]

>>> Integer(1) + Integer(1)/(Integer(2) + Integer(1)/(Integer(3) + Integer(1)/
\rightarrow (Integer(4) + Integer(1)/Integer(5)))))
\frac{225}{157}

>>> (fibonacci(Integer(20))/fibonacci(Integer(19))).continued_fraction_list()
\# needs sage.libs pari
\[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2\]

>>> (-Integer(1)/Integer(3)).continued_fraction_list()
\[-1, 1, 2\]
\end{verbatim}

Check that the partial quotients of an integer \(n\) is simply \([n]\):

\texttt{sage: QQ(1).continued_fraction_list()}
\[1\]
\texttt{sage: QQ(0).continued_fraction_list()}
\[0\]
\texttt{sage: QQ(-1).continued_fraction_list()}
\[-1\]

\begin{verbatim}
>>> from sage.all import *

>>> QQ(Integer(1)).continued_fraction_list()
[1]

>>> QQ(Integer(0)).continued_fraction_list()
[0]

>>> QQ(-Integer(1)).continued_fraction_list()
[-1]
\end{verbatim}

Hirzebruch-Jung continued fractions:

\texttt{sage: (11/19).continued_fraction_list("hj")}
\[1, 3, 2, 3, 2\]
\texttt{sage: 1 - 1/(3 - 1/(2 - 1/(3 - 1/2)))}
\frac{11}{19}

(continues on next page)
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

>>> from sage.all import *

>>> (Integer(11)/Integer(19)).continued_fraction_list("hj")
[1, 3, 2, 3, 2]
>>> Integer(1) - Integer(1)/(Integer(3) - Integer(1)/(Integer(2) - Integer(1)/Integer(3) - Integer(1)/Integer(2))))
11/19

>>> (Integer(225)/Integer(137)).continued_fraction_list("hj")
[2, 3, 5, 10]
>>> Integer(2) - Integer(1)/(Integer(3) - Integer(1)/(Integer(5) - Integer(1)/Integer(10))))
225/137

>>> (-Integer(23)/Integer(19)).continued_fraction_list("hj")
[-1, 5, 4]
>>> -Integer(1) - Integer(1)/(Integer(5) - Integer(1)/Integer(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

>>> from sage.all import *

>>> x = -Integer(5)/Integer(11)

>>> x.denominator()
11

>>> x = Integer(9)/Integer(3)

>>> x.denominator()
1

>>> x = Integer(5)/Integer(13)
denominator()

Return the denominator of this rational number. \texttt{denom()} is an alias of \texttt{denominator()}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: x = -5/11
sage: x.denominator()
11

sage: x = 9/3
sage: x.denominator()
1

sage: x = 5/13
sage: x.denom()
13
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> x = -Integer(5)/Integer(11)

>>> x.denominator()
11

>>> x = Integer(9)/Integer(3)

>>> x.denominator()
1

>>> x = Integer(5)/Integer(13)

>>> x.denom()
13
\end{verbatim}

\textbf{factor()}

Return the factorization of this rational number.

\textbf{OUTPUT: Factorization}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: (-4/17).factor()
-1 * 2^2 * 17^-1

>>> from sage.all import *

>>> (-Integer(4)/Integer(17)).factor()
-1 * 2^2 * 17^-1
\end{verbatim}

Trying to factor 0 gives an arithmetic error:

\begin{verbatim}
sage: (0/1).factor()
Traceback (most recent call last):
...
ArithmeticError: factorization of 0 is not defined

>>> from sage.all import *

>>> (Integer(0)/Integer(1)).factor()
\end{verbatim}
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
```

```
>>> from sage.all import *
>>> n = Integer(5)/Integer(3); n.floor()
1
>>> n = -Integer(17)/Integer(19); n.floor()
-1
>>> n = -Integer(7)/Integer(2); n.floor()
-4
>>> n = Integer(7)/Integer(2); n.floor()
3
>>> n = Integer(10)/Integer(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: # needs sage.symbolic
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)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> gamma(Integer(1)/Integer(2))
sqrt(pi)
```

(continues on next page)
This function accepts an optional precision argument:

```python
sage: (1/3).gamma(prec=100) # needs sage.rings.real_mpfr
2.6789385347077476336556929410
sage: (1/2).gamma(prec=100) # needs sage.rings.real_mpfr
1.7724538509055160272981674833
```

```
>>> from sage.all import *
```

```python
>>> (Integer(1)/Integer(3)).gamma(prec=Integer(100)) # needs sage.rings.real_mpfr
2.6789385347077476336556929410
```

```python
>>> (Integer(1)/Integer(2)).gamma(prec=Integer(100)) # needs sage.rings.real_mpfr
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:

```python
sage: # needs sage.rings.real_mpfr
sage: a = QQ(6/25)
sage: a.global_height_arch() + a.global_height_non_arch()
3.218775824862820
sage: a.global_height() 3.218775824862820
sage: a = QQ(0)
sage: a.global_height() 0.000000000000000
sage: a = QQ(1)
sage: a.global_height() 0.000000000000000
```
from sage.all import *
# needs sage.rings.real_mpfr

a = QQ(Integer(6)/Integer(25))
a.global_height_arch() + a.global_height_non_arch()
3.2187582486820
a.global_height()
3.2187582486820
(Integer(1)/a).global_height()
3.2187582486820
QQ(Integer(0)).global_height()
0.000000000000000
QQ(Integer(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 \( \mathbb{Q} \) 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:

```python
sage: a = QQ(6/25)
sage: a.global_height_arch()  # needs sage.rings.real_mpfr
0.000000000000000
sage: (1/a).global_height_arch()  # needs sage.rings.real_mpfr
1.42711635564015
sage: (1/a).global_height_arch(100)  # needs sage.rings.real_mpfr
1.4271163556401457483890413081
```

global_height_non_arch (prec= None)
Return the total non-archimedean component of the height of this rational number.

INPUT:
• \texttt{prec (int)} – desired floating point precision (default: default \texttt{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()  # needs sage.rings.real_mpfr
1.79175946922805
sage: [a.local_height(p) for p in a.support()]  # needs sage.rings.real_mpfr
[0.693147180559945, 1.0986128886811, 0.000000000000000]
sage: sum([a.local_height(p) for p in a.support()])  # needs sage.rings.real_mpfr
1.79175946922805
```

```
>>> from sage.all import *

>>> a = ZZ(2)/ZZ(3)  # needs sage.rings.real_mpfr

>>> a.support()
[2, 3]

>>> a.global_height_non_arch()  # needs sage.rings.real_mpfr
1.79175946922805

>>> [a.local_height(p) for p in a.support()]  # needs sage.rings.real_mpfr
[0.693147180559945, 1.0986128886811, 0.000000000000000]

>>> sum([a.local_height(p) for p in a.support()])  # needs sage.rings.real_mpfr
1.79175946922805
```

**height ()**

The max absolute value of the numerator and denominator of \texttt{self}, as an \texttt{Integer}.

**OUTPUT:** \texttt{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
```

```
>>> from sage.all import *

>>> a = ZZ(2)/ZZ(3)
```

(continues on next page)
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
```

```
>>> from sage.all import *
>>> (Integer(1)/Integer(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]
```

```
>>> from sage.all import *
>>> QQ(Integer(1)/Integer(2)).is_S_integral()
False
>>> QQ(Integer(1)/Integer(2)).is_S_integral([Integer(2)])
```
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:**

```python
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 $\mathbb{Z}$).

**OUTPUT:** bool

**EXAMPLES:**

```python
sage: QQ(1/2).is_integral()
False
sage: QQ(4/4).is_integral()
True
```

2.2. Rational Numbers
**is_integral()**

Determine if a rational number is integral (i.e., is in \( \mathbb{Z} \)).

**OUTPUT:** bool

**EXAMPLES:**

```python
sage: QQ(1/2).is_integral()
False
sage: QQ(4/4).is_integral()
True
```

```python
>>> from sage.all import *
>>> QQ(Integer(1)/Integer(2)).is_integral()
False
>>> QQ(Integer(4)/Integer(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 the PARI function `pari:bnfnorm`. See `_bnfnorm()`.

**EXAMPLES:**

```python
sage: # needs sage.rings.number_field
sage: x = polygen(QQ, 'x')
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)
```

```python
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> x = polygen(QQ, 'x')
```
The number field doesn’t have to be defined by an integral polynomial:

```python
sage: B, e = (1/5).is_norm(QuadraticField(5/4, a), element=True)     # needs sage.rings.number_field
sage: B                         # needs sage.rings.number_field
True
sage: e.norm()                 # needs sage.rings.number_field
1/5
```

A non-Galois number field:

```python
sage: # needs sage.rings.number_field
sage: K.<a> = NumberField(x^3 - 2)
sage: B, e = (3/5).is_norm(K, element=True); B
True
sage: e.norm()                 # needs sage.groups
3/5
```

```
>>> from sage.all import *
>>> B, e = (Integer(1)/Integer(5)).is_norm(QuadraticField(Integer(5)/
˓→Integer(4), 'a'), element=True)                               # needs sage.rings.number_field
>>> B                         # needs sage.rings.number_field
True
>>> e.norm()                 # needs sage.rings.number_field
1/5
```

2.2. Rational Numbers 289
>>> B, e = (Integer(3)/Integer(5)).is_norm(K, element=True); B
True
>>> e.norm()
3/5
>>> Integer(7).is_norm(K)
˓→ # needs sage.groups
Traceback (most recent call last):
... NotImplementedError: is_norm is not implemented unconditionally
for norms from non-Galois number fields
>>> Integer(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

>>> from sage.all import *

>>> QQ(Integer(25)/Integer(4)).is_nth_power(Integer(2))
True
>>> QQ(Integer(125)/Integer(8)).is_nth_power(Integer(3))
True
>>> QQ(-Integer(125)/Integer(8)).is_nth_power(Integer(3))
True
>>> QQ(Integer(25)/Integer(4)).is_nth_power(-Integer(2))
is_one()
Determine if a rational number is one.
OUTPUT: bool
EXAMPLES:

```python
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 $\mathbb{Q}_p$ (or in $R$ when $p = \infty$).
INPUT:

- $p$ – a prime number, or infinity
- check – (default: True); check if $p$ is prime

EXAMPLES:

```python
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:**

```python
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
```

```python
>>> from sage.all import *
>>> (Integer(4)/Integer(9)).is_perfect_power()
True
>>> (Integer(144)/Integer(1)).is_perfect_power()
True
>>> (Integer(4)/Integer(3)).is_perfect_power()
False
>>> (Integer(2)/Integer(27)).is_perfect_power()
False
>>> (Integer(4)/Integer(27)).is_perfect_power()
False
>>> (-Integer(1)/Integer(25)).is_perfect_power()
False
>>> (-Integer(1)/Integer(27)).is_perfect_power()
True
>>> (Integer(0)/Integer(1)).is_perfect_power()
True
```

The second parameter does not change the result, but may change the runtime.

```python
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
```

```python
>>> from sage.all import *
>>> (-Integer(1)/Integer(27)).is_perfect_power(True)
True
>>> (-Integer(1)/Integer(25)).is_perfect_power(True)
False
>>> (Integer(2)/Integer(27)).is_perfect_power(True)
False
```

(continues on next page)
This test makes sure we workaround a bug in GMP (see Issue #4612):

```python
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:**

```python
sage: (3/4).is_rational()
True
```

```python
>>> from sage.all import *
>>> (Integer(3)/Integer(4)).is_rational()
True
```

### is_square()

Return whether or not this rational number is a square.

**OUTPUT:** bool

**EXAMPLES:**

```python
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
```

```python
>>> from sage.all import *
>>> x = Integer(9)/Integer(4)
>>> x.is_square()
True
>>> x = (Integer(7)/Integer(53))**Integer(100)
>>> x.is_square()
```

(continues on next page)
True
>>> x = Integer(4)/Integer(3)
>>> x.is_square()
False
>>> x = -Integer(1)/Integer(4)
>>> 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]

>>> from sage.all import *
>>> m = Integer(5)/Integer(3)
>>> 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
- \(\text{prec} \ (\text{int})\) – desired floating point precision (default: default RealField precision).

**OUTPUT:**

(\text{real}) The local height of this rational number at the prime \(p\).

**EXAMPLES:**

```
sage: a = QQ(25/6)
sage: a.local_height(2)  # needs sage.rings.real_mpfr
0.693147180559945
sage: a.local_height(3)  # needs sage.rings.real_mpfr
1.09861228866811
sage: a.local_height(5)  # needs sage.rings.real_mpfr
0.000000000000000

>>> from sage.all import *
>>> a = QQ(Integer(25)/Integer(6))
>>> a.local_height(Integer(2))  # needs sage.rings.real_mpfr
0.693147180559945
>>> a.local_height(Integer(3))
```

(continues on next page)
local_height_arch \( (\text{prec} = \text{None}) \)

Return the Archimedean local height of this rational number at the infinite place.

**INPUT:**

- \( \text{prec} \) (int) – desired floating point precision (default: default RealField precision).

**OUTPUT:**

(\( \text{real} \)) The local height of this rational number \( x \) at the unique infinite place of \( \mathbb{Q} \), which is \( \max(\log(|x|), 0) \).

**EXAMPLES:**

```python
sage: a = QQ(6/25)
sage: a.local_height_arch()
# needs sage.rings.real_mpfr
0.00000000000000
sage: (1/a).local_height_arch()
# needs sage.rings.real_mpfr
1.42711635564015
sage: (1/a).local_height_arch(Integer(100))
# needs sage.rings.real_mpfr
1.4271163556401457483890413081
```

log \( (\text{m} = \text{None}, \text{prec} = \text{None}) \)

Return the log of \( \text{self} \).

**INPUT:**

- \( \text{m} \) – the base (default: natural log base \( e \))
- \( \text{prec} \) – integer (optional); the precision in bits

**OUTPUT:**

When \( \text{prec} \) is not given, the log as an element in symbolic ring unless the logarithm is exact. Otherwise the log is a RealField approximation to \( \text{prec} \) bit precision.

**EXAMPLES:**

```python
>>> from sage.all import *
>>> a = QQ(Integer(6)/Integer(25))
>>> a.local_height_arch()
# needs sage.rings.real_mpfr
0.00000000000000
>>> (Integer(1)/a).local_height_arch()
# needs sage.rings.real_mpfr
1.42711635564015
>>> (Integer(1)/a).local_height_arch(Integer(100))
# needs sage.rings.real_mpfr
1.4271163556401457483890413081
```
sage: (124/345).log(5)  # needs sage.symbolic
log(124/345)/log(5)
sage: (124/345).log(5, 100)  # needs sage.rings.real_mpfr
-0.63578895682825611710391773754
sage: log(QQ(125))  # needs sage.symbolic
3*log(5)
sage: log(QQ(125), 5)
3
sage: log(QQ(125), 3)  # needs sage.symbolic
3*log(5)/log(3)
sage: QQ(8).log(1/2)
-3
sage: (1/8).log(1/2)
3
sage: (1/2).log(1/8)
1/3
sage: (1/2).log(8)
-1/3
sage: (16/81).log(8/27)  # needs sage.libs.pari
4/3
sage: (8/27).log(16/81)  # needs sage.libs.pari
3/4
sage: log(27/8, 16/81)  # needs sage.libs.pari
-3/4
sage: (125/8).log(5/2)  # needs sage.libs.pari
3
sage: (125/8).log(5/2, prec=53)  # needs sage.rings.real_mpfr
3.00000000000000

>>> from sage.all import *
>>> (Integer(124)/Integer(345)).log(Integer(5))  # needs sage.symbolic
log(124/345)/log(5)
>>> (Integer(124)/Integer(345)).log(Integer(5), Integer(100))  # needs sage.rings.real_mpfr
-0.63578895682825611710391773754
>>> log(QQ(Integer(125)))  # needs sage.symbolic
3*log(5)
>>> log(QQ(Integer(125)), Integer(5))
3
>>> log(QQ(Integer(125)), Integer(3))  # needs sage.symbolic
3*log(5)/log(3)
>>> QQ(Integer(8)).log(Integer(1)/Integer(2))
`minpoly (var='x')`

Return the minimal polynomial of this rational number. This will always be just \( x - \text{self} \); 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
sage: (1/3).minpoly()
x - 1/3
sage: (1/3).minpoly('y')
y - 1/3
```

```python
>>> from sage.all import *
>>> (Integer(1)/Integer(3)).minpoly()
x - 1/3
>>> (Integer(1)/Integer(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:

```bash
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.
```

```bash
>>> from sage.all import *

>>> (-Integer(4)/Integer(17)).mod_ui(Integer(3))
1
>>> (-Integer(4)/Integer(17)).mod_ui(Integer(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:

```bash
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
```

```bash
>>> from sage.all import *

>>> QQ(Integer(1)).multiplicative_order()
1
>>> QQ('1/-1').multiplicative_order()
2
```

**norm()**

Return the norm from \( \mathbb{Q} \) to \( \mathbb{Q} \) of \( x \) (which is just \( x \)). This was added for compatibility with `NumberField`.

OUTPUT:

- Rational – reference to self

EXAMPLES:
AUTHORS:
- Craig Citro

\texttt{nth\_root}(n)

Computes the \( n \)-th root of \texttt{self}, or raises a \texttt{ValueError} if \texttt{self} is not a perfect \( n \)-th power.

INPUT:
- \( n \) – integer (must fit in \texttt{C int} type)

AUTHORS:
- David Harvey (2006-09-15)

EXAMPLES:

\begin{verbatim}
sage: (25/4).nth_root(2) 5/2
da = (125/8).nth_root(3) 5/2
da = (-125/8).nth_root(3) -5/2
da = (25/4).nth_root(-2) 2/5

>>> from sage.all import *
>>> (Integer(25)/Integer(4)).nth_root(Integer(2))
5/2
>>> (Integer(125)/Integer(8)).nth_root(Integer(3))
5/2
>>> (-Integer(125)/Integer(8)).nth_root(Integer(3))
-5/2
>>> (Integer(25)/Integer(4)).nth_root(-Integer(2))
2/5

sage: (9/2).nth_root(2)
Traceback (most recent call last):
...
ValueError: not a perfect 2nd power

>>> from sage.all import *
>>> (Integer(9)/Integer(2)).nth_root(Integer(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
\end{verbatim}
>>> from sage.all import *

>>> (-Integer(25)/Integer(4)).nth_root(Integer(2))
Traceback (most recent call last):
...
ValueError: cannot take even root of negative number

\texttt{numerator()}

Return the numerator of this rational number. \texttt{numerator} is an alias of \texttt{numerator()}.\par

\textbf{EXAMPLES:}

\begin{verbatim}
sage: x = 5/11
sage: x.numerator()
5

sage: x = 9/3
sage: x.numerator()
3

sage: x = -5/11
sage: x.numer()
-5

>>> from sage.all import *

>>> x = Integer(5)/Integer(11)

>>> x.numerator()
5

>>> x = Integer(9)/Integer(3)

>>> x.numerator()
3

>>> x = -Integer(5)/Integer(11)

>>> x.numer()
-5
\end{verbatim}

\texttt{numerator()}

Return the numerator of this rational number. \texttt{numerator} is an alias of \texttt{numerator()}.\par

\textbf{EXAMPLES:}

\begin{verbatim}
sage: x = 5/11
sage: x.numerator()
5

sage: x = 9/3
sage: x.numerator()
3

sage: x = -5/11
sage: x.numer()
-5
\end{verbatim}
>> x = Integer(9)/Integer(3)
>>> x.numerator()
3

>> x = -Integer(5)/Integer(11)
>>> x.numerator()
-5

\textbf{ord}(p)

Return the power of \( p \) in the factorization of \( \text{self} \).

**INPUT:**

- \( p \) – a prime number

**OUTPUT:**

(integer or infinity) \( \text{Infinity} \) if \( \text{self} \) is zero, otherwise the (positive or negative) integer \( e \) such that \( \text{self} = m \times p^e \) with \( m \) coprime to \( p \).

**Note:** See also \texttt{val_unit()} which returns the pair \((e, m)\). The function \texttt{ord()} is an alias for \texttt{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
```

```
>>> from sage.all import *

>>> x = -Integer(5)/Integer(9)
>>> x.ord(Integer(5))
1
>>> x.valuation(Integer(5))
1
>>> x.valuation(Integer(3))
-2
>>> x.valuation(Integer(2))
0
```

Some edge cases:

```
sage: (0/1).valuation(4)
+Infinity
sage: (7/16).valuation(4)
-2
```
>>> from sage.all import *
>>> (Integer(0)/Integer(1)).valuation(Integer(4))
+Infinity
>>> (Integer(7)/Integer(16)).valuation(Integer(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 $d = 2^a5^b m$ 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()  # needs sage.libs.pari
6
sage: RR(1/7)  # needs sage.rings.real_mpfr
0.142857142857143
sage: (1/8).period()  # needs sage.libs.pari
1
sage: RR(1/8)  # needs sage.rings.real_mpfr
0.125000000000000
sage: RR(1/6)  # needs sage.rings.real_mpfr
0.166666666666667
sage: (1/6).period()  # needs sage.libs.pari
1
sage: x = 333/106
sage: x.period()  # needs sage.libs.pari
13
sage: RealField(200)(x)  # needs sage.rings.real_mpfr
3.1415094339622641509433962264150943396226415094339622641509

(continues on next page)
0.166666666666667
>>> (Integer(1)/Integer(6)).period()
→ # needs sage.libs.pari
1
>>> x = Integer(333)/Integer(106)
>>> x.period() # needs sage.libs.pari
13
>>> RealField(Integer(200))(x) # needs sage.rings.real_mpfr
3.1415094339622641509433962264150943396226415094339622641509

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

>>> from sage.all import *

>>> QQ(Integer(3)/Integer(4)).prime_to_S_part()
3/4
>>> QQ(Integer(3)/Integer(4)).prime_to_S_part([Integer(2)])
3
>>> QQ(-Integer(3)/Integer(4)).prime_to_S_part([Integer(3)])
-1/4
>>> QQ(Integer(700)/Integer(99)).prime_to_S_part([Integer(2),Integer(3),
                                              -Integer(5)])
7/11
>>> QQ(-Integer(700)/Integer(99)).prime_to_S_part([Integer(2),Integer(3),
                                                -Integer(5)])
-7/11
>>> QQ(Integer(0)).prime_to_S_part([Integer(2),Integer(3),Integer(5)])
0

(continues on next page)
real()

Return the real part of self, which is self.

EXAMPLES:

```
sage: (1/2).real()
sage: 1/2
```

```
>>> from sage.all import *
```

```
>>> (Integer(1)/Integer(2)).real()
sage: 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()
sage: 6/5
sage: QQ(7/5).relative_norm()
sage: 7/5
```

```
>>> from sage.all import *
```

```
>>> (Integer(6)/Integer(5)).relative_norm()
sage: 6/5
>>> QQ(Integer(7)/Integer(5)).relative_norm()
sage: 7/5
```

round(mode=None)

Return the nearest integer to self, rounding away by default. Deprecation: in the future the default will be changed to rounding to even, 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()
doctest:...: DeprecationWarning: the default rounding for rationals, currently 'away', will be changed to 'even'. See https://github.com/sagemath/sage/issues/35473 for details.
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

from sage.all import *

>>> (Integer(9)/Integer(2)).round()
doctest:...: DeprecationWarning: the default rounding for rationals, currently 'away', will be changed to 'even'. See https://github.com/sagemath/sage/issues/35473 for details.
5
>>> n = Integer(4)/Integer(3); n.round()
1
>>> n = -Integer(17)/Integer(4); n.round()
-4
>>> n = -Integer(5)/Integer(2); n.round()
-3
>>> n.round("away")
-3
>>> n.round("up")
-2
>>> n.round("down")
-3
>>> n.round("even")
-2
>>> 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

(continues on next page)
sage: (-1/6).sign()
-1

```python
>>> from sage.all import *

>>> (Integer(2)/Integer(3)).sign()
1
>>> (Integer(0)/Integer(3)).sign()
0
>>> (-Integer(1)/Integer(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. 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:**

```python
sage: x = 25/9
sage: x.sqrt()
5/3
sage: sqrt(x)
5/3
sage: x = 64/4
sage: x.sqrt()
4
sage: x = 100/1
sage: x.sqrt()
10
sage: x = 25/9
sage: x.sqrt(all=True)
[10, -10]
```

*needs sage.symbolic*  
9*sqrt(1/5)

```python
sage: x = -81/3
sage: x.sqrt()
-3*sqrt(-3)
```

*needs sage.symbolic*  
3*sqrt(-3)
>>> x = Integer(100)/Integer(1)
>>> x.sqrt()
10
>>> x.sqrt(all=True)
[10, -10]
>>> x = Integer(81)/Integer(5)
>>> x.sqrt()  # needs sage.symbolic
9*sqrt(1/5)
>>> x = -Integer(81)/Integer(3)
>>> x.sqrt()  # needs sage.symbolic
3*sqrt(-3)

sage: n = 2/3
sage: n.sqrt()  # needs sage.symbolic
sqrt(2/3)

sage: # needs sage.rings.real_mpfr
sage: n.sqrt(prec=Integer(10))
0.82
sage: n.sqrt(prec=Integer(100))
0.81649658092772603273242802490
sage: n.sqrt(prec=Integer(100))**Integer(2)
0.66666666666666666666666666667
sage: n.sqrt(extend=False)
Traceback (most recent call last):
  ...
ValueError: square root of 2/3 not a rational number
sage: n.sqrt(extend=False, all=True)
[]

>>> from sage.all import *
>>> n = Integer(2)/Integer(3)
>>> n.sqrt()  # needs sage.symbolic
sqrt(2/3)

>>> # needs sage.rings.real_mpfr
>>> n.sqrt(prec=Integer(10))
0.82
>>> n.sqrt(prec=Integer(100))
0.81649658092772603273242802490
>>> n.sqrt(prec=Integer(100))**Integer(2)
0.66666666666666666666666666667
AUTHORS:

- Naqi Jaffery (2006-03-05): some examples

\textbf{squarefree_part ()}

Return the square free part of \( x \), i.e., an integer \( z \) such that \( x = zy^2 \), for a perfect square \( y^2 \).

\textbf{EXAMPLES:}

```python
sage: a = 1/2
sage: a.squarefree_part()
2
sage: b = a/a.squarefree_part()
```

```python
sage: b, b.is_square()
(1/4, True)
```

```python
sage: a = 24/5
sage: a.squarefree_part()
30
```

\textbf{str (base=10)}

Return a string representation of \texttt{self} in the given \texttt{base}.

\textbf{INPUT:}

- \texttt{base} – integer (default: 10); base must be between 2 and 36.

\textbf{OUTPUT:} string

\textbf{EXAMPLES:}
sage: (-4/17).str()
'{-4/17}'
sage: (-4/17).str(2)
'{-100/10001}'

>>> from sage.all import *

>>> (-Integer(4)/Integer(17)).str()
'{-4/17}'

>>> (-Integer(4)/Integer(17)).str(Integer(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.
Traceback (most recent call last):
...
ArithmeticError: Support of 0 not defined.

trace()

Return the trace from \(\mathbb{Q}\) to \(\mathbb{Q}\) of \(x\) (which is just \(x\)). This was added for compatibility with \texttt{NumberFields}.

OUTPUT:

- \texttt{Rational} – reference to \texttt{self}

EXAMPLES:

```
sage: \(1/3\).trace()
sage: \(1/3\)
```

```
>>> from sage.all import *
>>> \((\texttt{Integer}(1)/\texttt{Integer}(3)).trace())
1/3
```

AUTHORS:

- Craig Citro

trunc()

Round this rational number to the nearest integer toward zero.

EXAMPLES:

```
sage: \(5/3\).trunc()
sage: 1
sage: \((-5/3).trunc()
sage: -1
sage: \(\texttt{QQ}(42).trunc()

sage: 42
sage: \(\texttt{QQ}(-42).trunc()

sage: -42
```

```
>>> from sage.all import *

>>> \((\texttt{Integer}(5)/\texttt{Integer}(3)).trunc()

1
```

```
>>> \((-\texttt{Integer}(5)/\texttt{Integer}(3)).trunc()

-1
```

```
>>> \(\texttt{QQ}(\texttt{Integer}(42)).trunc()

42
```

```
>>> \(\texttt{QQ}(\texttt{-Integer}(42)).trunc()

-42
```

val_unit \((p)\)

Return a pair: the \(p\)-adic valuation of \texttt{self}, and the \(p\)-adic unit of \texttt{self}, as a \texttt{Rational}.

We do not require the \(p\) be prime, but it must be at least 2. For more documentation see \texttt{Integer}. \texttt{val_unit()}.

INPUT:

- \(p\) – a prime

OUTPUT:
• int – the $p$-adic valuation of this rational
• Rational – $p$-adic unit part of self

EXAMPLES:

```sage
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)
```

```python
>>> from sage.all import *

>>> (-Integer(4)/Integer(17)).val_unit(Integer(2))
(2, -1/17)

>>> (-Integer(4)/Integer(17)).val_unit(Integer(17))
(-1, -4)

>>> (Integer(0)/Integer(1)).val_unit(Integer(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
sage: x = -5/9
sage: x.valuation(5)
1
sage: x.ord(5)
1
sage: x.valuation(3)
-2
sage: x.valuation(2)
0
```

```python
>>> from sage.all import *

>>> x = -Integer(5)/Integer(9)

>>> x.valuation(Integer(5))
1

>>> x.ord(Integer(5))
1
```

(continues on next page)
Some edge cases:

```python
sage: (0/1).valuation(4)
+Infinity
sage: (7/16).valuation(4)
-2
```

```python
>>> from sage.all import *
```

```python
>>> (Integer(0)/Integer(1)).valuation(Integer(4))
+Infinity
>>> (Integer(7)/Integer(16)).valuation(Integer(4))
-2
```

class sage.rings.rational.Z_to_Q

Bases: Morphism

A morphism from \( \mathbb{Z} \) to \( \mathbb{Q} \).

**is_surjective()**

Return whether this morphism is surjective.

**EXAMPLES:**

```python
sage: QQ.coerce_map_from(ZZ).is_surjective()
False
```

```python
>>> from sage.all import *
```

```python
>>> QQ.coerce_map_from(ZZ).is_surjective()
False
```

**section()**

Return a section of this morphism.

**EXAMPLES:**

```python
sage: f = QQ.coerce_map_from(ZZ).section(); f
Generic map:
    From: Rational Field
    To:   Integer Ring
```

```python
>>> from sage.all import *
```

```python
>>> 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 Issue #15618):

```python
sage: f.parent()
Set of Morphisms from Rational Field to Integer Ring
in Category of sets with partial maps
```
>>> from sage.all import *
>>> 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: Morphism
     A morphism from Python 3 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:

```python
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
```
>>> integer_rational_power(Integer(27), Integer(1)/Integer(3))
3
>>> integer_rational_power(-Integer(27), Integer(1)/Integer(3)) is None
True
>>> integer_rational_power(-Integer(27), Integer(2)/Integer(3)) is None
True
>>> integer_rational_power(Integer(512), Integer(7)/Integer(9))
128
>>> integer_rational_power(Integer(27), Integer(1)/Integer(4)) is None
True
>>> integer_rational_power(-Integer(16), Integer(1)/Integer(4)) is None
True
>>> integer_rational_power(Integer(0), Integer(7)/Integer(9))
0
>>> integer_rational_power(Integer(0), Integer(7)/Integer(9))
1
>>> integer_rational_power(-Integer(1), Integer(7)/Integer(9)) is None
True
>>> integer_rational_power(-Integer(1), Integer(8)/Integer(9)) is None
True
>>> integer_rational_power(-Integer(1), Integer(9)/Integer(8)) is None
True

TESTS (Issue #11228):

sage: integer_rational_power(-10, QQ(2))
100
sage: integer_rational_power(0, QQ(0))
1

>>> from sage.all import *

>>> integer_rational_power(-Integer(10), QQ(Integer(2)))
100
>>> integer_rational_power(Integer(0), QQ(Integer(0)))
1

sage.rings.rational.is_Rational(x)

Return True if x is of the Sage Rational type.

EXAMPLES:

sage: from sage.rings.rational import is_Rational
sage: is_Rational(2)
doctest:warning...
DeprecationWarning: The function is_Rational is deprecated; use 'isinstance(..., Rational)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
False
sage: is_Rational(2/1)
True
sage: is_Rational(int(2))
False
sage: is_Rational('5')
False
from sage.all import *
from sage.rings.rational import is_Rational
is_Rational(Integer(2))
doctest:warning...
DeprecationWarning: The function is_Rational is deprecated; use 'isinstance(..., Rational)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
False
is_Rational(Integer(2)/Integer(1))
True
is_Rational(int(Integer(2)))
False
is_Rational('5')
False

sage.rings.rational.make_rational(s)
Make a rational number from s (a string in base 32)

INPUT:
• s — string in base 32

OUTPUT: Rational

EXAMPLES:
sage: (-7/15).str(32)
'-7/f'
sage: sage.rings.rational.make_rational('-7/f')
-7/15

sage.rings.rational.rational_power_parts(a, b, factor_limit=100000)
Compute rationals or integers c and d such that \(a^b = c \cdot d^b\) 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)

(continues on next page)
```python
sage: rational_power_parts(9/1000, 1/2)
(3/10, 1/10)
```

```python
>>> from sage.all import *
>>> from sage.rings.rational import rational_power_parts
>>> rational_power_parts(Integer(27), Integer(1)/Integer(2))
(3, 3)
>>> rational_power_parts(-Integer(128), Integer(3)/Integer(4))
(8, -8)
>>> rational_power_parts(-Integer(4), Integer(1)/Integer(2))
(2, -1)
>>> rational_power_parts(-Integer(4), Integer(1)/Integer(3))
(1, -4)
>>> rational_power_parts(Integer(9)/Integer(1000), Integer(1)/Integer(2))
(3/10, 1/10)
```
CHAPTER
THREE

INDICES AND TABLES

• Index
• Module Index
• Search Page
Python Module Index

a
sage.arith.functions, 121
sage.arith.misc, 129
sage.arith.multi_modular, 124
sage.arith.power, 123

r
sage.rings.bernmm, 104
sage.rings.bernoulli_mod_p, 106
sage.rings.factorint, 110
sage.rings.factorint_flint, 114
sage.rings.factorint_pari, 115
sage.rings.fast_arith, 116
sage.rings.integer, 21
sage.rings.integer_ring, 1
sage.rings.rational, 272
sage.rings.rational_field, 249
sage.rings.sum_of_squares, 118
Non-alphabetical

__pow__() (sage.rings.integer.Integer method), 26

A

absolute_degree() (sage.rings.integer_ring.IntegerRing_class method), 7
absolute_degree() (sage.rings.rational_field.RationalField method), 252
absolute_discriminant() (sage.rings.rational_field.RationalField method), 252
absolute_norm() (sage.rings.rational.Rational method), 275
absolute_polynomial() (sage.rings.rational_field.RationalField method), 252
additive_order() (sage.rings.integer.Integer method), 28
additive_order() (sage.rings.rational.Rational method), 275
algdep() (in module sage.arith.misc), 147
algebraic_closure() (sage.rings.rational_field.RationalField method), 253
algebraic_dependency() (in module sage.arith.misc), 150
arith_int (class in sage.rings.fast_arith), 116
arith_llong (class in sage.rings.fast_arith), 116
as_integer_ratio() (sage.rings.integer.Integer method), 28
as_integer_ratio() (sage.rings.rational.Rational method), 276
aurifeuillian() (in module sage.rings.factorint), 110
automorphisms() (sage.rings.rational_field.RationalField method), 253

B

balanced_digits() (sage.rings.integer.Integer method), 28
bernmm_bern_modp() (in module sage.rings.bernoulli_mod_p), 106
bernoulli_mod_p_single() (in module sage.rings.bernoulli_mod_p), 107
binary() (sage.rings.integer.Integer method), 30
binomial() (in module sage.arith.misc), 155
binomial() (sage.rings.integer.Integer method), 30
binomial_coefficients() (in module sage.arith.misc), 158
bit_length() (sage.rings.integer.Integer method), 31
bits() (sage.rings.integer.Integer method), 32

C

carmichael_lambda() (in module sage.arith.misc), 159
ceil() (sage.rings.integer.Integer method), 32
ceil() (sage.rings.rational.Rational method), 276
characteristic() (sage.rings.integer_ring.IntegerRing_class method), 7
characteristic() (sage.rings.rational_field.RationalField method), 253
charpoly() (sage.rings.rational.Rational method), 276
class_number() (sage.rings.integer.Integer method), 33
class_number() (sage.rings.rational_field.RationalField method), 254
completion() (sage.rings.integer_ring.IntegerRing_class method), 8
completion() (sage.rings.rational_field.RationalField method), 254
complex_embedding() (sage.rings.rational_field.RationalField method), 254
conjugate() (sage.rings.integer.Integer method), 34
conjugate() (sage.rings.rational.Rational method), 277
construction() (sage.rings.rational_field.RationalField method), 255
content() (sage.rings.rational.Rational method), 277
continuant() (in module sage.arith.misc), 161
continued_fraction() (sage.rings.rational.Rational method), 278
continued_fraction_list() (sage.rings.rational.Rational method), 278

coprime_integers() (sage.rings.integer.Integer method), 34

CRT() (in module sage.arith.misc), 129
crt() (in module sage.arith.misc), 163
crt() (sage.arith.multi_modular.MultiModularBasis_base method), 125
crt() (sage.rings.integer.Integer method), 35

CRT_basis() (in module sage.arith.misc), 133
crt_basis() (in module sage.rings.integer_ring), 19

CRT_list() (in module sage.arith.misc), 134

CRT_vectors() (in module sage.arith.misc), 135

dedekind_psi() (in module sage.arith.misc), 167
dedekind_sum() (in module sage.arith.misc), 167
defining_polynomial() (sage.rings.rational_field.RationalField method), 255
degree() (sage.rings.integer_ring.IntegerRing_class method), 8
degree() (sage.rings.rational_field.RationalField method), 255
denom() (sage.rings.rational.Rational method), 280
denominator() (sage.rings.integer.Integer method), 35
denominator() (sage.rings.rational.Rational method), 281
differences() (in module sage.arith.misc), 170
digits() (sage.rings.integer.Integer method), 35
discriminant() (sage.rings.rational_field.RationalField method), 255
divide_knowing_divisible_by() (sage.rings.integer.Integer method), 39
divides() (sage.rings.integer.Integer method), 39
divisors() (in module sage.arith.misc), 171
divisors() (sage.rings.integer.Integer method), 40

eddings() (sage.rings.rational_field.RationalField method), 256

eratosthenes() (in module sage.arith.misc), 173
euclidean_degree() (sage.rings.integer.Integer method), 42

Euler_Phi (class in sage.arith.misc), 136

exact_log() (sage.rings.integer.Integer method), 42

exp() (sage.rings.integer.Integer method), 44

extend_with_primes() (sage.arith.multi_modular.MultiModularBasis_base method), 126

extension() (sage.rings.integer_ring.IntegerRing_class method), 8

extension() (sage.rings.rational_field.RationalField method), 256

F

factor() (in module sage.arith.misc), 173

factor() (sage.rings.integer.Integer method), 45

factor() (sage.rings.rational.Rational method), 281

factor_aurifeuillian() (in module sage.rings.factorint), 112

factor_cunningham() (in module sage.rings.factorint), 113

factor_trial_division() (in module sage.rings.factorint), 113

factor_using_flint() (in module sage.rings.factorint_flint), 114

factor_using_pari() (in module sage.rings.factorint_pari), 115

factorial() (in module sage.arith.misc), 178

factorial() (sage.rings.integer.Integer method), 48

falling_factorial() (in module sage.arith.misc), 179

floor() (sage.rings.integer.Integer method), 49

floor() (sage.rings.rational.Rational method), 282

four_squares() (in module sage.arith.misc), 181

four_squares_pyx() (in module sage.rings.sum_of_squares), 118

frac() (in module sage.rings.rational_field), 271

fraction_field() (sage.rings.rational_field.RationalField method), 9

free_integer_pool() (in module sage.rings.integer), 102

from_bytes() (sage.rings.integer_ring.IntegerRing_class method), 9

fundamental_discriminant() (in module sage.arith.misc), 182

G

gamma() (sage.rings.integer.Integer method), 49

gamma() (sage.rings.rational.Rational method), 282

gauss_sum() (in module sage.arith.misc), 183

GCD() (in module sage.arith.misc), 139

gcd() (in module sage.arith.misc), 184

gcd_int() (sage.rings.fast_arith.int_method), 116

GCD_list() (in module sage.rings.integer), 23

gcd_longlong() (sage.rings.fast_arith.longlong_method), 116

gen() (sage.rings.integer_ring.IntegerRing_class method), 10

gen() (sage.rings.rational_field.RationalField method), 257

generic_power() (in module sage.arith.power), 123

gens() (sage.rings.integer_ring.IntegerRing_class method), 10

gens() (sage.rings.rational_field.RationalField method), 257

gcd() (in module sage.arith.misc), 185

gcd_inverse_mod() (in module sage.arith.misc), 186
global_height() (sage.rings.integer.Integer method), 49
global_height() (sage.rings.rational.Rational method), 283
global_height_arch() (sage.rings.rational.Rational method), 284
global_height_non_arch() (sage.rings.rational.Rational method), 284

H
height() (sage.rings.rational.Rational method), 285
hex() (sage.rings.integer.Integer method), 50
hilbert_conductor() (in module sage.arith.misc), 186
hilbert_conductor_inverse() (in module sage.arith.misc), 187
hilbert_symbol() (in module sage.arith.misc), 188
hilbert_symbol_negative_at_S() (sage.rings.rational_field.RationalField method), 257

I
imag() (sage.rings.integer.Integer method), 50
imag() (sage.rings.rational.Rational method), 286
int_to_Q (class in sage.rings.rational), 313
int_to_Z (class in sage.rings.integer), 102
Integer (class in sage.rings.integer), 24
integer_ceil() (in module sage.arith.misc), 190
integer_floor() (in module sage.arith.misc), 191
integer_rational_power() (in module sage.rings.integer), 313
integer_trunc() (in module sage.arith.misc), 192
IntegerRing() (in module sage.rings.integer_ring), 2
IntegerRing_class (class in sage.rings.integer_ring), 2
IntegerWrapper (class in sage.rings.integer), 102
inverse_mod() (in module sage.arith.misc), 192
inverse_mod() (sage.rings.integer.Integer method), 51
inverse_mod_int() (sage.rings.fast_arith.fast_arith_int method), 116
inverse_mod_longlong() (sage.rings.fast_arith.fast_arith_long method), 116
inverse_of_unit() (sage.rings.integer.Integer method), 52
is_absolute() (sage.rings.rational_field.RationalField method), 258
is_discriminant() (sage.rings.integer.Integer method), 52
is_field() (sage.rings.integer_ring.IntegerRing_class method), 11
is_fundamental_discriminant() (sage.rings.integer.Integer method), 53
is_Integer() (in module sage.rings.integer), 103
is_squarefree() (in module sage.arith.misc), 201
is_squarefree() (sage.rings.integer.Integer method), 64
is_sum_of_two_squares_pxy() (in module sage.rings.sum_of_squares), 118
is_surjective() (sage.rings.rational.Z_to_Q method), 312
is_unit() (sage.rings.integer.Integer method), 65
isqrt() (sage.rings.integer.Integer method), 65
J
jacobi() (sage.rings.integer.Integer method), 65
jacobi_symbol() (in module sage.arith.misc), 203
K
kronecker() (in module sage.arith.misc), 204
kronecker() (sage.rings.integer.Integer method), 66
kronecker_symbol() (in module sage.arith.misc), 205
krull_dimension() (sage.rings.integer_ring.IntegerRing_class method), 11
L
lcm() (in module sage.arith.functions), 122
LCM_list() (in module sage.arith.functions), 121
legendre_symbol() (in module sage.arith.functions), 206
list() (sage.arith.multi_modular.MultiModularBasis_base method), 126
list() (sage.rings.integer.Integer method), 67
list() (sage.rings.rational.Rational method), 294
local_height() (sage.rings.rational.Rational method), 294
local_height_arch() (sage.rings.rational.Rational method), 295
log() (sage.rings.integer.Integer method), 67
log() (sage.rings.rational.Rational method), 295
M
make_integer() (in module sage.rings.integer), 103
make_rational() (in module sage.rings.rational), 315
maximal_order() (sage.rings.rational_field.RationalField method), 259
minpoly() (sage.rings.rational.Rational method), 297
mod_ui() (sage.rings.rational.Rational method), 297
module
sage.arith.functions, 121
sage.arith.misc, 129
sage.arith.multi_modular, 124
sage.arith.power, 123
sage.rings.bernmm, 104
sage.rings.bernoulli_mod_p, 106
sage.rings.factorint, 110
sage.rings.factorint_fflint, 114

sage.rings.factorint_pari, 115
sage.rings.fast_arith, 116
sage.rings.integer, 21
sage.rings.integer_ring, 1
sage.rings.rational, 272
sage.rings.rational_field, 249
sage.rings.sum_of_squares, 118
Moebius (class in sage.arith.misc), 140
mqrr_rational_reconstruction() (in module sage.arith.misc), 207
multifactorial() (sage.rings.integer.Integer method), 70
MultiModularBasis (class in sage.arith.multi_modular), 124
MultiModularBasis_base (class in sage.arith.multi_modular), 124
multinomial() (in module sage.arith.misc), 208
multinomial_coefficients() (in module sage.arith.misc), 209
multiplicative_order() (sage.rings.integer.Integer method), 71
multiplicative_order() (sage.rings.rational.Rational method), 298
MutableMultiModularBasis (class in sage.arith.multi_modular), 127
N
nbits() (sage.rings.integer.Integer method), 71
ndigits() (sage.rings.integer.Integer method), 71
next_prime() (in module sage.arith.misc), 211
next_prime() (sage.arith.multi_modular.MutableMultiModularBasis method), 127
next_prime() (sage.rings.integer.Integer method), 72
next_prime_power() (in module sage.arith.misc), 212
next_prime_power() (sage.rings.integer.Integer method), 73
next_probable_prime() (in module sage.arith.misc), 213
next_probable_prime() (sage.rings.integer.Integer method), 74
ngens() (sage.rings.integer_ring.IntegerRing_class method), 12
ngens() (sage.rings.rational_field.RationalField method), 259
norm() (sage.rings.rational.Rational method), 298
nth_prime() (in module sage.arith.misc), 213
nth_root() (sage.rings.integer.Integer method), 75
nth_root() (sage.rings.rational.Rational method), 299
number_field() (sage.rings.rational_field.RationalField method), 259
number_of_divisors() (in module sage.arith.misc), 214
tuple() (sage.rings.rational.Rational method), 300
<table>
<thead>
<tr>
<th>Function</th>
<th>Module/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>numerator()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>77</td>
</tr>
<tr>
<td>numerator()</td>
<td>(sage.rings.rational.Rational method)</td>
<td>300</td>
</tr>
<tr>
<td>oct()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>78</td>
</tr>
<tr>
<td>odd_part()</td>
<td>(in module sage.arith.misc)</td>
<td>215</td>
</tr>
<tr>
<td>odd_part()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>79</td>
</tr>
<tr>
<td>ord()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>79</td>
</tr>
<tr>
<td>ord()</td>
<td>(sage.rings.rational.Integer method)</td>
<td>301</td>
</tr>
<tr>
<td>order()</td>
<td>(sage.rings.integer_ring.IntegerRing_class method)</td>
<td>12</td>
</tr>
<tr>
<td>order()</td>
<td>(sage.rings.rational_field.RationalField method)</td>
<td>259</td>
</tr>
<tr>
<td>ordinal_str()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>80</td>
</tr>
<tr>
<td>p_primary_part()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>81</td>
</tr>
<tr>
<td>parameter()</td>
<td>(sage.rings.integer_ring.IntegerRing_class method)</td>
<td>12</td>
</tr>
<tr>
<td>partial_product()</td>
<td>(sage.arith.multi_modular.MultiModularBasis_base method)</td>
<td>126</td>
</tr>
<tr>
<td>perfect_power()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>82</td>
</tr>
<tr>
<td>period()</td>
<td>(sage.rings.rational.Rational method)</td>
<td>302</td>
</tr>
<tr>
<td>places()</td>
<td>(sage.rings.rational_field.RationalField method)</td>
<td>260</td>
</tr>
<tr>
<td>plot()</td>
<td>(sage.arith.misc.Euler_Phi method)</td>
<td>138</td>
</tr>
<tr>
<td>plot()</td>
<td>(sage.arith.misc.Moebius method)</td>
<td>142</td>
</tr>
<tr>
<td>plot()</td>
<td>(sage.arith.misc.Sigma method)</td>
<td>144</td>
</tr>
<tr>
<td>polynomial()</td>
<td>(sage.rings.rational_field.RationalField method)</td>
<td>260</td>
</tr>
<tr>
<td>popcount()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>83</td>
</tr>
<tr>
<td>power_basis()</td>
<td>(sage.rings.rational_field.RationalField method)</td>
<td>261</td>
</tr>
<tr>
<td>power_mod()</td>
<td>(in module sage.arith.misc)</td>
<td>216</td>
</tr>
<tr>
<td>powermod()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>83</td>
</tr>
<tr>
<td>precomputation_list()</td>
<td>(sage.arith.multi_modular.MultiModularBasis_base method)</td>
<td>127</td>
</tr>
<tr>
<td>previous_prime()</td>
<td>(in module sage.arith.misc)</td>
<td>216</td>
</tr>
<tr>
<td>previous_prime()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>84</td>
</tr>
<tr>
<td>previous_prime_power()</td>
<td>(in module sage.arith.misc)</td>
<td>217</td>
</tr>
<tr>
<td>previous_prime_power()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>85</td>
</tr>
<tr>
<td>prime_divisors()</td>
<td>(in module sage.arith.misc)</td>
<td>219</td>
</tr>
<tr>
<td>prime_divisors()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>86</td>
</tr>
<tr>
<td>prime_factors()</td>
<td>(in module sage.arith.misc)</td>
<td>220</td>
</tr>
<tr>
<td>prime_factors()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>87</td>
</tr>
<tr>
<td>prime_powers()</td>
<td>(in module sage.arith.misc)</td>
<td>222</td>
</tr>
<tr>
<td>prime_range()</td>
<td>(in module sage.rings.fast_arith)</td>
<td>116</td>
</tr>
<tr>
<td>prime_to_m_part()</td>
<td>(module sage.arith.misc)</td>
<td>223</td>
</tr>
<tr>
<td>prime_to_m_part()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>88</td>
</tr>
<tr>
<td>prime_to_S_part()</td>
<td>(sage.rings.rational.Integer method)</td>
<td>303</td>
</tr>
<tr>
<td>primes()</td>
<td>(in module sage.arith.misc)</td>
<td>224</td>
</tr>
<tr>
<td>primes_first_n()</td>
<td>(in module sage.arith.misc)</td>
<td>225</td>
</tr>
<tr>
<td>primes_of_bounded_norm_iter()</td>
<td>(sage.rings.rational_field.RationalField method)</td>
<td>261</td>
</tr>
<tr>
<td>primitive_root()</td>
<td>(in module sage.arith.misc)</td>
<td>226</td>
</tr>
<tr>
<td>prod()</td>
<td>(sage.arith.multi_modular.MultiModularBasis_base method)</td>
<td>127</td>
</tr>
<tr>
<td>Python Enhancement Proposals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PEP 3127, 25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>quadratic_defect()</td>
<td>(sage.rings.rational_field.RationalField method)</td>
<td>262</td>
</tr>
<tr>
<td>quadratic_residues()</td>
<td>(in module sage.arith.misc)</td>
<td>227</td>
</tr>
<tr>
<td>quo_rem()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>89</td>
</tr>
<tr>
<td>quotient()</td>
<td>(sage.rings.integer_ring.IntegerRing_class method)</td>
<td>12</td>
</tr>
<tr>
<td>radical()</td>
<td>(in module sage.arith.misc)</td>
<td>228</td>
</tr>
<tr>
<td>random_element()</td>
<td>(sage.rings.integer_ring.IntegerRing_class method)</td>
<td>13</td>
</tr>
<tr>
<td>random_element()</td>
<td>(sage.rings.rational_field.RationalField method)</td>
<td>262</td>
</tr>
<tr>
<td>random_prime()</td>
<td>(in module sage.arith.misc)</td>
<td>229</td>
</tr>
<tr>
<td>range()</td>
<td>(sage.arith.misc.Moebius method)</td>
<td>142</td>
</tr>
<tr>
<td>range()</td>
<td>(sage.rings.integer_ring.IntegerRing_class method)</td>
<td>16</td>
</tr>
<tr>
<td>range_by_height()</td>
<td>(sage.rings.rational_field.RationalField method)</td>
<td>263</td>
</tr>
<tr>
<td>Rational (class in sage.rings.rational)</td>
<td></td>
<td>272</td>
</tr>
<tr>
<td>rational_power_parts()</td>
<td>(in module sage.rings.rational)</td>
<td>315</td>
</tr>
<tr>
<td>rational_recon_int()</td>
<td>(sage.rings.fast_arith.arith_int method)</td>
<td>116</td>
</tr>
<tr>
<td>rational_recon_longlong()</td>
<td>(sage.rings.fast_arith.arith_long method)</td>
<td>116</td>
</tr>
<tr>
<td>rational_reconstruction()</td>
<td>(in module sage.arith.misc)</td>
<td>230</td>
</tr>
<tr>
<td>rational_reconstruction()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>90</td>
</tr>
<tr>
<td>RationalField (class in sage.rings.rational_field)</td>
<td></td>
<td>250</td>
</tr>
<tr>
<td>real()</td>
<td>(sage.rings.integer.Integer method)</td>
<td>90</td>
</tr>
<tr>
<td>real()</td>
<td>(sage.rings.rational.Rational method)</td>
<td>304</td>
</tr>
</tbody>
</table>
relative_discriminant() (sage.rings.rational_field.RationalField method), 264
relative_norm() (sage.rings.rational.Rational method), 304
replace_prime() (sage.arith.multi_modular.MutableMultiModularBasis method), 128
residue_field() (sage.rings.integer_ring.IntegerRing_class method), 17
residue_field() (sage.rings.rational_field.RationalField method), 264
rising_factorial() (in module sage.arith.misc), 232
round() (sage.rings.integer.Integer method), 90
round() (sage.rings.rational.Rational method), 304
sage.arith.functions module, 121
sage.arith.misc module, 129
sage.arith.multi_modular module, 124
sage.arith.power module, 123
sage.rings.bernmm module, 104
sage.rings.bernoulli_mod_p module, 106
sage.rings.factorint module, 110
sage.rings.factorint_flint module, 114
sage.rings.factorint_pari module, 115
sage.rings.fast_arith module, 116
sage.rings.integer module, 21
sage.rings.integer_ring module, 1
sage.rings.rational module, 272
sage.rings.rational_field module, 249
sage.rings.sum_of_squares module, 118
section() (sage.rings.rational.Q_to_Z method), 272
section() (sage.rings.rational.Z_to_Q method), 312
selmer_generators() (sage.rings.rational_field.RationalField method), 265
selmer_group() (sage.rings.rational_field.RationalField method), 266
selmer_group_iterator() (sage.rings.rational_field.RationalField method), 266
selmer_space() (sage.rings.rational_field.RationalField method), 267
Sigma (class in sage.arith.misc), 143
sign() (sage.rings.integer.Integer method), 91
sign() (sage.rings.rational.Rational method), 305
signature() (sage.rings.rational_field.RationalField method), 269
some_elements() (sage.rings.rational_field.RationalField method), 269
sort_complex_numbers_for_display() (in module sage.arith.misc), 233
sqrt() (sage.rings.integer.Integer method), 91
sqrt() (sage.rings.rational.Rational method), 306
sqrtrim() (sage.rings.integer.Integer method), 92
squarefree_divisors() (in module sage.arith.misc), 235
squarefree_part() (sage.rings.integer.Integer method), 93
squarefree_part() (sage.rings.rational.Rational method), 308
str() (sage.rings.integer.Integer method), 94
str() (sage.rings.rational.Rational method), 308
subfactorial() (in module sage.arith.misc), 236
sum_of_k_squares() (in module sage.arith.misc), 236
support() (sage.rings.integer.Integer method), 95
support() (sage.rings.rational.Rational method), 309

test_bit() (sage.rings.integer.Integer method), 96
three_squares() (in module sage.arith.misc), 239
three_squares_pyx() (in module sage.rings.sum_of_squares), 119
to_bytes() (sage.rings.integer.Integer method), 97
trace() (sage.rings.rational.Rational method), 310
trailing_zero_bits() (sage.rings.integer.Integer method), 97
trial_division() (in module sage.arith.misc), 240
trial_division() (sage.rings.integer.Integer method), 98
trunc() (sage.rings.integer.Integer method), 100
trunc() (sage.rings.rational.Rational method), 310
two_squares() (in module sage.arith.misc), 241
two_squares_pyx() (in module sage.rings.sum_of_squares), 120

val_unit() (sage.rings.integer.Integer method), 100
val_unit() (sage.rings.rational.Rational method), 310
valuation() (in module sage.arith.misc), 242
valuation() (sage.rings.integer_ring.IntegerRing_class method), 18
valuation() (sage.rings.integer.Integer method), 101
valuation() (sage.rings.rational_field.RationalField method), 270
valuation() (sage.rings.rational.Rational method), 311
verify_bernoulli_mod_p() (in module sage.rings.bernoulli_mod_p), 109

X
XGCD () (in module sage.arith.misc), 144
xgcd() (in module sage.arith.misc), 244
xgcd() (sage.rings.integer.Integer method), 102
xgcd_int() (sage.rings.fast_arith.arith_int method), 116
xkcd() (in module sage.arith.misc), 247
xlcm() (in module sage.arith.misc), 247

Z
Z_to_Q (class in sage.rings.rational), 312
zeta() (sage.rings.integer_ring.IntegerRing_class method), 19
zeta() (sage.rings.rational_field.RationalField method), 270