Standard Commutative Rings

Release 10.2

The Sage Development Team

Dec 06, 2023
## CONTENTS

1. Integers .......................... 1
2. Rationals ......................... 157
3. Indices and Tables .............. 201
   Python Module Index ............ 203
   Index ................................ 205
1.1 Ring \( \mathbb{Z} \) of Integers

The \texttt{IntegerRing} class represents the ring \( \mathbb{Z} \) of (arbitrary precision) integers. Each integer is an instance of \texttt{Integer}, which is defined in a Pyrex extension module that wraps GMP integers (the \texttt{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 \texttt{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
```

```python
sage.rings.integer_ring.IntegerRing()
Return the integer ring.

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

```python
class sage.rings.integer_ring.IntegerRing_class
    Bases: PrincipalIdealDomain
    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.

\begin{verbatim}
sage: Z = IntegerRing(); Z
Integer Ring
sage: Z.characteristic()
0
sage: Z.is_field()
False
sage: Z.category()
Join of Category of euclidean domains
and Category of infinite enumerated sets
and Category of metric spaces
sage: Z(2^(2^5) + 1)
4294967297
\end{verbatim}

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

\begin{verbatim}
sage: parent('37')
<... 'str'>
sage: parent(Z('37'))
Integer Ring
sage: Z('0x10')
16
sage: Z('0x1a')
26
sage: Z('0o20')
16
\end{verbatim}

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 $\text{base}^i$:

\begin{verbatim}
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
\end{verbatim}

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

\begin{verbatim}
sage: Z("sage", base=32)
928270
sage: Z("SAGE", base=32)
928270
sage: Z("Sage", base=32)
928270
sage: Z([14, 16, 28], base=32)
\end{verbatim}
We next illustrate basic arithmetic in $\mathbb{Z}$:

<table>
<thead>
<tr>
<th>sage:</th>
<th>a = Z(1234); a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1234</td>
</tr>
<tr>
<td>sage:</td>
<td>b = Z(5678); b</td>
</tr>
<tr>
<td></td>
<td>5678</td>
</tr>
<tr>
<td>sage:</td>
<td>type(a)</td>
</tr>
<tr>
<td></td>
<td>&lt;class 'sage.rings.integer.Integer'&gt;</td>
</tr>
<tr>
<td>sage:</td>
<td>a + b</td>
</tr>
<tr>
<td></td>
<td>6912</td>
</tr>
<tr>
<td>sage:</td>
<td>b + a</td>
</tr>
<tr>
<td></td>
<td>6912</td>
</tr>
<tr>
<td>sage:</td>
<td>a * b</td>
</tr>
<tr>
<td></td>
<td>7006652</td>
</tr>
<tr>
<td>sage:</td>
<td>b * a</td>
</tr>
<tr>
<td></td>
<td>7006652</td>
</tr>
<tr>
<td>sage:</td>
<td>a - b</td>
</tr>
<tr>
<td></td>
<td>-4444</td>
</tr>
<tr>
<td>sage:</td>
<td>b - a</td>
</tr>
<tr>
<td></td>
<td>4444</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>sage:</th>
<th>a / b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>617/2839</td>
</tr>
<tr>
<td>sage:</td>
<td>type(a/b)</td>
</tr>
<tr>
<td></td>
<td>&lt;class 'sage.rings.rational.Rational'&gt;</td>
</tr>
<tr>
<td>sage:</td>
<td>a/a</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>sage:</td>
<td>type(a/a)</td>
</tr>
<tr>
<td></td>
<td>&lt;class 'sage.rings.rational.Rational'&gt;</td>
</tr>
</tbody>
</table>

For floor division, use the $//$ operator instead:

<table>
<thead>
<tr>
<th>sage:</th>
<th>a // b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>sage:</td>
<td>type(a//b)</td>
</tr>
<tr>
<td></td>
<td>&lt;class 'sage.rings.integer.Integer'&gt;</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>sage:</th>
<th>a + 17</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1251</td>
</tr>
<tr>
<td>sage:</td>
<td>a * 374</td>
</tr>
<tr>
<td></td>
<td>461516</td>
</tr>
<tr>
<td>sage:</td>
<td>374 * a</td>
</tr>
<tr>
<td></td>
<td>461516</td>
</tr>
<tr>
<td>sage:</td>
<td>a/19</td>
</tr>
</tbody>
</table>

(continues on next page)
Integers can be coerced:

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

We can create integers from several types of objects:

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

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

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

```
sage: b = Z([4,1,7], base=100)
sage: b
70104
sage: b.digits(base=71)
[27, 64, 13]
sage: Z(15).digits(2)
[1, 1, 1, 1]
sage: Z(15).digits(3)
[0, 2, 1]
```

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

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

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

**absolute_degree()**

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

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

**EXAMPLES:**
characteristic()  
Return the characteristic of the integers, which is 0.

EXAMPLES:

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

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

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

INPUT:

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

OUTPUT:

- The completion of \(\mathbb{Z}\) 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
```  

degree()

Return the degree of the integers, which is 1.

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

EXAMPLES:

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

extension\((poly, names, **kwds)\)

Return the order generated by the specified list of polynomials.

INPUT:

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

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.

1.1. Ring \(\mathbb{Z}\) of Integers
EXAMPLES:

\begin{verbatim}
sage: x = polygen(ZZ, 'x')
sage: ZZ.extension(x^2 - 5, 'a')
# needs sage.rings.number_field
Order 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 in Number Field in a with defining polynomial x^2 + 1 over its base field
\end{verbatim}

\textbf{fraction_field()}

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

EXAMPLES:

\begin{verbatim}
sage: ZZ.fraction_field()
Rational Field
sage: ZZ.fraction_field() == QQ
True
\end{verbatim}

\textbf{gen(n=0)}

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

INPUT:

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

EXAMPLES:

\begin{verbatim}
sage: ZZ.gen()
1
sage: type(ZZ.gen())
<class 'sage.rings.integer.Integer'>
\end{verbatim}

\textbf{gens()}

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

EXAMPLES:

\begin{verbatim}
sage: ZZ.gens(); ZZ.gens()[0]
(1,)
1
sage: type(ZZ.gens()[0])
<class 'sage.rings.integer.Integer'>
\end{verbatim}

\textbf{is_field(proof=True)}

Return \texttt{False} since the integers are not a field.

EXAMPLES:

\begin{verbatim}
sage: ZZ.is_field()
False
\end{verbatim}
**is_integrally_closed()**
Return that the integer ring is, in fact, integrally closed.

EXAMPLES:

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

**is_noetherian()**
Return True since the integers are a Noetherian ring.

EXAMPLES:

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

**krull_dimension()**
Return the Krull dimension of the integers, which is 1.

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

random_element(x=None, y=None, distribution=None)

Return a random integer.

INPUT:

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

OUTPUT:

• With no input, return a random integer.
  If only one integer x is given, return an integer between 0 and x – 1.
  If two integers are given, return an integer between x and y – 1 inclusive.
If at least one bound is given, the default distribution is the uniform distribution; otherwise, it is the
distribution described below.
If the distribution '1/n' is specified, the bounds are ignored.
If the distribution 'mpz_rrandomb' is specified, the output is in the range from 0 to 2^x – 1.
If the distribution 'gaussian' is specified, the output is sampled from a discrete Gaussian distribution
with parameter \( \sigma = x \) and centered at zero. That is, the integer \( v \) is returned with probability proportional
to \( \exp(-v^2/(2\sigma^2)) \). See sage.stats.distributions.discrete_gaussian_integer for details. Note that if many samples from the same discrete Gaussian distribution are
needed, it is faster to construct a sage.stats.distributions.discrete_gaussian_integer.
DiscreteGaussianDistributionIntegerSampler object which is then repeatedly queried.

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

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

EXAMPLES:
sage: ZZ.random_element().parent() is ZZ
True

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

sage: from collections import defaultdict
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")

Here we use the distribution '1/n':

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

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

sage: -10 <= ZZ.random_element(-10, 10) < 10
True
sage: 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)
while ZZ.random_element(10^50) < 10^49:
    pass

Notice that the right endpoint is not included:

all(ZZ.random_element(-2, 2) < 2 for _ in range(100))

We return a sample from a discrete Gaussian distribution:

residue_field(prime, check=True, names=None)

Return the residue field of the integers modulo the given prime, i.e. $\mathbb{Z}/p\mathbb{Z}$.
• 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
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
sage: pi(123/234)
6
sage: pi(1/61)
Traceback (most recent call last):
  ...ZeroDivisionError: Cannot reduce rational 1/61 modulo 61:
  it has negative valuation
sage: lift = F.lift_map(); lift
Lifting map:
  From: Residue field of Integers modulo 61
  To:   Integer Ring
sage: lift(F(12345/67890))
33
sage: (12345/67890) % 61
33
```

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

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

**valuation(p)**

Return the discrete valuation with uniformizer $p$.

EXAMPLES:

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

See also:

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

**zeta**(n=2)

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

INPUT:
• n – (default 2) a positive integer

OUTPUT:
an n-th root of unity in \( \mathbb{Z} \)

EXAMPLES:

<table>
<thead>
<tr>
<th>sage: ZZ.zeta()</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sage: ZZ.zeta(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sage: ZZ.zeta(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traceback (most recent call last):</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>ValueError: no nth root of unity in integer ring</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sage: ZZ.zeta(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traceback (most recent call last):</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>ValueError: n must be positive in zeta()</td>
</tr>
</tbody>
</table>

\[
\text{sage.rings.integer_ring.crt_basis}(X, xgcd=None)
\]

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 \( \text{len}(X) \) extended GCD’s.

EXAMPLES:

<table>
<thead>
<tr>
<th>sage: X = [11,20,31,51]</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage: E = crt_basis([11,20,31,51])</td>
</tr>
<tr>
<td>sage: E[0]%X[0], E[1]%X[0], E[2]%X[0], E[3]%X[0]</td>
</tr>
</tbody>
</table>

(continues on next page)
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:

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

Add an integer and a real number:

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

Add an integer and a rational number:

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

Add an integer and a complex number:

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

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

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

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

### Multiplication:

```
(1, 0, 0, 0)
sage: E[0]%X[1], E[1]%X[1], E[2]%X[1], E[3]%X[1]
(0, 1, 0, 0)
sage: E[0]%X[2], E[1]%X[2], E[2]%X[2], E[3]%X[2]
(0, 0, 1, 0)
sage: E[0]%X[3], E[1]%X[3], E[2]%X[3], E[3]%X[3]
(0, 0, 0, 1)
```
sage: a = Integer(3); b = Integer(4)
sage: a * b == 12
True
sage: loads((a * 4.0).dumps()) == a*b
True
sage: a * Rational(2)/5
6/5
sage: [2,3] * 4
[2, 3, 2, 3, 2, 3, 2, 3]
sage: 'sage' * Integer(3)
'sagesagesage'

COERCIONS:

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

sage: n = 9390823
sage: RR = RealField(200)
    # needs sage.rings.real_mpfr
sage: 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 github issue #12116)
sage.rings.integer.GCD_list(n)

Return the greatest common divisor of a list of integers.

INPUT:

- n – list or tuple

Elements of n 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'>
```

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

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

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

```python
sage: 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 Integer interprets strings that begin with 0o as octal numbers, strings that begin with 0x as hexadecimal numbers and strings that begin with 0b as binary numbers.

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

EXAMPLES:

```python
sage: Integer(123)
123
sage: Integer("123")
123
```
Sage Integers support PEP 3127 literals:

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

Conversion from PARI:

```python
sage: Integer(pari('-10380104371593008048799446356441519384'))  # needs sage.libs.pari
-10380104371593008048799446356441519384
sage: Integer(pari('Pol([-3]'))))  # needs sage.libs.pari
-3
```

Conversion from gmpy2:

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

```
__pow__(left, right, modulus)
```

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

**EXAMPLES:**

```python
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:

```python
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:

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

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

```python
sage: 2^x  # symbolic x
2^x

sage: 2^1.5  # real number
2.82842712474619

sage: 2^float(1.5)  # python float abs tol 3e-16
2.8284271247461903

sage: 2^I  # complex number
2^I

sage: r = 2 ^ int(-3); r; type(r)
1/8

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

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

A symbolic sum:

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

### additive_order()

Return the additive order of self.

**EXAMPLES:**

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

### as_integer_ratio()

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

**EXAMPLES:**

```python
sage: # needs sage.rings.rational.Rational
sage: 2^x  # symbolic x
2^x
```
### balanced_digits

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

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

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

**INPUT:**

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

**EXAMPLES:**

```
sage: 8.balanced_digits(3)
[-1, 0, 1]
sage: (-15).balanced_digits(5)
[0, 2, -1]
sage: 17.balanced_digits(6)
[-1, 3]
sage: 17.balanced_digits(6, positive_shift=False)
[-1, -3, 1]
sage: (-46).balanced_digits()
[4, 5, -1]
sage: (-46).balanced_digits(positive_shift=False)
[4, -5]
sage: (-23).balanced_digits(12)
[1, -2]
sage: (-23).balanced_digits(12, positive_shift=False)
[1, -2]
sage: 0.balanced_digits(7)
[]
sage: 14.balanced_digits(5.8)
Traceback (most recent call last):
  ...
ValueError: base must be an integer
sage: 14.balanced_digits(2)
Traceback (most recent call last):
  ...
ValueError: base must be > 2
```

**See also:**

- `digits`
- `binary()`

Return the binary digits of `self` as a string.
EXAMPLES:

```
sage: print(Integer(15).binary())
1111
sage: print(Integer(16).binary())
10000
sage: print(Integer(16938402384092843092843098243).binary())
1101101011101100011110001110010100111010001101010001111101010000000101111000010000011
```

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

Return the binomial coefficient “self choose m”.

INPUT:

• m – an integer

• algorithm – 'gmp' (default), 'mpir' (an alias for gmp), or 'pari': 'gmp' is faster for small m, and 'pari' tends to be faster for large m

OUTPUT: integer

EXAMPLES:
```
sage: 10.binomial(2)
45
sage: 10.binomial(2, algorithm='pari') #...
˓→needs sage.libs.pari
45
sage: 10.binomial(-2)
0
sage: (-2).binomial(3)
-4
sage: (-3).binomial(0)
1
```

The argument m or (self - m) must fit into 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
```

```
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
```

(continues on next page)
bits()

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

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

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
```

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:

```python
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()  # needs sage.libs.pari
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)  # needs sage.libs.pari
[1, 3, 5, 7]
sage: n.coprime_integers(11)  # needs sage.libs.pari
[1, 3, 5, 7, 9]
sage: n = 5; n.coprime_integers(10)  # needs sage.libs.pari
[1, 2, 3, 4, 6, 7, 8, 9]
sage: n.coprime_integers(5)  # needs sage.libs.pari
[1, 2, 3, 4]
sage: n = 99; n.coprime_integers(99)  # needs sage.libs.pari
[1, 2, 4, 5, 7, 8, 10, 13, 14, 16, 17, 19, 20, 23, 25, 26, 28, 29, 31, 32, 34, 35, 37, 38, 40, 41, 43, 46, 47, 49, 50, 52, 53, 56, 58, 59, 61, 62, 64, 65, 67, 68, 70, 71, 73, 74, 76, 79, 80, 82, 83, 85, 86, 89, 91, 92, 94, 95, 97, 98]
```
AUTHORS:

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

ALGORITHM:

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

crt\((y, m, n)\)

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

EXAMPLES:

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

denominator()

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

EXAMPLES:

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

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

Return a list of digits for \( self \) in the given base in little endian order.

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

INPUT:

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

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

Also see \( ndigits() \).

EXAMPLES:
We support large bases.

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

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 \( \text{ZZ()} \) with a base:

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

1.2. Elements of the ring \( \mathbb{Z} \) of integers
Using `sum()` and `enumerate()` to do the same thing is slightly faster in many cases (and `balanced_sum()` may be faster yet). Of course it gives the same result:

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

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

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

This function is comparable to `str()` for speed.

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

```python
divide_knowing_divisible_by(right)
```

Return the integer `self/right` when `self` is divisible by `right`.

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

EXAMPLES:

```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
```
divides\( (n) \)

Return True if self divides n.

EXAMPLES:

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

divisors\( (\text{method=None}) \)

Return the list of all positive integer divisors of this integer, sorted in increasing order.

EXAMPLES:

```python
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 * 5^2 * 17).divisors()
[1, 2, 3, 4, 5, 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)
172800
sage: prod(e + 1 for p, e in factor(a))
172800
# needs sage.libs.pari
```

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

```python
sage: n = 2^551 - 1
sage: L = n.divisors()
# needs sage.libs.pari
sage: len(L)
# needs sage.libs.pari
256
sage: L[-1] == n
# 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.

---

### euclidean_degree()

Return the degree of this element as an element of an Euclidean domain.

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

**EXAMPLES:**

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

### 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.

**INPUT:**

- \( m \) - integer \( \geq 2 \)

**AUTHORS:**

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

**EXAMPLES:**

```python
sage: Integer(125).exact_log(5)
3
sage: Integer(124).exact_log(5)
2
sage: Integer(126).exact_log(5)
3
sage: Integer(3).exact_log(5)
0
sage: Integer(1).exact_log(5)
```

(continues on next page)
sage: Integer(178^1700).exact_log(178)
1700
sage: Integer(178^1700-1).exact_log(178)
1699
sage: Integer(178^1700+1).exact_log(178)
1700
sage: # we need to exercise the large base code path too
sage: Integer(1780^1700-1).exact_log(1780)  # needs sage.rings.real_interval_field
1699

sage: # The following are very very fast.
sage: # Note that for base m a perfect power of 2, we get the exact log by counting bits.

sage: n = 2983579823750185701375109835; m = 32
sage: n.exact_log(m)
18

sage: # The next is a favorite of mine. The log2 approximate is exact and immediately provable.

sage: n = 90153710570912709517902579010793251709257901270941709247901209742124
sage: m = 213509721309572
sage: 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

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

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

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:

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

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

```python
sage: y = Integer(145^145)
sage: y.exp()
# needs sage.symbolic
e^25024207011349679210459585279553675697932183658421565626032359240943270730655416822487611009401445089575929624277525047615...
```

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

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

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

INPUT:

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

• proof - bool (default: True) whether or not to prove primality of each factor (only applicable for ’pari’ and ’ecm’).

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

OUTPUT:

• a Factorization object containing the prime factors and their multiplicities

EXAMPLES:

```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):

\[
\begin{align*}
\text{sage: } & f = 60\text{.factor()} \\
\text{sage: } & \text{list}(f) \\
& [(2, 2), (3, 1), (5, 1)] \\
\text{sage: } & f[2] \\
& (5, 1)
\end{align*}
\]

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

\[
\begin{align*}
\text{sage: } & f = (3^6)\text{.factor()} \\
\text{sage: } & \text{dict}(f) \\
& {3: 6} \\
\text{sage: } & \text{dict}(f)[3] \\
& 6
\end{align*}
\]

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

\[
\begin{align*}
\text{sage: } & n = 92038409282439042384290348203948092384082349082 \\
\text{sage: } & n\text{.factor(proof=}False\text{)} \\
& 2 \times 11 \times 1531 \times 4402903 \times 10023679 \times 61916295472170540533894518173 \\
\text{sage: } & n\text{.factor(proof=}True\text{)} \\
& 2 \times 11 \times 1531 \times 4402903 \times 10023679 \times 61916295472170540533894518173
\end{align*}
\]

We factor using trial division only:

\[
\begin{align*}
\text{sage: } & n = 9285640583745019265831379463815822381094652231 \\
\text{sage: } & n\text{.factor(limit=}1000\text{)} \\
& 2 \times 11 \times 41835640583745019265831379463815822381094652231
\end{align*}
\]

An example where FLINT is used:

\[
\begin{align*}
\text{sage: } & n = 8286238573237628428164127822 \\
\text{sage: } & n\text{.factor(algorithm=}\text{flint}\text{'}) \\
& 2 \times 3 \times 11 \times 13 \times 41 \times 73 \times 22650083 \times 1424602265462161
\end{align*}
\]

We factor using a quadratic sieve algorithm:

\[
\begin{align*}
\text{sage: } & n = 92038409282439042384290348203948092384082349082 \\
\text{sage: } & n\text{.factor(algorithm=}\text{qsieve}\text{'}) \\
& \text{doctest:... RuntimeWarning: the factorization returned by qsieve may be incomplete (the factors may not be prime)}
\end{align*}
\]

We factor using the elliptic curve method:
# 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
```

**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
```

Large integers raise an **OverflowError**:

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

And negative ones a **ValueError**:

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

**floor()**

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

**EXAMPLES:**

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

**gamma()**

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

**EXAMPLES:**

```
sage: # needs sage.symbolic
sage: gamma(5)
```

(continues on next page)
24
\[
sage: \gamma(0)
\]
Infinity
\[
sage: \gamma(-1)
\]
Infinity
\[
sage: \gamma(-2^{150})
\]
Infinity

**global\_height** *(prec=None)*

Return the absolute logarithmic height of this rational integer.

**INPUT:**

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

**OUTPUT:**

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

**ALGORITHM:**

The height of the integer \( n \) is \( \log |n| \).

**EXAMPLES:**

\[
\begin{align*}
\text{sage: } & \text{# needs sage.rings.real_mpfr} \\
\text{sage: } & \text{ZZ(5).global\_height()} \\
& 1.60943791243410 \\
\text{sage: } & \text{ZZ(-2).global\_height(prec=100)} \\
& 0.69314718055994530941723212146 \\
\text{sage: } & \text{exp(_)} \\
& 2.0000000000000000000000000000
\end{align*}
\]

**hex()**

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

**Note:** ‘0x’ is not prepended to the result like is done by the corresponding Python function on `int`. 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:**

\[
\begin{align*}
\text{sage: } & \text{print(Integer(15).hex())} \\
& f \\
\text{sage: } & \text{print(Integer(16).hex())} \\
& 10 \\
\text{sage: } & \text{print(Integer(16938402384092843092843098243).hex())} \\
& 36bble3929d1a8fe2802f083
\end{align*}
\]

**imag()**

Return the imaginary part of `self`, which is zero.

**EXAMPLES:**
inverse_mod(n)
Return the inverse of self modulo n, if this inverse exists.
Otherwise, raise a ZeroDivisionError exception.

INPUT:
• self - Integer
• n - Integer, or ideal of integer ring

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

IMPLEMENTATION:
Call the mpz_invert GMP library function.

EXAMPLES:

sage: a = Integer(189)
sage: a.inverse_mod(10000)
4709
sage: a.inverse_mod(-10000)
4709
sage: a.inverse_mod(1890)
Traceback (most recent call last):
  ... ZeroDivisionError: inverse of Mod(189, 1890) does not exist
sage: a = Integer(19)**100000  # long time
sage: c = a.inverse_mod(a*a)  # long time
Traceback (most recent call last):
  ... ZeroDivisionError: inverse of Mod(..., ...) does not exist

We check that github issue #10625 is fixed:

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

We check that github issue #9955 is fixed:

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

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

EXAMPLES:

sage: (1).inverse_of_unit()
1
sage: (-1).inverse_of_unit()
-1
sage: 5.inverse_of_unit()
Traceback (most recent call last):
  ... 
ArithmeticError: inverse does not exist
sage: 0.inverse_of_unit()
Traceback (most recent call last):
  ... 
ArithmeticError: inverse does not exist

is_discriminant()
Return True if this integer is a discriminant.

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

EXAMPLES:

sage: (-1).is_discriminant()
False
sage: (-4).is_discriminant()
True
sage: 100.is_discriminant()
True
sage: 101.is_discriminant()
True

is_fundamental_discriminant()
Return True 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.

EXAMPLES:

sage: (-4).is_fundamental_discriminant() #← needs sage.libs.pari
True
sage: (-12).is_fundamental_discriminant()
False
sage: 101.is_fundamental_discriminant() #← needs sage.libs.pari
True

is_integer()
Return True as they are integers

EXAMPLES:

sage: sqrt(4).is_integer()
True

1.2. Elements 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()  # needs sage.libs.pari
False
sage: z = 7
sage: z.is_irreducible()  # needs sage.libs.pari
True
sage: z = -7
sage: z.is_irreducible()  # needs sage.libs.pari
True
```

is_norm(K, element=False, proof=True)
See QQ(self).is_norm().

EXAMPLES:

```
sage: n = 7
sage: n.is_norm(QQ)
True
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)
```

is_one()
Return True if the integer is 1, otherwise False.

EXAMPLES:

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

**is_perfect_power()**

Return True if self is a perfect power, i.e., if there exist integers $a$ and $b > 1$ 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:

```python
sage: Integer(-27).is_perfect_power()
True
sage: Integer(12).is_perfect_power()
False
sage: z = 8
sage: z.is_perfect_power()
True
sage: 144.is_perfect_power()
True
sage: 10.is_perfect_power()
False
sage: (-8).is_perfect_power()
True
sage: (-4).is_perfect_power()
False
```

**is_power_of(n)**

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.
- **is_perfect_power()**: If you don’t know the base but just want to know if this integer is a perfect power, use this function.
- **is_prime_power()**: Checks whether the base is prime.

EXAMPLES:

```python
sage: Integer(64).is_power_of(4)
True
```

(continues on next page)
sage: Integer(64).is_power_of(16)
False

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

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

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

**is_prime**(proof=None)

Test whether self is prime.

**INPUT:**

- proof – Boolean or None (default). If False, use a strong pseudo-primality test (see `is_pseudoprime()`). If True, use a provable primality test. If unset, use the default arithmetic proof flag.
Note: Integer primes are by definition positive! This is different than Magma, but the same as in PARI. See also the `is_irreducible()` method.

EXAMPLES:

```
sage: z = 2^31 - 1
sage: z.is_prime() # 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
```

```
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
```

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

```
sage: proof.arithmetic()
True
sage: n = 10^100 + 267
sage: timeit("n.is_prime()") # not tested # 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
```

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 pseudoprimitality test instead see `is_pseudoprime_power()`.
INPUT:

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

See also:

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

EXAMPLES:

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

With the get_data keyword set to True:

```sage
# 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)
sage: (p^97).is_prime_power(get_data=True)
(100000000000000000039, 97)
sage: q = p.next_prime(); q
```

(continues on next page)
The method works for large entries when \texttt{proof=False}:

\begin{verbatim}
sage: proof.arithmetic(False)
sage: ((10^500 + 961)^4).is_prime_power()  # needs sage.libs.pari
True
\end{verbatim}

We check that \texttt{github issue #4777} is fixed:

\begin{verbatim}
sage: n = 150607571^14
sage: n.is_prime_power()  # needs sage.libs.pari
True
\end{verbatim}

\begin{function}{is_pseudoprime()}
Test whether \texttt{self} is a pseudoprime.
This uses \texttt{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 \texttt{Wikipedia article Baillie-PSW_primality_test}

\begin{verbatim}
EXAMPLES:
sage: z = 2^31 - 1
sage: z.is_pseudoprime()  # needs sage.libs.pari
True
sage: z = 2^31
sage: z.is_pseudoprime()  # needs sage.libs.pari
False
\end{verbatim}

\begin{function}{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}

\begin{verbatim}
EXAMPLES:
sage: # needs sage.libs.pari
sage: x = 10^200 + 357
sage: x.is_pseudoprime()
True
sage: (x^12).is_pseudoprime_power()
True
\end{verbatim}
is_pseudoprime_power() (continued from previous page)

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

is_rational()

Return True as an integer is a rational number.

EXAMPLES:

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

is_square()

Return True if self is a perfect square.

EXAMPLES:

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

is_squarefree()

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

EXAMPLES:

```python
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
```

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    -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.
```

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

EXAMPLES:

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

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

EXAMPLES:

```python
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
```

(continues on next page)
sage: a.kronecker(b) == b.kronecker(a)
True

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

EXAMPLES:

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

log(m=None, prec=None)
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:

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)  # needs sage.rings.real_mpfr
3
sage: Integer(125).log(5, prec=53)  # needs sage.rings.real_mpfr
3

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)
100000.000000000

sage: (x + 1).log(3, 1000)
100000.
```

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

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

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

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

The logarithm of zero is done likewise:

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

Some rational bases yield integer logarithms (github issue #21517):

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

Check that Python ints are accepted (github issue #21518):

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

### multifactorial($k$)

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

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

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

**EXAMPLES:**

```
sage: 5.multifactorial(1)
120
sage: 5.multifactorial(2)
```

(continues on next page)
15
 sage: 5.multifactorial(3)
 10

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

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

When entries are too large an OverflowError is raised:

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

 multiplicative_order()

 Return the multiplicative order of self.

 EXAMPLES:

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

 nbits()

 Alias for bit_length().

 ndigits(base=10)

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

 INPUT:

 • base - integer (default: 10)

 EXAMPLES:

 sage: n = 52
 sage: n.ndigits()
 2
\begin{verbatim}
sage: n = -10003
sage: n.ndigits()
5
sage: n = 15
sage: n.ndigits(2)
4
sage: n = 100**1000000+1
sage: n.ndigits()
# needs sage.rings.real_interval_field
3000001
sage: n = 100**1000000-1
sage: n.ndigits()
# needs sage.rings.real_interval_field
3000000
sage: n = 10**1000000-10**999999
sage: n.ndigits()
# needs sage.rings.real_interval_field
10000000
\end{verbatim}

**next_prime** (\texttt{proof=None})

Return the next prime after self.

This method calls the PARI function \texttt{pari:nextprime}.

**INPUT:**

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

**EXAMPLES:**

\begin{verbatim}
sage: 100.next_prime()
# needs sage.libs.pari
101
sage: (10^50).next_prime()
# needs sage.libs.pari
1000000000000000000000000000000000000000000000000151
\end{verbatim}

Use \texttt{proof=False}, which is way faster since it does not need a primality proof:

\begin{verbatim}
sage: b = (2^1024).next_prime(proof=False)
# needs sage.libs.pari
sage: b - 2^1024
# needs sage.libs.pari
643
sage: Integer(0).next_prime()
# needs sage.libs.pari
2
sage: Integer(1001).next_prime()
# needs sage.libs.pari
1009
\end{verbatim}

**next_prime_power** (\texttt{proof=None})

Return the next prime power after self.

1.2. Elements of the ring $\mathbb{Z}$ of integers
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()
2
sage: 2.next_prime_power()
3
sage: 103.next_prime_power()  # needs sage.libs.pari
107
sage: 107.next_prime_power()
109
sage: 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()
13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298166903427690031858186486050853753882811946569946433649006084171
sage: 0.next_probable_prime()
2
sage: 126.next_probable_prime()
127
sage: 144168.next_probable_prime()
144169
```

nth_root(n, truncate_mode=0)

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

INPUT:
• \(n\) - integer \(\geq 1\) (must fit in the `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:
```python
sage: Integer(125).nth_root(3)
5
sage: Integer(124).nth_root(3)
Traceback (most recent call last):
  ... ValueError: 124 is not a 3rd power
sage: Integer(124).nth_root(3, truncate_mode=1)
(4, False)
sage: Integer(125).nth_root(3, truncate_mode=1)
(5, True)
sage: Integer(126).nth_root(3, truncate_mode=1)
(5, False)
sage: Integer(-125).nth_root(3)
-5
sage: Integer(-125).nth_root(3, truncate_mode=1)
(-5, True)
sage: Integer(-124).nth_root(3, truncate_mode=1)
(-4, False)
sage: Integer(-126).nth_root(3, truncate_mode=1)
(-5, False)
sage: Integer(125).nth_root(2, True)
(11, False)
sage: Integer(125).nth_root(3, True)
(5, True)
sage: Integer(125).nth_root(-5)
Traceback (most recent call last):
  ... ValueError: n (=-5) must be positive
sage: Integer(-25).nth_root(2)
Traceback (most recent call last):
  ... ValueError: cannot take even root of negative number
```

1.2. Elements of the ring \(\mathbb{Z}\) of integers
\begin{verbatim}
sage: a=9
da = 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)
\end{verbatim}

\textbf{numerator()}

Return the numerator of this integer.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: x = 5
sage: x.numerator()
5

sage: x = 0
sage: x.numerator()
0
\end{verbatim}

\textbf{oct()}

Return the digits of \texttt{self} in base 8.

\textbf{Note:} ‘0’ (or ‘0o’) is \textit{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())
155354361622472152472800570203
\end{verbatim}

Behavior of Sage integers vs. Python integers:
odd_part()

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

IMPLEMENTATION:

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

EXAMPLES:

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

ord($p$)

Return the $p$-adic valuation of self.

INPUT:

- $p$ - an integer at least 2.

EXAMPLES:

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

We do not require that $p$ is a prime:

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

**p_primary_part(p)**

Return the p-primary part of self.

**INPUT:**

- p – a prime integer.

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

**EXAMPLES:**

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

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:**

```sage
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)
```

**popcount()**

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

**EXAMPLES:**

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

sage: n = -17
sage: n.popcount()       +Infinity
```
**powermod**(exp, mod)

Compute self**exp** modulo mod.

**EXAMPLES:**

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

As expected, the following is invalid:

```sage
sage: z.powermod(31,0)
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:**

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

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

```sage
sage: b = (2**1024).previous_prime(proof=False)  
```

(continues on next page)
```python
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:**

```python
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
```

### 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:**

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

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:

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:

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:

```
sage: 43434.prime_to_m_part(20)
21717
sage: 2048.prime_to_m_part(2)
1
sage: 2048.prime_to_m_part(3)
2048
sage: 0.prime_to_m_part(2)
Traceback (most recent call last):
... ArithmeticError: self must be nonzero
```

quo_rem(other)
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:

```
sage: z = Integer(231)
sage: z.quo_rem(2)
(115, 1)
sage: z.quo_rem(-2)
(-116, -1)
sage: z.quo_rem(0)
Traceback (most recent call last):
... ZeroDivisionError: Integer division by zero
sage: a = ZZ.random_element(10**50)
sage: b = ZZ.random_element(10**15)
sage: q, r = a.quo_rem(b)
sage: q*b + r == a
True
sage: 3.quo_rem(ZZ['x'].0)
(0, 3)
```
rational_reconstruction($m$)

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

INPUT:

• self – Integer
• m – Integer

OUTPUT:

• a Rational

EXAMPLES:

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

real()

Return the real part of self, which is self.

EXAMPLES:

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

round($mode=\text{'away'}$)

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

EXAMPLES:

This example addresses github issue #23502:

```
sage: n = 6
sage: 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
```

sqrt($\text{prec=\text{None}, extend=True, all=\text{False}}$)

The square root function.

INPUT:
• **prec** – integer (default: None): if None, return an exact square root; otherwise return a numerical square root, to the given bits of precision.

• **extend** – bool (default: True): if True, return a square root in an extension ring, if necessary. Otherwise, raise a **ValueError** if the square is not in the base ring. Ignored if **prec** is not None.

• **all** - bool (default: False); if True, return all square roots of **self** (a list of length 0, 1, or 2).

**EXAMPLES:**

```
sage: Integer(144).sqrt() 12
sage: sqrt(Integer(144))
12
sage: Integer(102).sqrt()  # needs sage.symbolic
sqrt(102)
sage: n = 2
sage: n.sqrt(all=True)  # needs sage.symbolic
[sqrt(2), -sqrt(2)]
sage: n.sqrt(prec=10)  # needs sage.rings.real_mpfr
1.4
sage: n.sqrt(prec=100)  # needs sage.rings.real_mpfr
1.4142135623730950488016887242
sage: n.sqrt(prec=100, all=True)  # needs sage.rings.real_mpfr
[1.4142135623730950488016887242, -1.4142135623730950488016887242]
sage: n.sqrt(extend=False)
Traceback (most recent call last):
... ArithmeticError: square root of 2 is not an integer
sage: (-1).sqrt(extend=False)
Traceback (most recent call last):
... ArithmeticError: square root of -1 is not an integer
sage: Integer(144).sqrt(all=True)
[12, -12]
sage: Integer(0).sqrt(all=True)
[0]
```

**sqrtrem()**

Return \((s, r)\) where \(s\) is the integer square root of \(\text{self}\) 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()
```

(continues on next page)
(0, 0)

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

`squarefree_part` *(bound=-1)*

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

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

If `self` is 0, just returns 0.

**EXAMPLES:**

```python
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
```

```python
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)  # needs sage.libs.pari
49
```

`str` *(base=10)*

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

**EXAMPLES:**
sage: Integer(2^10).str(2)
'10000000000'
sage: Integer(2^10).str(17)
'394'
sage: two = Integer(2)
sage: two.str(1)
Traceback (most recent call last):
  ...  
ValueError: base (=1) must be between 2 and 36
sage: two.str(37)
Traceback (most recent call last):
  ...  
ValueError: base (=37) must be between 2 and 36
sage: big = 10^5000000
sage: s = big.str()  # long time (2s on sage.math, 2014)
5000001
sage: s[:10]  # long time (depends on above defn of s)
'1000000000'
support()
Return a sorted list of the primes dividing this integer.
OUTPUT: The sorted list of primes appearing in the factorization of this rational with positive exponent.
EXAMPLES:

sage: factorial(10).support()
[2, 3, 5, 7]
sage: (-999).support()
[3, 37]
Trying to find the support of 0 raises an ArithmeticError:

sage: 0.support()
Traceback (most recent call last):
  ...  
ArithmeticError: Support of 0 not defined.
test_bit(index)
Return the bit at index.
If the index is negative, returns 0.
Although internally a sign-magnitude representation is used for integers, this method pretends to use a two's complement representation. This is illustrated with a negative integer below.
EXAMPLES:

sage: w = 6
sage: w.str(2)
(continues on next page)
trailing_zero_bits()

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

EXAMPLES:

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

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

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

INPUT:

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

OUTPUT: A positive integer

EXAMPLES:

```sage
trial_division(bound='LONG_MAX', start=2)
''''
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)
1000003
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

sage: n.trial_division(bound=10^4)  
1000030000000000000000000000000000000012100363

sage: (-n).trial_division(bound=10^4)  
1000030000000000000000000000000000000012100363

sage: (-n).trial_division()  
100003

sage: n = 2 * next_prime(10^40); n.trial_division()  
2

sage: n = 3 * next_prime(10^40); n.trial_division()  
3

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

sage: n = 2 * next_prime(10^4); n.trial_division()  
2

sage: n = 3 * next_prime(10^4); n.trial_division()  
3

sage: n = 5 * next_prime(10^4); n.trial_division()  
5

You can specify a starting point:

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

trunc()

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

EXAMPLES:

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

val_unit(p)


INPUT:

• p - an integer at least 2.

OUTPUT:

1.2. Elements of the ring \( \mathbb{Z} \) of integers
• $v_p(self)$ - the $p$-adic valuation of $self$
• $u_p(self)$ - $self / p^{v_p(self)}$

EXAMPLES:

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

valuation($p$)
Return the $p$-adic valuation of $self$.

INPUT:
• $p$ - an integer at least 2.

EXAMPLES:

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

We do not require that $p$ is a prime:

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

xgcd($n$)
Return the extended gcd of this element and $n$.

INPUT:
• $n$ – an integer

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

$$g = s \cdot self + t \cdot n.$$
Note: There is no guarantee that the cofactors will be minimal. If you need the cofactors to be minimal use \texttt{._xgcd()}. Also, using \texttt{._xgcd()} directly might be faster in some cases, see \texttt{github issue #13628}.

EXAMPLES:

\begin{verbatim}
sage: 6.xgcd(4)
(2, 1, -1)
\end{verbatim}

\begin{verbatim}
class sage.rings.integer.IntegerWrapper
    Bases: Integer

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

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

    The constructor of \texttt{IntegerWrapper} further allows for specifying an alternative parent to \texttt{IntegerRing}.
\end{verbatim}

\begin{verbatim}
class sage.rings.integer.free_integer_pool()

\end{verbatim}

\begin{verbatim}
class sage.rings.integer.int_to_Z
    Bases: Morphism

    Morphism from Python ints to Sage integers.

    EXAMPLES:
    \begin{verbatim}
    sage: f = ZZ.coerce_map_from(int)
sage: type(f)
<class 'sage.rings.integer.long_to_Z'>
sage: f(5r)
5
sage: type(f(5r))
<class 'sage.rings.integer.Integer'>
sage: 1 + 2r
3
sage: type(1 + 2r)
<class 'sage.rings.integer.Integer'>
    \end{verbatim}

    This is intended for internal use by the coercion system, to facilitate fast expressions mixing ints and more complex Python types. Note that (as with all morphisms) the input is forcably coerced to the domain \texttt{int} if it is not already of the correct type which may have undesirable results:
    \begin{verbatim}
    sage: f.domain()
Set of Python objects of class 'int'
sage: f(1/3)
0
sage: f(1.7)
1
    \end{verbatim}
\end{verbatim}

1.2. Elements of the ring \( \mathbb{Z} \) of integers
A pool is used for small integers:

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

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

Return True if x is of the Sage `Integer` type.

**EXAMPLES:**

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

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

Bases: `Morphism`

**EXAMPLES:**

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

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

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

**EXAMPLES:**

```python
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
```
1.3 Cython wrapper for bernmm library

AUTHOR:
• David Harvey (2008-06): initial version

sage.rings.bernmm.bernmm_bernpmod(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_bernpmod
sage: bernoulli(0) % 5, bernmm_bernpmod(5, 0)
(1, 1)
sage: bernoulli(1) % 5, bernmm_bernpmod(5, 1)
(2, 2)
sage: bernoulli(2) % 5, bernmm_bernpmod(5, 2)
(1, 1)
sage: bernoulli(3) % 5, bernmm_bernpmod(5, 3)
(0, 0)
sage: bernoulli(4), bernmm_bernpmod(5, 4)
(-1/30, -1)
sage: bernoulli(18) % 5, bernmm_bernpmod(5, 18)
(4, 4)
sage: bernoulli(19) % 5, bernmm_bernpmod(5, 19)
(0, 0)
sage: p = 10000019; k = 1000
sage: bernoulli(k) % p
1972762
sage: bernmm_bernpmod(p, k)
1972762
```

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

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

INPUT:
• \( k \) – non-negative integer
• \( \text{num\_threads} \) – integer \( \geq 1 \), 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)
\rightarrow 94598037819122125295227433069493721872702841533066936133385696204311395415197247711/33330
sage: bernmm_bern_rat(100, 3)
\rightarrow 94598037819122125295227433069493721872702841533066936133385696204311395415197247711/33330

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

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 $[B(0), B(2), \ldots B(p-3)]$.

ALGORITHM:

Described in accompanying latex file.

PERFORMANCE:

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

EXAMPLES:

Check the results against PARI’s C-library implementation (that computes exact rationals) for $p = 37$:
sage: [bernoulli(n) % 37 for n in range(0, 36, 2)]
[1, 31, 16, 15, 16, 4, 17, 32, 22, 31, 15, 15, 17, 12, 29, 2, 0, 2]

Boundary case:
sage: bernoulli_mod_p(3)
[1]

AUTHOR:
• David Harvey (2006-08-06)
sage.rings.bernoulli_mod_p.bernoulli_mod_p_single(p, k)
Return the Bernoulli number $B_k$ mod $p$.
If $B_k$ is not $p$-integral, an ArithmeticError is raised.

INPUT:
• p – integer, a prime
• k – non-negative integer

OUTPUT:
The $k$-th Bernoulli number mod $p$.

EXAMPLES:
sage: bernoulli_mod_p_single(1009, 48)
628
sage: bernoulli(48) % 1009
628
sage: bernoulli_mod_p_single(1, 5)
Traceback (most recent call last):
  ... ValueError: p (=1) must be a prime >= 3
sage: bernoulli_mod_p_single(100, 4)
Traceback (most recent call last):
  ... ValueError: p (=100) must be a prime
sage: bernoulli_mod_p_single(19, 5)
0
sage: bernoulli_mod_p_single(19, 18)
Traceback (most recent call last):
  ... ArithmeticError: B_k is not integral at p
sage: bernoulli_mod_p_single(19, -4)
Traceback (most recent call last):
  ... ValueError: k must be non-negative
Check results against \textit{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)
[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]
\end{verbatim}

\textbf{AUTHOR:}

\begin{itemize}
  \item David Harvey (2007-08-31)
  \item David Harvey (2008-06): rewrote to use bernmm library
\end{itemize}

\texttt{sage.rings.bernoulli\_mod\_p.verify\_bernoulli\_mod\_p(data)}

\begin{flushleft}
Compute checksum for Bernoulli numbers.
It checks the identity
\[
\frac{(p-3)}{2} \sum_{n=0}^{\langle p \rangle/2} 2^{2n}(2n+1)B_{2n} \equiv -2 \pmod{p}
\]
(see “Irregular Primes to One Million”, Buhler et al)
\end{flushleft}

\textbf{INPUT:}

\begin{itemize}
  \item data – list, same format as output of \texttt{bernoulli\_mod\_p()} function
\end{itemize}

\textbf{OUTPUT: bool – True if checksum passed}

\textbf{EXAMPLES:}

\begin{verbatim}
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])
\end{verbatim}
True
\begin{verbatim}
sage: verify_bernoulli_mod_p([1, 2, 3, 4, 5])
False
\end{verbatim}

This one should test that long longs are working:
\begin{verbatim}
sage: verify_bernoulli_mod_p(bernoulli_mod_p(next_prime(20000)))
True
\end{verbatim}

AUTHOR: David Harvey

1.5 Integer factorization functions

AUTHORS:

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

sage.rings.factorint.\texttt{aurifeuillian}(n, m, F=None, check=True)

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

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:

\begin{verbatim}
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)
\end{verbatim}
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

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

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

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

**INPUT:**
- $n$ – integer

**OUTPUT:**
List of factors of $n$ found by Aurifeuillian factorization.

**EXAMPLES:**

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

**REFERENCES:**
- [http://mathworld.wolfram.com/AurifeuillianFactorization.html](http://mathworld.wolfram.com/AurifeuillianFactorization.html)
- [Bre1993] Theorem 3

```python
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 *
  37455059850181093658177663096313181393
sage: factor_cunningham((3^101+1)*(2^60).next_prime(), proof=False)  # optional - cunningham_tables
2^2 * 379963 * 1152921504606847009 * 1017291527198723292208309354658785077827527
```

```python
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
sage: from sage.rings.factorint import factor_trial_division
sage: n = 920384092842390423848290348203948092384082349082
sage: factor_trial_division(n, 1000)
2 * 11 * 41835640583745019265831379463815822381094652231
sage: factor_trial_division(n, 2000)
2 * 11 * 1531 * 27325696005058797691594630609938486205809701
```

## 1.6 Integer factorization using FLINT

AUTHORS:

• Michael Orlitzky (2023)

```python
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)
```

(continues on next page)
sage: prod( f^e for (f,e) in factors ) == n
True

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

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

The factorization of unity is empty:

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

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

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

Check that PARI’s debug level is properly reset (github issue #18792):

sage: alarm(0.5); factor(2^1000 - 1, verbose=5)
Traceback (most recent call last):
  ...
AlarmInterrupt
1.8 Basic arithmetic with C integers

```python
sage: pari.get_debug_level()
0
```

```python
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.
```

```python
sage.rings.fast_arith.prime_range(start, stop=None, algorithm=None, py_ints=False)
    Return a list of all primes between start and stop - 1, inclusive.
    If the second argument is omitted, this returns the primes up to the first argument.
    The sage command primes() is an alternative that uses less memory (but may be slower), because it returns an iterator, rather than building a list of the primes.
    INPUT:
    • start – integer, lower bound (default: 1)
    • stop – integer, upper bound
    • algorithm – optional string (default: None), one of:
        - None: Use algorithm "pari_primes" if stop <= 436273009 (approximately 4.36E8). Otherwise use algorithm "pari_isprime".
        - "pari_primes": Use PARI's pari:primes function to generate all primes from 2 to stop. This is fast but may crash if there is insufficient memory. Raises an error if stop > 436273009.
        - "pari_isprime": Wrapper for list(primes(start, stop)). Each (odd) integer in the specified range is tested for primality by applying PARI's pari:isprime function. This is slower but will work for much larger input.
    • py_ints – optional boolean (default False), return Python ints rather than Sage Integers (faster). Ignored unless algorithm "pari_primes" is being used.
    EXAMPLES:
```
sage: prime_range(10)
[2, 3, 5, 7]
sage: prime_range(7)
[2, 3, 5]
sage: prime_range(2000, 2020)
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 \(\text{start}^*\) and \(\text{stop}^*\), so the output will be the primes between \(\text{start}^*\) and \(\text{stop}^* - 1\), which may not be exactly the same as the primes between \(\text{start}\) and \(\text{stop} - 1\).

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

### 1.9 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 (github issue #16374)

```python
def four_squares_pyx(n):
    return four_squares(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:

```python
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
sage: from sage.rings.sum_of_squares import is_sum_of_two_squares_pyx
sage: [x for x in range(30) if is_sum_of_two_squares_pyx(x)]
[0, 1, 2, 4, 5, 8, 9, 10, 13, 16, 17, 18, 20, 25, 26, 29]
sage: is_sum_of_two_squares_pyx(2**32)
Traceback (most recent call last):
... OverflowError: ...
```

```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)
```

(continues on next page)
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
sage: three_squares_pyx(107)
(1, 5, 9)
sage: three_squares_pyx(2**32)
Traceback (most recent call last):
... OverflowError: ...

\begin{verbatim}
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 \texttt{ValueError} is raised. The input must be lesser than \(2^{32} = 4294967296\), otherwise an \texttt{OverflowError} is raised.

\textbf{See also:}
\texttt{two_squares()} is much more suited for large inputs

\textbf{EXAMPLES:}
\end{verbatim}

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

sage.arith.functions.LCM_list(v)

Return the LCM of an iterable v.
Elements of v are converted to Sage objects if they aren’t already.
This function is used, e.g., by \( \text{lcm}() \).

INPUT:
• v – an iterable

OUTPUT: integer

EXAMPLES:

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

The inputs are converted to Sage integers:

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

sage.arith.functions.lcm(a, b=None)
The least common multiple of a and b, or if a is a list and b is omitted the least common multiple of all elements of a.

Note that LCM is an alias for lcm.

INPUT:
• a, b – two elements of a ring with lcm or
• a – a list or tuple of elements of a ring with lcm

OUTPUT:

First, the given elements are coerced into a common parent. Then, their least common multiple \textit{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
```
1.11 Generic implementation of powering

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

\texttt{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:

\begin{verbatim}
sage: from sage.arith.power import generic_power
sage: generic_power(int(12), int(0))
1
sage: generic_power(int(0), int(100))
0
sage: generic_power(Integer(10), Integer(0))
1
sage: generic_power(Integer(0), Integer(23))
0
sage: sum([generic_power(2,i) for i in range(17)]) #test all 4-bit combinations
131071
sage: F = Zmod(5)
sage: a = generic_power(F(2), 5); a
2
sage: a.parent() is F
True
sage: a = generic_power(F(1), 2)
sage: a.parent() is F
True
sage: generic_power(int(5), 0)
1
sage: generic_power(2, 5/4)
Traceback (most recent call last):
... NotImplentedError: non-integral exponents not supported
\end{verbatim}
```python
sage: x = SymbolicMul("x")
sage: print(generic_power(x, 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 \( \prod_{j \neq i} m_j \) and \( \prod_{j} = 1 \mod m_i \).

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

**EXAMPLES:**

```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 [...]

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

**crt**

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

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

**INPUT:**

- \( b \) - a list of length at most self.n

**OUTPUT:**

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

**EXAMPLES:**

```python
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
```

(continues on next page)
sage: res % 10009
5
sage: res % 10037
7
sage: res % 10039
9

extend_with_primes(plist, partial_products=None, check=True)

Extend the stored list of moduli with the given primes in plist.

EXAMPLES:

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

list()

Return a list with the prime moduli.

EXAMPLES:

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

partial_product(n)

Return a list containing precomputed partial products.

EXAMPLES:

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

precomputation_list()

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

EXAMPLES:

sage: from sage.arith.multi_modular import MultiModularBasis_base
sage: mm = MultiModularBasis_base([46307, 10007]); mm
MultiModularBasis with moduli [46307, 10007]
sage: mm.precomputation_list()
[1, 4013]
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 precomputed data, and return the new prime modulus.

EXAMPLES:

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

replace_prime(ix)

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

INPUT:

- ix – index into list of moduli

OUTPUT: the new prime modulus

EXAMPLES:

```
sage: from sage.arith.multi_modular import MutableMultiModularBasis
sage: mm = MutableMultiModularBasis([10007, 10009, 10037, 10039])
sage: mm
MultiModularBasis with moduli [10007, 10009, 10037, 10039]
sage: prev_prod = mm.prod(); prev_prod
10092272478850909
sage: mm.precomputation_list()
[1, 5004, 6536, 6060]
sage: mm.partial_product(2)
```

(continues on next page)
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 ≡ a mod m and x ≡ b mod n, if one exists. By the Chinese Remainder Theorem, a solution to the simultaneous congruences exists if and only if a ≡ 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 ≡ a_i (mod b_i), if one exists.

See also:

• CRT_list()

EXAMPLES:

Using crt by giving it pairs of residues and moduli:

<table>
<thead>
<tr>
<th>sage:</th>
<th>crt(2, 1, 3, 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sage:</th>
<th>crt(13, 20, 100, 301)</th>
</tr>
</thead>
<tbody>
<tr>
<td>28013</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sage:</th>
<th>crt([2, 1], [3, 5])</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sage:</th>
<th>crt([13, 20], [100, 301])</th>
</tr>
</thead>
<tbody>
<tr>
<td>28013</td>
<td></td>
</tr>
</tbody>
</table>
You can also use upper case:

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

Note that this also works for polynomial rings:

```
sage: # 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
```

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
```

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):
...
```

(continues on next page)
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--

sage: crt(int(2), int(3), int(7), int(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
```
sage: from gmpy2 import mpz
```
```
sage: crt(mpz(2), mpz(3), mpz(7), numpy.int8(11))
```
# needs numpy
```
sage: from gmpy2 import mpz
```
```
sage: CRT_basis
```
```
sage: CRT_basis(moduli)
```
Return a CRT basis for the given moduli.

**INPUT:**

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

**OUTPUT:**

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

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

**EXAMPLES:**

```
sage: a1 = ZZ(mod(42,5))
sage: a2 = ZZ(mod(42,13))
sage: c1,c2 = CRT_basis([5,13])
sage: mod(a1*c1+a2*c2,5*13)
42
```
A polynomial example:
sage: x=polygen(QQ)
sage: mods = [x,x^2+1,2*x-3]
sage: b = CRT_basis(mods)
sage: b
[-2/3*x^3 + x^2 - 2/3*x + 1, 6/13*x^3 - x^2 + 6/13*x, 8/39*x^3 + 8/39*x]
sage: [[bi % mj for mj in mods] for bi in b]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]

sage.arith.misc.CRT_list(values, moduli)

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:

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

It also works if the moduli are not coprime:

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

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

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

The arguments must be lists:

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

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

sage: CRT_list([1,2,3],[2,3,5])
23
sage: CRT_list([1,2,3],[2,3])
(continues on next page)
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 $\text{CRT}()$ to each entry.

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

**OUTPUT:**
- list - application of CRT componentwise.

**EXAMPLES:**

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

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

class sage.arith.misc.Euler_Phi

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 $n$\leq0 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
```
Notice that euler_phi is defined to be 0 on negative numbers and 0.

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

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

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

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

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

The phi function also has a special plotting method.

```
sage: P = plot(euler_phi, -3, 71)  # needs sage.libs.pari sage.plot
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8  # needs numpy
sage: euler_phi(int8(37))  # needs numpy sage.libs.pari
36
sage: from gmpy2 import mpz
sage: euler_phi(mpz(37))  # needs sage.libs.pari
36
```

AUTHORS:

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

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

Plot the Euler phi function.

INPUT:

- xmin - default: 1
- xmax - default: 50
- pointsize - default: 30
- rgbcolor - default: (0,0,1)
\begin{itemize}
  \item join - default: True; whether to join the points.
  \item **kwds - passed on
\end{itemize}

EXAMPLES:

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
sage.arith.misc.GCD(a, b=None, **kwargs)
\end{verbatim}

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:

\begin{verbatim}
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
\end{verbatim}

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

\begin{verbatim}
sage: gcd(3, 6, 2)
Traceback (most recent call last):
  ...
TypeError: ...gcd() takes ...
sage: gcd([3, 6, 2])
1
\end{verbatim}

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

\begin{verbatim}
sage: gcd(3)
Traceback (most recent call last):
\end{verbatim}
... 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
class sage.arith.misc.Moebius
Bases: object

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

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

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

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

**INPUT:**
- \(n\) - anything that can be factored.

**OUTPUT:** 0, 1, or -1

**EXAMPLES:**

```python
sage: # needs sage.libs.pari
sage: moebius(-5)
-1
sage: moebius(9)  # potentially nonstandard!
0
sage: moebius(12)  # potentially nonstandard!
0
sage: moebius(-35)
1
sage: moebius(-1)
1
sage: moebius(7)
-1
sage: moebius(0)  # potentially nonstandard!
0
```

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

```python
sage: x = GF(7)['x'].0
sage: moebius(x + 2)  # needs sage.libs.pari
-1
```

Tests with numpy and gmpy2 numbers:
sage: from numpy import int8
˓→# needs numpy
sage: moebius(int8(-5))
˓→# needs numpy sage.libs.pari -1
sage: from gmpy2 import mpz
sage: moebius(mpz(-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:

sage: from sage.arith.misc import Moebius
sage: p = Moebius().plot()  # needs sage.libs.pari sage.plot
sage: p.ymax()  # needs sage.libs.pari sage.plot 1.0

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

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

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

EXAMPLES:

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

class sage.arith.misc.Sigma

Bases: object

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

INPUT:
• \( n \) - integer
• \( k \) - integer (default: 1)

OUTPUT: integer

EXAMPLES:

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

The sigma function also has a special plotting method.

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

This method also works with \( k \)-th powers.

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

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:

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

EXAMPLES:

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

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

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

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

**INPUT:**

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

**OUTPUT:**

- \( g, s, t \) - such that \( g = s \cdot a + t \cdot b \)

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

**EXAMPLES:**

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

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

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

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

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

```
\[
\text{sage.arith.misc.algdep}(z, \text{degree}, \text{known_bits}=\text{None}, \text{use_bits}=\text{None}, \text{known_digits}=\text{None}, \text{use_digits}=\text{None}, \\
\text{height_bound}=\text{None}, \text{proof}=\text{False})
\]

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

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

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

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

Note that algebraic_dependency is a synonym for algdep.

INPUT:

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

EXAMPLES:

```python
sage: algdep(1.888888888888888, 1)
# needs sage.libs.pari
9*x - 17
sage: algdep(0.1212121212121212, 1)
# needs sage.libs.pari
33*x - 4
sage: algdep(sqrt(2), 2)
```

(continues on next page)
This example involves a complex number:

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

This example involves a $p$-adic number:

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

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

```
sage: # 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:

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

For stronger results, we need more precision:

```
sage: # needs sage.libs.pari sage.symbolic
sage: algdep(pi.n(), 5, height_bound=100, proof=True) is None
```
Traceback (most recent call last):
...
ValueError: insufficient precision for non-existence proof
sage: algdep(pi.n(200), 5, height_bound=100, proof=True) is None
True
sage: algdep(pi.n(), 10, height_bound=10, proof=True) is None
Traceback (most recent call last):
...
ValueError: insufficient precision for non-existence proof
sage: algdep(pi.n(200), 10, height_bound=10, proof=True) is None
True

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

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

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

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

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

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

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

Note that algebraic_dependency is a synonym for algdep.

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

EXAMPLES:
This example involves a complex number:

\[
\begin{align*}
sage: \ z &= \left(\frac{1}{2}\right) \ast (1 + \mathrm{RDF}(\sqrt{3})) \ast \mathrm{CC}.0); \ z \\
\noalign{\hline}
&\text{\textasciitilde needs sage.symbolic} \\
&0.500000000000000 + 0.866025403784439\mathrm{i} \\
sage: \ \text{algdep(}z, \ 6) \\
&\text{\textasciitilde needs sage.symbolic} \\
&x^2 - x + 1
\end{align*}
\]

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

\[
\begin{align*}
sage: \ K &= \mathrm{Qp}(3, \ \text{print_mode}=\text{\textquoteleft series\textquoteright}) \\
\noalign{\hline}
&\text{\textasciitilde needs sage.rings.padics} \\
sage: \ a &= K(7/19); \ a \\
&\text{\textasciitilde needs sage.rings.padics} \\
&1 + 2^3 + 3^2 + 3^3 + 2^3 \ast 4 + 2^3 \ast 5 + 3^8 + 2^3 \ast 9 + 3^11 + 3^12 + 2^3 \ast 15 + 2^3 \ast 16 + . \ast \text{\textasciitilde needs sage.rings.padics} \\
sage: \ \text{algdep(}a, \ 1) \\
&\text{\textasciitilde needs sage.rings.padics} \\
&19^x - 7
\end{align*}
\]

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:

\[
\begin{align*}
sage: \ # \text{\textasciitilde needs sage.libs.pari sage.rings.real_mpfr} \\
sage: \ z &= \sqrt{\mathrm{RealField}(200)(2)} + (1/2)^33 \\
sage: \ p &= \text{algdep(}z, \ 4); \ p \\
&227004321085^x \ast 4 - 216947902586^x \ast 3 - 99411220986^x \ast 2 + 82234881648^x \ast x - 211871195088 \\
sage: \ \text{factor(}p) \\
&227004321085^x \ast 4 - 216947902586^x \ast 3 - 99411220986^x \ast 2 + 82234881648^x \ast x - 211871195088 \\
sage: \ \text{algdep(}z, \ 4, \ \text{known_bits}=32) \\
&x^2 - 2 \\
sage: \ \text{algdep(}z, \ 4, \ \text{known_digits}=10) \\
&x^2 - 2 \\
sage: \ \text{algdep(}z, \ 4, \ \text{use_bits}=25) \\
&x^2 - 2 \\
sage: \ \text{algdep(}z, \ 4, \ \text{use_digits}=8) \\
&x^2 - 2
\end{align*}
\]

Using the \texttt{height\_bound} and \texttt{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:

\[
\begin{align*}
sage: \ \text{algdep(}\pi.\text{n()}, \ 5, \ \text{height\_bound}=10, \ \text{proof=True}) \text{\ is None} \\
&\text{\textasciitilde needs sage.libs.pari sage.symbolic} \\
&\text{True}
\end{align*}
\]
For stronger results, we need more precision:

```
sage: # needs sage.libs.pari sage.symbolic
sage: algdep(pi.n(), 5, height_bound=100, proof=True) is None
Traceback (most recent call last):
... ValueError: insufficient precision for non-existence proof
sage: algdep(pi.n(200), 5, height_bound=100, proof=True) is None
True
sage: algdep(pi.n(), 10, height_bound=10, proof=True) is None
Traceback (most recent call last):
... ValueError: insufficient precision for non-existence proof
sage: algdep(pi.n(200), 10, height_bound=10, proof=True) is None
True
```

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

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

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

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

**INPUT:**

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

**EXAMPLES:**
We demonstrate each of the alternative algorithms:

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

AUTHOR:
• David Joyner and William Stein

\[
\binom{x}{m} = \frac{x(x-1)\cdots(x-m+1)}{m!}
\]

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:
Standard Commutative Rings, Release 10.2

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 1
sage: binomial(3, -1)
0
sage: binomial(20, 10)
184756
sage: binomial(-2, 5)
-6
sage: binomial(-5, -2)
˓→ needs sage.rings.real_mpfr 1
sage: binomial(RealField()('2.5'), 2)
˓→ needs sage.rings.real_mpfr 1.87500000000000
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

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 \{\( (k_1, k_2) : C_{k,n} \) \} where \( C_{k,n} \) are binomial coefficients and \( n = k_1 + k_2 \).

INPUT:
* n - an integer

OUTPUT: dict

1.13. Miscellaneous arithmetic functions 99
EXAMPLES:

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

Notice the coefficients above are the same as below:

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

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8  # needs numpy
sage: sorted(binomial_coefficients(int8(3)).items())  # needs numpy
[((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 \( a^k \equiv 1 \pmod{n} \) 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 \( \lambda(p^k) = p^k - 1 = \varphi(p^k) \). If \( k \geq 3 \), then \( \lambda(2^k) = 2^{k-2} \). Now consider the case where \( n > 3 \) is composite and let \( n = p_1^{h_1}p_2^{h_2} \cdots p_k^{h_k} \) be the prime factorization of \( n \). Then

\[
\lambda(n) = \lambda(p_1^{h_1}p_2^{h_2} \cdots p_k^{h_k}) = \text{lcm}(\lambda(p_1^{h_1}), \lambda(p_2^{h_2}), \ldots, \lambda(p_k^{h_k}))
\]

**EXAMPLES:**

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

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

```
sage: carmichael_lambda(2) == euler_phi(2)
True
sage: carmichael_lambda(4) == euler_phi(4)
needs sage.libs.pari
True
sage: p = random_prime(1000, lbound=3, proof=True)
needs sage.libs.pari
sage: k = randint(1, 1000)
sage: carmichael_lambda(p^k) == euler_phi(p^k)
needs sage.libs.pari
True
```

A case where \( \lambda(n) \neq \varphi(n) \):

```
sage: k = randint(3, 1000)
sage: carmichael_lambda(2^k) == 2^(k - 2)
needs sage.libs.pari
True
sage: carmichael_lambda(2^k) == 2^(k - 2) == euler_phi(2^k)
needs sage.libs.pari
False
```

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 \((\mathbb{Z}/n\mathbb{Z})^*\).

```
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)))
sage: def my_carmichael(n):
....:     if n == 1:
....:         return 1
....:     for k in range(1, n):
....:         L = znpower(n, k)
....:         ones = [1] * len(L)
....:         T = [L[i] == ones[i] for i in range(len(L))]
....:         if all(T):
....:             return k
sage: 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.
sage: from sage.arith.misc import carmichael_lambda
sage: n = randint(2, 1000)

sage: ZnZ = IntegerModRing(n)

sage: M = ZnZ.list_of_elements_of_multiplicative_group()

sage: ones = [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 = None or n > len(v) the default n = len(v) is used.

INPUT:

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

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

EXAMPLES:

sage: continuant([1, 2, 3])
10
sage: p = continuant([2, 1, 2, 1, 4, 1, 1, 6, 1, 8, 1, 1, 10])

sage: q = continuant([1, 2, 1, 4, 1, 1, 6, 1, 8, 1, 1, 10])

sage: p/q
517656/190435

sage: F = continued_fraction([2, 1, 2, 1, 4, 1, 1, 6, 1, 8, 1, 1, 10])

sage: F.convergent(14)
517656/190435

sage: x = PolynomialRing(RationalField(), 'x', 5).gens()

sage: continuant(x)
x0*x1*x2*x3*x4 + x0*x1*x2 + x0*x1*x4 + x0*x3*x4 + x2*x3*x4 + x0 + x2 + x4

sage: continuant(x, 3)
x0*x1*x2 + x0 + x2

sage: continuant(x, 2)
x0*x1 + 1

We verify the identity

\[ K_n(z, z, \ldots, z) = \sum_{k=0}^{n} \binom{n-k}{k} z^{n-2k} \]
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,z,z), 6)
z^6 + 5*z^4 + 6*z^2 + 1
sage: continuant(9)
Traceback (most recent call last):
  ...TypeError: object of type 'sage.rings.integer.Integer' has no len()
```

Tests with numpy and gmpy2 numbers:

```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)
```

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 \( a \equiv b \mod \gcd(m, n) \). 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 `crt` by giving it pairs of residues and moduli:

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

(continues on next page)
\begin{lstlisting}[language=Sage]
sage: crt([13, 20], [100, 301])
28013

You can also use upper case:

\begin{lstlisting}[language=Sage]
sage: c = CRT(2,3, 3, 5); c
8
sage: c % 3 == 2
True
sage: c % 5 == 3
True
\end{lstlisting}

Note that this also works for polynomial rings:

\begin{lstlisting}[language=Sage]
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*a + 3/26)*y^4 + (5/26*a - 5/26)*y^3 + (15/26*a - 15/26)*y + 27/52*a + 25/52
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{lstlisting}

You can also do this for any number of moduli:

\begin{lstlisting}[language=Sage]
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
\end{lstlisting}

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

\begin{lstlisting}[language=Sage]
\end{lstlisting}
crt also work with numpy and gmpy2 numbers:

```python
sage: import numpy
# needs numpy
sage: crt(numpy.int8(2), numpy.int8(3), numpy.int8(7), numpy.int8(11))  # needs numpy
58
```

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

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

Return the value of the Dedekind psi function at $N$.

**INPUT:**

- $N$ – a positive integer

**OUTPUT:**

an integer

The Dedekind psi function is the multiplicative function defined by

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

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

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

where

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

**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:** a rational number

Several small values:

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

Check relations for restricted arguments:

```
sage: q = 23; dedekind_sum(1, q); (q-1)*(q-2)/(12*q) # needs sage.libs.flint
77/46
77/46
```

(continues on next page)
sage: p, q = 100, 723  # must be coprime
sage: dedekind_sum(p, q) + dedekind_sum(q, p)
→ needs sage.libs.flint
31583/86760
sage: -1/4 + (p/q + q/p + 1/(p*q))/12
31583/86760

We check that evaluation works with large input:

sage: dedekind_sum(3^54 - 1, 2^93 + 1)
→ needs sage.libs.flint
459340694971839990630374299870/29710560942849126597578981379
sage: dedekind_sum(3^54 - 1, 2^93 + 1, algorithm='pari')
→ needs sage.libs.pari
459340694971839990630374299870/29710560942849126597578981379

We check consistency of the results:

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

Tests with numpy and gmpy2 numbers:

sage: from numpy import int8
→ needs numpy
sage: dedekind_sum(int8(5), int8(7), algorithm='default')
→ needs numpy sage.libs.flint
-1/14
sage: from gmpy2 import mpz
sage: dedekind_sum(mpz(5), mpz(7), algorithm='default')
→ needs sage.libs.flint
-1/14

REFERENCES:

• [Ap1997]
• Wikipedia article Dedekind_sum
sage.arith.misc.differences(lis, n=1)

Return the n successive differences of the elements in lis.

EXAMPLES:

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

Tests with numpy and gmpy2 numbers:

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

AUTHORS:

- Timothy Clemans (2008-03-09)

sage.arith.misc.divisors(n)

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

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

INPUT:

- n - the element

EXAMPLES:

Divisors of integers:

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

This function works whenever one has unique factorization:

sage: # needs sage.rings.number_field
sage: K.<a> = QuadraticField(7)

sage: divisors(K.ideal(7))
[Fractional ideal (1), Fractional ideal (a), Fractional ideal (7)]

sage: divisors(K.ideal(3))
[Fractional ideal (1), Fractional ideal (3), Fractional ideal (a - 2), Fractional ideal (a + 2)]

sage: divisors(K.ideal(35))
[Fractional ideal (1), Fractional ideal (5), Fractional ideal (a), Fractional ideal (7), Fractional ideal (5*a), Fractional ideal (35)]

sage.arith.misc.eratosthenes(n)

Return a list of the primes \( \leq n \).

This is extremely slow and is for educational purposes only.

INPUT:

• n - a positive integer

OUTPUT:

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

EXAMPLES:

sage: eratosthenes(3)
[2, 3]

sage: eratosthenes(50)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

sage: len(eratosthenes(100))
25

sage: eratosthenes(213) == prime_range(213)
#␣˓
→ 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, \texttt{n.factor(proof=proof, **kwds)} gets called. See \texttt{n.factor??} for more documentation in this case.
Warning: This means that applying factor() to an integer result of a symbolic computation will not factor
the integer, because it is considered as an element of a larger symbolic ring.

EXAMPLES:

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

INPUT:

- n – a nonzero integer
- proof – bool or None (default: None)
- int_ – bool (default: False) whether to return answers as Python ints
- algorithm – string
  - 'pari' – (default) use the PARI c library
  - 'kash' – use KASH computer algebra system (requires that kash be installed)
  - 'magma' – use Magma (requires magma be installed)
- verbose – integer (default: 0); PARI’s debug variable is set to this; e.g., set to 4 or 8 to see lots of output
during factorization.

OUTPUT:

- factorization of n

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

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

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

EXAMPLES:

```
sage: factor(500)
2^2 * 5^3
sage: factor(-20)
-1 * 2^2 * 5
sage: f=factor(-20)
sage: list(f)
[(2, 2), (5, 1)]
sage: f.unit()
```

(continues on next page)
-1
sage: f.value()
-20
sage: factor(-next_prime(10^2) * next_prime(10^7))
-1 * 101 * 10000019

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

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

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

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

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

To access the data in a factorization:

1.13. Miscellaneous arithmetic functions 111
We can factor Python, numpy and gmpy2 numbers:

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

```
sage.arith.misc.factorial(n, algorithm='gmp')
Compute the factorial of n, which is the product 1 · 2 · 3 · · · (n − 1) · n.

INPUT:
• n - an integer
• algorithm - string (default: ‘gmp’):
  – 'gmp' - use the GMP C-library factorial function
  – 'pari' - use PARI’s factorial function

OUTPUT: an integer

EXAMPLES:
```
```
True
sage: factorial(-32)
Traceback (most recent call last):
...
ValueError: factorial -- must be nonnegative

Tests with numpy and gmpy2 numbers:

```sage
sage: from numpy import int8

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

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

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

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

Return the falling factorial $\left(\begin{array}{c}x \\ a\end{array}\right)$.  

The notation in the literature is a mess: often $\left(\begin{array}{c}x \\ a\end{array}\right)$, but there are many other notations: GKP: Concrete Mathematics uses $x^a$.

**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
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'))
1310.1163396601
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)  # needs sage.modules
sage: A = M([1,0,1,0, 1,0,1,0, 1,0,10,10, 1,0,1,1])  # needs sage.modules
sage: falling_factorial(A, 2)  # A(A - I)
[ 1 0 10 10]
[ 1 0 10 10]
[ 20 0 101 100]
[ 2 0 11 10]

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

AUTHORS:

• Jaap Spies (2006-03-05)

sage.arith.misc.\texttt{four_squares}(n)

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

INPUT:

• $n$ – an integer

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

EXAMPLES:

sage: four_squares(3)
(0, 1, 1, 1)
sage: four_squares(13)
(0, 0, 2, 3)
sage: four_squares(130)
(0, 0, 3, 11)
sage: four_squares(1101011011004)
sage: four_squares(10^100 - 1)  # needs sage.libs.pari
(155024616290, 2612183768627, 14142135623730950488016887, 99999999999999999999999999999999999999999999999999)

sage: for i in range(2^129, 2^129 + 10000):  # long time  # needs sage.libs.pari
    ....: S = four_squares(i)
    ....: assert sum(x^2 for x in S) == i

sage.arith.misc.fundamental_discriminant(D)

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

INPUT:

- \( D \) - an integer

OUTPUT:

- an integer, the fundamental discriminant

EXAMPLES:

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

Tests with numpy and gmpy2 numbers:

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

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

Return the Gauss sums for a general finite field.

INPUT:

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

OUTPUT:

an element of the parent ring of \( char_value \), that can be any field containing enough roots of unity, for example the UniversalCyclotomicField, QQbar or ComplexField
For a finite field $F$ of characteristic $p$, the Gauss sum associated to a multiplicative character $\chi$ (with values in a ring $K$) is defined as

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

where $\zeta_p \in K$ is a primitive root of unity of order $p$ and $\text{Tr}$ 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:

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

```python
sage: # needs sage.libs.pari sage.rings.number_field
sage: F = GF(11**2); q = 11**2
sage: zq = UniversalCyclotomicField().zeta(q - 1)
sage: g = gauss_sum(zq**4, F)
sage: g*g.conjugate()
121
```

See also:

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

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

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

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

**INPUT:**

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

Additional keyword arguments are passed to the respectively called methods.

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

EXAMPLES:

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

Note that to take the gcd of \( n \) elements for \( n \neq 2 \) you must put the elements into a list by enclosing them in \([\ldots]\).

Before github issue #4988 the following wrongly returned 3 since the third parameter was just ignored:

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

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

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

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

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

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

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

EXAMPLES:

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

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

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

EXAMPLES:
sage.arith.misc.hilbert_conductor
def 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 Q.

    INPUT:
    • a, b – integers

    OUTPUT:
    squarefree positive integer

    EXAMPLES:

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

    Tests with numpy and gmpy2 numbers:

    sage: hilbert_conductor(int8(-3), int8(-17))
    17
    sage: hilbert_conductor(mpz(-3), mpz(-17))
    17

    AUTHOR:
    • Gonzalo Tornaria (2009-03-02)

sage.arith.misc.hilbert_conductor_inverse

def hilbert_conductor_inverse(d):
    Finds a pair of integers (a, b) such that hilbert_conductor(a, b) == d.

    The quaternion algebra (a, b) over Q will then have (reduced) discriminant d.

    INPUT:
    • d – square-free positive integer

    OUTPUT: pair of integers
EXAMPLES:

```python
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
```

AUTHOR:
• Gonzalo Tornaria (2009-03-02)

```
```

1.13. Miscellaneous arithmetic functions 119

```
```
sage: hilbert_symbol(3, -1, 2, algorithm='all')
-1
sage: hilbert_symbol(QQ(-1)/QQ(4), -1, 2) == -1  #→ needs sage.libs.pari
True
sage: hilbert_symbol(QQ(-1)/QQ(4), -1, 3) == 1  #→ needs sage.libs.pari
True

Tests with numpy and gmpy2 numbers:

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:

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

Tests with numpy and gmpy2 numbers:

sage: from numpy import float32
#→ needs numpy
sage: integer_ceil(float32(5.4))  #→ needs numpy
6
sage: from gmpy2 import mpfr
sage: integer_ceil(mpfr(5.4))
6

sage.arith.misc.integer_floor(x)
  Return the largest integer ≤ 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
sage: integer_floor(x)
# needs sage.symbolic
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  # needs numpy
sage: integer_floor(float32(5.4))
5
sage: from gmpy2 import mpfr
sage: integer_floor(mpfr(5.4))
5
```

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

Truncate to the integer closer to zero

EXAMPLES:

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

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

The inverse of the ring element $a$ modulo $m$.

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

```python
sage: inverse_mod(7, 1)
0
sage: inverse_mod(5, 14)
3
sage: inverse_mod(3, -5)
2
```
Tests with numpy and mpz numbers:

```python
sage: from numpy import int8
˓→needs numpy
sage: inverse_mod(int8(5), int8(14))  #→ needs numpy
3
sage: from gmpy2 import mpz
sage: inverse_mod(mpz(5), mpz(14))
3
```

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

Return whether `n` is a power of 2.

**INPUT:**

- `n` – integer

**OUTPUT:**

boolean

**EXAMPLES:**

```python
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:

```python
sage: from numpy import int8
˓→needs numpy
sage: is_power_of_two(int8(16))  #→ needs numpy
True
sage: is_power_of_two(int8(24))  #→ needs numpy
False
sage: from gmpy2 import mpz
sage: is_power_of_two(mpz(16))
True
sage: is_power_of_two(mpz(24))
False
```

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

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
sage: is_prime(-1)
False
sage: is_prime(1)
False
sage: is_prime(-2)
False
```

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

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

Test whether \( n \) is a positive power of a prime number.

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

**INPUT:**

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

**EXAMPLES:**

```
sage: # needs sage.libs.pari
sage: is_prime_power(389)
True
sage: is_prime_power(2000)
```

(continues on next page)
False
sage: is_prime_power(2)
True
sage: is_prime_power(1024)
True
sage: is_prime_power(1024, get_data=True)
(2, 10)

The same results can be obtained with:

sage: # 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)

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

sage.arith.misc.is_pseudoprime_power(n, get_data=False)
Test if n is a power of a pseudoprime.
The result is *NOT* proven correct - *this IS a pseudo-primality test!* Note that a prime power is a positive power of a prime number so that 1 is not a prime power.

**INPUT:**

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

**EXAMPLES:**

```plaintext
sage: # needs sage.libs.pari
sage: is_pseudoprime_power(389)
True
sage: is_pseudoprime_power(2000)
False
sage: is_pseudoprime_power(2)
True
sage: is_pseudoprime_power(1024)
True
sage: is_pseudoprime_power(-1)
False
sage: is_pseudoprime_power(1)
False
sage: is_pseudoprime_power(997^100)
True
```

Use of the get_data keyword:

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

```plaintext
sage: from numpy import int16  # needs numpy
sage: is_pseudoprime_power(int16(1024))  # needs numpy sage.libs.pari
True
sage: from gmpy2 import mpz
sage: is_pseudoprime_power(mpz(1024))
True
```

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

Return whether or not \(n\) is square.

If \(n\) is a square also return the square root. If \(n\) is not square, also return None.

**INPUT:**

1.13. Miscellaneous arithmetic functions
• \( n \) – an integer
• \( \text{root} \) – whether or not to also return a square root (default: False)

OUTPUT:
• bool – whether or not a square
• \( \text{object} \) – (optional) an actual square if found, and None otherwise.

EXAMPLES:

```
sage: is_square(2)
False
sage: is_square(4)
True
sage: is_square(2.2)
True
sage: is_square(-2.2)
False
sage: is_square(CDF(-2.2))
# needs sage.rings.complex_double
True
sage: is_square((x-1)^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)
```

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
```

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
```

```
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
```

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

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

```python
sage: 0 = ZZ[sqrt(-5)] # needs sage.rings.number_field sage.symbolic
sage: a = 0.gen(1) # needs sage.rings.number_field sage.symbolic
sage: is_squarefree(a - 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: # needs sage.libs.pari
sage: from numpy import int8 # needs numpy
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
```

(continues on next page)
sage: is_squarefree(mpz(101))
True

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

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

If
\[ b = p_1^{e_1} \ldots p_r^{e_r} \]
then
\[ (a|b) = (a|p_1)^{e_1} \ldots (a|p_r)^{e_r} \]
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
```

Tests with numpy and gmpy2 numbers:

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

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

**INPUT:**
- x – integer
- y – integer

**OUTPUT:**
- an integer
EXAMPLES:

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

This is also available as $\text{kroncker()}$:

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

Tests with numpy and gmpy2 numbers:

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

```
sage.arith.misc.kronecker_symbol(x, y)
The Kronecker symbol $(x|y)$.

INPUT:

- $x$ – integer
- $y$ – integer

OUTPUT:

- an integer

EXAMPLES:

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

This is also available as $\text{kroncker()}$:

```
sage: kroncker(3,5)
-1
sage: kroncker(3,15)
0
```

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

```sage
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
```

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

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

**INPUT:**
- $x$ - integer
- $p$ - an odd prime number

**EXAMPLES:**

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

Tests with numpy and gmpy2 numbers:

```sage
sage: from numpy import int8   # needs numpy
sage: legendre_symbol(int8(2), int8(3))   # needs numpy
```

(continues on next page)
from gmpy2 import mpz
sage: legendre_symbol(mpz(2),mpz(3))
-1

sage.arith.misc.mqrr_rational_reconstruction(u, m, T)
Maximal Quotient Rational Reconstruction.
For research purposes only - this is pure Python, so slow.

INPUT:
• u, m, T - integers such that m > u ≥ 0, T > 0.

OUTPUT:
Either integers n, d such that d > 0, gcd(n, d) = 1, n/d = u mod m, and T · d · |n| < m, or None.

Reference: Monagan, Maximal Quotient Rational Reconstruction: An Almost Optimal Algorithm for Rational
Reconstruction (page 11)
This algorithm is probabilistic.

EXAMPLES:
p = sage: mqrr_rational_reconstruction(21, 3100, 13)
(21, 1)

Tests with numpy and gmpy2 numbers:
p = sage: from numpy import int16 #
# needs numpy
sage: mqrr_rational_reconstruction(int16(21), int16(3100), int16(13)) #
# needs numpy
(21, 1)
p = sage: from gmpy2 import mpz
sage: mqrr_rational_reconstruction(mpz(21), mpz(3100), mpz(13))
(21, 1)

sage.arith.misc.multinomial(*ks)
Return the multinomial coefficient.

INPUT:
• either an arbitrary number of integer arguments k₁,...,kₙ
• or an iterable (e.g. a list) of integers [k₁,...,kₙ]

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:
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
˓
AUTHORS:
• Gabriel Ebner

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

INPUT:
• m - integer
• n - integer

OUTPUT: dict

EXAMPLES:

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

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

```
sage: R.<x,y> = QQ[

x^5 + 5*x^4*y + 10*x^3*y^2 + 10*x^2*y^3 + 5*x*y^4 + y^5

```

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

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

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

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

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

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

**INPUT:**

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

**EXAMPLES:**

```python
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
```

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

```python
sage: next_prime(5) # needs sage.libs.pari
7
sage: next_prime(2004) # needs sage.libs.pari
2011
```

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

Return the smallest prime power greater than *n*. Note that if *n* is a prime power, then this function does not return *n*, but the next prime power after *n*. This function just calls the method **Integer.next_prime_power()** of Integers.

**See also:**

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

```
sage: # 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
```

The same results can be obtained with:

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

Note that 2 is the smallest prime power:

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

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

INPUT:
  • n - an integer

EXAMPLES:
```
sage: # 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)
15525180923007089351489794884625025552568860171166966111390520380260509526863768633047840882864647
```

```
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
```

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

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

INPUT:
• n - a nonzero integer

OUTPUT:
• an integer, the number of divisors of n

EXAMPLES:

```
sage: number_of_divisors(100)  # needs sage.libs.pari
9
sage: number_of_divisors(-720)  # needs sage.libs.pari
30
```

Tests with numpy and gmpy2 numbers:

```
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
```

```
sage.arith.misc.odd_part(n)
The odd part of the integer n. This is n/2^v, where v = valuation(n, 2).

EXAMPLES:
```
Tests with numpy and gmpy2 numbers:

```
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:

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

This function works for fairly general rings:

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

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

The largest prime < \( n \). The result is provably correct. If \( n \leq 1 \), 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)
(continues on next page)
sage: previous_prime(7)
5
sage: previous_prime(5)
3
sage: previous_prime(3)
2
sage: previous_prime(2)
Traceback (most recent call last):
  ... ValueError: no previous prime
sage: previous_prime(1)
Traceback (most recent call last):
  ... ValueError: no previous prime
sage: previous_prime(-20)
Traceback (most recent call last):
  ... ValueError: no previous prime

sage.arith.misc.previous_prime_power(n)

Return the largest prime power smaller than n.

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

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

See also:

• is_prime_power() (and Integer.is_prime_power())
• next_prime_power() (and Integer.next_prime_power())

EXAMPLES:

sage: # 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

The same results can be obtained with:

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

Input less than or equal to 2 raises errors:

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

sage: n = previous_prime_power(2^16 - 1) # 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

sage.arith.misc.prime_divisors(n)
Return the list of prime divisors (up to units) of this element of a unique factorization domain.

INPUT:

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

OUTPUT:

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

EXAMPLES:

Prime divisors of positive integers:

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

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

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

For polynomials we get all irreducible factors:

sage: R.<x> = PolynomialRing(QQ)
sage: prime_divisors(x^12 - 1) # 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: 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]
```

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

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

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

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

**EXAMPLES:**
Prime divisors of positive integers:

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

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

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

For polynomials we get all irreducible factors:

```
sage: R.<x> = PolynomialRing(QQ)
sage: prime_divisors(x^12 - 1)
    # 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: 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]
```
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(b)
194
sage: a[:10]  # needs sage.libs.pari
[97, 101, 103, 107, 109, 113, 121, 125, 127, 128]
sage: b[:10]  # needs sage.libs.pari
[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]
```

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:**

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

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

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

Tests with numpy and gmpy2 numbers:

```python
sage: from numpy import int16
c−→ needs numpy
sage: prime_to_m_part(int16(240), int16(2))
15
sage: from gmpy2 import mpz
c−→ needs gmpy2
sage: prime_to_m_part(mpz(240), mpz(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 (optional, 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
sage: # needs sage.libs.pari
sage: for p in primes(5, 10):
    ....:     print(p)
5
7
sage: list(primes(13))
[2, 3, 5, 7, 11]
sage: list(primes(10000000000, 10000000100))
[10000000019, 10000000033, 10000000061, 10000000069, 10000000097]
sage: max(primes(10^100, 10^100+10^4, proof=False))
100000000009631
sage: next(p for p in primes(10^20, infinity) if is_prime(2*p+1))
100000000000000001243
```

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

Return the first `n` primes.

INPUT:

• `n` - a nonnegative integer

OUTPUT:

• a list of the first `n` prime numbers.

EXAMPLES:

```python
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)
[]
```

`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:

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

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

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

sage.arith.misc.quadratic_residues($n$)
Return a sorted list of all squares modulo the integer $n$ in the range $0 \leq x < |n|$.

EXAMPLES:

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

Tests with numpy and gmpy2 numbers:

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

(continues on next page)
sage: quadratic_residues(mpz(11))
[0, 1, 3, 4, 5, 9]

sage.arith.misc.radical(n, *args, **kwds)
Return the product of the prime divisors of n.
This calls n.radical(*args, **kwds).

EXAMPLES:

sage: radical(2 * 3^2 * 5^5)
30
sage: radical(0)
Traceback (most recent call last):
  ... ArithmeticError: radical of 0 is not defined

Tests with numpy and gmpy2 numbers:

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

sage.arith.misc.random_prime(n, proof=None, lbound=2)
Return a random prime \( p \) between \( lbound \) and \( n \).
The returned prime \( p \) satisfies \( lbound \leq p \leq n \).
The returned prime \( p \) is chosen uniformly at random from the set of prime
numbers less than or equal to \( n \).

INPUT:

- \( n \) - an integer \( \geq 2 \).
- \( proof \) - 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:

sage: # needs sage.libs.pari
sage: p = random_prime(100000)
sage: p.is_prime()  
True
sage: p <= 100000
(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 \( |d| \leq \sqrt{m}/2 \). Return \((0, 0)\) if no such number exists.

The algorithm for rational reconstruction is described (with a complete nontrivial proof) on pages 656-657 of Knuth, Vol 2, 3rd ed. as the solution to exercise 51 on page 379. See in particular the conclusion paragraph right in the middle of page 657, which describes the algorithm thus:

This discussion proves that the problem can be solved efficiently by applying Algorithm 4.5.2X with \( u = m \) and \( v = a \), but with the following replacement for step X2: If \( v3 \leq \sqrt{m}/2 \), the algorithm terminates. The pair \((x, y) = (|v2|, v3 \ast \text{sign}(v2))\) is then the unique solution, provided that \( x \) and \( y \)
are coprime and \( x \leq \sqrt{m/2} \); otherwise there is no solution. (Alg 4.5.2X is the extended Euclidean algorithm.)

Knuth remarks that this algorithm is due to Wang, Kornerup, and Gregory from around 1983.

**EXAMPLES:**

```python
sage: m = 100000
sage: (119*inverse_mod(53,m))%m
11323
sage: rational_reconstruction(11323,m)
119/53
sage: rational_reconstruction(400,1000)
Traceback (most recent call last):
... ArithmeticError: rational reconstruction of 400 (mod 1000) does not exist
sage: rational_reconstruction(3, 292393)
3
sage: a = Integers(292393)(45/97); a
204977
sage: rational_reconstruction(a, 292393, algorithm='fast')
45/97
sage: rational_reconstruction(293048, 292393)
Traceback (most recent call last):
... ArithmeticError: rational reconstruction of 655 (mod 292393) does not exist
sage: rational_reconstruction(0, 0)
Traceback (most recent call last):
... ZeroDivisionError: rational reconstruction with zero modulus
sage: rational_reconstruction(0, 1, algorithm='foobar')
Traceback (most recent call last):
... ValueError: unknown algorithm 'foobar'
```

Tests with numpy and gmpy2 numbers:

```python
sage: from numpy import int32
sage: rational_reconstruction(int32(3), int32(292393))
3
sage: from gmpy2 import mpz
sage: rational_reconstruction(mpz(3), mpz(292393))
3
```

```python
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 \(x^{\underline{a}}\).

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:

\[
\begin{align*}
\text{- } & \text{ } x \text{ – element of a ring} \\
\text{- } & \text{ } a \text{ – a non-negative integer or} \\
\text{- } & \text{ } x \text{ and } a \text{ – any numbers}
\end{align*}
\]

OUTPUT: the rising factorial

See also:

\textit{falling_factorial()} 

EXAMPLES:

\texttt{sage: rising_factorial(10,3)}
\begin{verbatim}
1320
\end{verbatim}

\texttt{sage: \# needs sage.symbolic}
\texttt{sage: rising_factorial(10, RR('3.0'))}
\begin{verbatim}
1320.00000000000
\end{verbatim}

\texttt{sage: rising_factorial(10, RR('3.3'))}
\begin{verbatim}
2826.38895824964
\end{verbatim}

\texttt{sage: a = rising_factorial(1+I, I); a}
\begin{verbatim}
gamma(2*I + 1)/gamma(I + 1)
\end{verbatim}

\texttt{sage: CC(a)}
\begin{verbatim}
0.266816390637832 + 0.122783354006372*I
\end{verbatim}

\texttt{sage: a = rising_factorial(I, 4); a}
\begin{verbatim}
-10
\end{verbatim}

\texttt{sage: x = polygen(ZZ)}
\texttt{sage: rising_factorial(x, 4)}
\begin{verbatim}
x^4 + 6*x^3 + 11*x^2 + 6*x
\end{verbatim}

AUTHORS:

\[
\begin{align*}
\text{- } & \text{ } \text{Jaap Spies (2006-03-05)}
\end{align*}
\]

\texttt{sage.arith.misc.sort_complex_numbers_for_display(nums)}

Given a list of complex numbers (or a list of tuples, where the first element of each tuple is a complex number), we sort the list in a “pretty” order.

Real numbers (with a zero imaginary part) come before complex numbers, and are sorted. Complex numbers are sorted by their real part unless their real parts are quite close, in which case they are sorted by their imaginary part.

This is not a useful function mathematically (not least because there is no principled way to determine whether the real components should be treated as equal or not). It is called by various polynomial root-finders; its purpose is to make doctest printing more reproducible.

We deliberately choose a cumbersome name for this function to discourage use, since it is mathematically meaningless.

EXAMPLES:
sage: # 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)]
sage: for i in range(3):
    ....:     nums.append(CDF(i + RDF.random_element(-3e-11, 3e-11),
    ....:         RDF.random_element()))
    ....:     nums.append(CDF(i + RDF.random_element(-3e-11, 3e-11),
    ....:         RDF.random_element()))
sage: shuffle(nums)
sage: nums = sort_c(nums)
sage: for i in range(len(nums)):
    ....:     if nums[i].imag():
    ....:         first_non_real = i
    ....:         break
    ....:     else:
    ....:         first_non_real = len(nums)
sage: assert first_non_real >= 3
sage: for i in range(first_non_real - 1):
    ....:     assert nums[i].real() <= nums[i + 1].real()
sage: def truncate(n):
    ....:     if n.real() < 1e-10:
    ....:         return 0
    ....:     else:
    ....:         return n.real().n(digits=9)
sage: for i in range(first_non_real, len(nums)-1):
    ....:     assert truncate(nums[i]) <= truncate(nums[i + 1])
    ....:     if truncate(nums[i]) == truncate(nums[i + 1]):
    ....:         assert nums[i].imag() <= nums[i+1].imag()

sage.arith.misc.squarefree_divisors(x)

Return an iterator over the squarefree divisors (up to units) of this ring element.

Depends on the output of the prime_divisors function.

Squarefree divisors of an integer are not necessarily yielded in increasing order.

INPUT:

* x – an element of any ring for which the prime_divisors function works.

EXAMPLES:

Integers with few prime divisors:

sage: list(squarefree_divisors(7))
[1, 7]
sage: list(squarefree_divisors(6))
[1, 2, 3, 6]
sage: list(squarefree_divisors(12))
[1, 2, 3, 6]

Squarefree divisors are not yielded in increasing order:

sage: list(squarefree_divisors(30))
[1, 2, 3, 6, 5, 10, 15, 30]
sage.arith.misc.subfactorial(n)

Subfactorial or rencontres numbers, or derangements: number of permutations of \( n \) elements with no fixed points.

INPUT:

• \( n \) - non negative integer

OUTPUT:

• integer - function value

EXAMPLES:

```
sage: subfactorial(0)
1
sage: subfactorial(1)
0
sage: subfactorial(8)
14833
```

Tests with numpy and gmpy2 numbers:

```
sage: from numpy import int8
sage: subfactorial(int8(8))
14833
sage: from gmpy2 import mpz
sage: subfactorial(mpz(8))
14833
```

AUTHORS:

• Jaap Spies (2007-01-23)

sage.arith.misc.sum_of_k_squares(k, n)

Write the integer \( n \) as a sum of \( k \) integer squares if possible; otherwise raise a \texttt{ValueError}.

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)
(19215400822645944253860920437586326284, 37204645194585992174252915693267578306, #
    needs sage.libs.pari
```

(continues on next page)
(continued from previous page)

\[
\begin{array}{c}
3473564819477394665857484221256136567800161086815834297092488779216863122,
\rightarrow 5860191799617673633547572610351797996721850737768032876360978911074629287841061578270832330322236796556721252602860754789786937515870682024273948,
\rightarrow 204574232945581824940019198123790239925388022037307910197285434397653478513163665370946968966991
\rightarrow 3116280954116781598492377386194583964975346960435809122253342693716118369103459303207008166496534
\end{array}
\]

```python
sage: sum_of_k_squares(7, 0)
(0, 0, 0, 0, 0, 0, 0)
sage: sum_of_k_squares(30, 999999)
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 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
```

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
(15, 97)
sage: from gmpy2 import mpz
sage: sum_of_k_squares(mpz(2), mpz(9634))
(15, 97)
```

`sage.arith.misc.three_squares(n)`

Write the integer \( n \) as a sum of three integer squares if possible; otherwise raise a `ValueError`. 
INPUT:

- n – an integer

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

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, 1798465042647412146620280340569649349251249)
sage: three_squares(11^111 - 1)
# needs sage.libs.pari
(616274160655975340150706442680, 90158293835735143295060746161,
  62703823876357441403940011363065331196796409998178859397233)
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
```

ALGORITHM:

See https://schorn.ch/lagrange.html

```
sage.arith.misc.trial_division(n, bound=None)
```

Return the smallest prime divisor \(\leq\) bound of the positive integer \(n\), or \(n\) if there is no such prime. If the optional argument \(bound\) is omitted, then \(bound \leq n\).

INPUT:

- \(n\) - a positive integer
- \(bound\) - (optional) a positive integer

OUTPUT:

- \(int\) - a prime \(p=bound\) that divides \(n\), or \(n\) if there is no such prime.

EXAMPLES:

```
sage: trial_division(15)
3
sage: trial_division(91)
7
sage: trial_division(11)
```

(continues on next page)
sage: trial_division(387833, 300)
387833

sage: # 300 is not big enough to split off a
sage: # factor, but 400 is.
sage: trial_division(387833, 400)
389

Tests with numpy and gmpy2 numbers:

sage: from numpy import int8
   #→ needs numpy
sage: trial_division(int8(91))
   #→ needs numpy
7

sage: from gmpy2 import mpz
sage: trial_division(mpz(91))
7

sage.arith.misc.two_squares(n)

Write the integer \(n\) as a sum of two integer squares if possible; otherwise raise a ValueError.

INPUT:

• \(n\) – an integer

OUTPUT: a tuple \((a, b)\) of non-negative integers such that \(n = a^2 + b^2\) with \(a \leq b\).

EXAMPLES:

sage: two_squares(389)
(10, 17)
sage: two_squares(21)
Traceback (most recent call last):
...
ValueError: 21 is not a sum of 2 squares
sage: two_squares(21^2)
(0, 21)
sage: a, b = two_squares(10000000000000000129); a, b
   #→ needs sage.libs pari
(4418521500, 8970878873)
sage: a^2 + b^2
   #→ needs sage.libs pari
10000000000000000129
sage: two_squares(2^2*2 + 1)
   #→ 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

ALGORITHM:
sage.arith.misc.valuation(m, *args, **kwds)

Return the valuation of \( m \).

This function simply calls the \( m.\text{valuation}() \) method. See the documentation of \( m.\text{valuation}() \) for a more precise description.

Note that the use of this function is discouraged as it is better to use \( m.\text{valuation}() \) directly.

Note: This is not always a valuation in the mathematical sense. For more information see: sage.rings.finite_rings.integer_mod.IntegerMod_int.valuation

EXAMPLES:

```python
sage: valuation(512,2)
9
sage: valuation(1,2)
0
sage: valuation(5/9, 3)
-2
```

Valuation of 0 is defined, but valuation with respect to 0 is not:

```python
sage: valuation(0,7)
+Infinity
```

```python
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:

```python
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):
```

(continues on next page)
...ValueError: You can only compute the valuation with respect to a integer larger than 1.
sage: from numpy import int16
# needs numpy
sage: valuation(int16(512), int16(2))  
# needs numpy
9
sage: from mpmath import mpz
sage: valuation(mpz(512), mpz(2))
9

sage.arith.misc.xgcd(a, b)

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

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

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

**OUTPUT:**
- \(g, s, t\) - such that \(g = s \cdot a + t \cdot b\)

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

**EXAMPLES:**

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

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

sage: x = polygen(QQ)
sage: xgcd(x^3 - 1, x^2 - 1)
(x - 1, 1, -x)
```
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}
sage: R.<x> = ZZ[]
sage: gcd(2*x*(x-1), x^2)
x
sage: xgcd(2*x*(x-1), x^2)
(2*x, -1, 2)
sage: (2*(x-1)).resultant(x) # needs sage.libs.pari
2
\end{Verbatim}

Tests with numpy and \texttt{gmpy2} types:

\begin{Verbatim}
sage: from numpy import int8

# needs numpy
sage: xgcd(4, int8(8))
(4, 1, 0)

# needs numpy
sage: xgcd(int8(4), int8(8))
(4, 1, 0)

# needs numpy
sage: from gmpy2 import mpz
sage: xgcd(mpz(4), mpz(8))
(4, 1, 0)

sage: xgcd(4, mpz(8))
(4, 1, 0)
\end{Verbatim}

\texttt{sage.arith.misc.xkcd}(\texttt{n}="")

This function is similar to the \texttt{xgcd} function, but behaves in a completely different way.

See https://xkcd.com/json.html for more details.

INPUT:

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

OUTPUT: a fragment of HTML

EXAMPLES:

\begin{Verbatim}
sage: xkcd(353) # optional - internet
<h1><img src="https://imgs.xkcd.com/comics/python.png" title="I wrote 20 short programs in Python yesterday. It was wonderful. Perl, I'm leaving you."/>
\end{Verbatim}
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
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_mpfr
22/7

sage: QQ(RealField().pi())          # needs 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
```

AUTHORS:

- Niles Johnson (2010-08): github issue #3893: $\texttt{random_element()}$ should pass on $\texttt{*args}$ and $\texttt{**kwds}$.
- Travis Scrimshaw (2012-10-18): Added additional docstrings for full coverage. Removed duplicates of $\texttt{discriminant()}$ and $\texttt{signature()}$.
- Anna Haensch (2018-03): Added function $\texttt{quadratic_defect()}$
class \texttt{sage.rings.rational_field.RationalField}

Bases: \texttt{Singleton}, \texttt{NumberField}

The class \texttt{RationalField} represents the field $\mathbb{Q}$ of rational numbers.

EXAMPLES:

```python
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(39029384023840928309482842098430284398243982394))
39029384023840928309482842098430284398243982394
sage: QQ(RR(3929329/32))
3929329/32
sage: QQ(-RR(3929329/32))
-3929329/32
sage: QQ(RR(1/7)) - 1/7
0
```

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

```python
sage: QQ(RR(3929329/32))
3929329/32
sage: QQ(-RR(3929329/32))
-3929329/32
sage: QQ(RR(1/7)) - 1/7
0
```

If you specify the optional second argument \texttt{base}, then the string representation of the float is used.

```python
sage: # needs sage.rings.real_mpfr
sage: QQ(23.2, 2)
6530219459687219/281474976710656
sage: 6530219459687219.0/281474976710656
23.2000000000000
sage: a = 23.2; a
23.2000000000000
sage: QQ(a, 10)
116/5
```

Here's a nice example involving elliptic curves:
absolute_degree()  
Return the absolute degree of \( Q \), which is 1.

EXAMPLES:

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

absolute_discriminant()  
Return the absolute discriminant, which is 1.

EXAMPLES:

```sage
sage: QQ.absolute_discriminant()
1
```

absolute_polynomial()  
Return a defining polynomial of \( Q \), as for other number fields.

This is also aliased to defining_polynomial() and absolute_polynomial().

EXAMPLES:

```sage
sage: QQ.polynomial()
x
```

algebraic_closure()  
Return the algebraic closure of \( Q \) (which is \( \overline{Q} \)).

EXAMPLES:

```sage
sage: QQ.algebraic_closure()  # needs sage.rings.number_field
Algebraic Field
```

automorphisms()  
Return all Galois automorphisms of \( Q \).

OUTPUT: a sequence containing just the identity morphism

EXAMPLES:

```sage
sage: QQ.automorphisms()
[ ]
Ring endomorphism of Rational Field
```

(continues on next page)
characteristic()

Return 0 since the rational field has characteristic 0.

EXAMPLES:

```
sage: c = QQ.characteristic(); c
0
sage: parent(c)
Integer Ring
```

class_number()

Return the class number of the field of rational numbers, which is 1.

EXAMPLES:

```
sage: QQ.class_number()
1
```

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

Return the completion of $\mathbb{Q}$ 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
```

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
```

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
defining_polynomial()
return a defining polynomial of \( \mathbb{Q} \), as for other number fields.
This is also aliased to `defining_polynomial()` and `absolute_polynomial()`.
```

```sage
sage: QQ.polynomial()
x
```

```sage
definition_of_degree()
Return the degree of \( \mathbb{Q} \), which is 1.
```

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

```sage
definition_of_discriminant()
Return the discriminant of the field of rational numbers, which is 1.
```

```sage
sage: QQ.discriminant()
1
```

```sage
definition_of_embeddings(K)
Return list of the one embedding of \( \mathbb{Q} \) into \( K \), if it exists.
```

```sage
sage: QQ.embeddings(QQ)
[Identity endomorphism of Rational Field]
sage: QQ.embeddings(CyclotomicField(5))
[Coercion map:
    From: Rational Field
    To:   Cyclotomic Field of order 5 and degree 4]
```

\( K \) must have characteristic 0:

```sage
sage: QQ.embeddings(GF(3))
Traceback (most recent call last):
  ... ValueError: no embeddings of the rational field into \( K \).
```

```sage
definition_of_extension(poly, names, **kwds)
Create a field extension of \( \mathbb{Q} \).
```

```sage
EXAMPLES:
We make a single absolute extension:
```
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

```python
sage: b^2
-3
```

```python
sage: a^3
-5
```

**gen**\((n=0)\)

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

There is only the 0-th generator, which is 1.

**EXAMPLES:**

```python
sage: QQ.gen()
1
```

**gens()**

Return a tuple of generators of \(Q\), 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 \(Q\) or as \(-1\)
- \(b\) – a non-zero rational number which is a non-square locally at every prime in \(S\).
- \(check\) – bool (default: True) perform additional checks on input and confirm the output.

**OUTPUT:**

- An integer \(a\) that has negative Hilbert symbol \((a, b)_p\) for every place \(p\) in \(S\) and no other place.

**EXAMPLES:**

```python
sage: QQ.hilbert_symbol_negative_at_S([-1,5,3,2,7,11,13,23], -10/7)
```

(continues on next page)
AUTHORS:

• Simon Brandhorst, Juanita Duque, Anna Haensch, Manami Roy, Sandi Rudzinski (10-24-2017)

is_absolute()  
Q is an absolute extension of Q.

EXAMPLES:

```
sage: QQ.is_absolute()
True
```

is_prime_field()  
Return True since Q is a prime field.

EXAMPLES:

```
sage: QQ.is_prime_field()
True
```

maximal_order()  
Return the maximal order of the rational numbers, i.e., the ring Z of integers.

EXAMPLES:

```
sage: QQ.maximal_order()
Integer Ring
sage: QQ.ring_of_integers()
Integer Ring
```

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

EXAMPLES:

```
sage: QQ.ngens()
1
```

number_field()  
Return the number field associated to Q. Since Q is a number field, this just returns Q again.

EXAMPLES:

```
sage: QQ.number_field() is QQ
True
```

2.1. Field Q of Rational Numbers
order()  
Return the order of \( \mathbb{Q} \), which is \( \infty \).

EXAMPLES:

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

places(all_complex=False, prec=None)  
Return the collection of all infinite places of \( \text{self} \), which in this case is just the embedding of \( \text{self} \) into \( \mathbb{R} \).

By default, this returns homomorphisms into \( \mathbb{R} \). If \( \text{prec} \) is not None, we simply return homomorphisms into \( \text{RealField(prec)} \) (or \( \text{RDF} \) if \( \text{prec}=53 \)).

There is an optional flag all_complex, which defaults to False. If all_complex is True, then the real embeddings are returned as embeddings into the corresponding complex field.

For consistency with non-trivial number fields.

EXAMPLES:

```sage
sage: QQ.places()  # needs sage.rings.real_mpfr
[Ring morphism:  
  From: Rational Field  
  To:   Real Field with 53 bits of precision  
  Defn: 1 |--> 1.0000000000000000000000000000000000000000000000000000000000]
sage: QQ.places(prec=53)  
[Ring morphism:  
  From: Rational Field  
  To:   Real Double Field  
  Defn: 1 |--> 1.0]
sage: QQ.places(prec=200, all_complex=True)  # needs sage.rings.real_mpfr
[Ring morphism:  
  From: Rational Field  
  To:   Complex Field with 200 bits of precision  
  Defn: 1 |--> 1.0000000000000000000000000000000000000000000000000000000000]
```

polynomial()  
Return a defining polynomial of \( \mathbb{Q} \), as for other number fields.

This is also aliased to \( \text{defining_polynomial()} \) and \( \text{absolute_polynomial()} \).

EXAMPLES:

```sage
sage: QQ.polynomial()
x
```

power_basis()  
Return a power basis for this number field over its base field.

The power basis is always \([1] \) for the rational field. This method is defined to make the rational field behave more like a number field.

EXAMPLES:
sage: QQ.power_basis()
[1]

primes_of_bounded_norm_iter($B$)
Iterator yielding all primes less than or equal to $B$.

INPUT:
• $B$ – a positive integer; upper bound on the primes generated.

OUTPUT:
An iterator over all integer primes less than or equal to $B$.

Note: This function exists for compatibility with the related number field method, though it returns prime integers, not ideals.

EXAMPLES:

```python
sage: it = QQ.primes_of_bounded_norm_iter(10)
sage: list(it)  
# needs sage.libs.pari
[2, 3, 5, 7]
sage: list(QQ.primes_of_bounded_norm_iter(1))
[]
```

quadratic_defect($a, p$, check=True)
Return the valuation of the quadratic defect of $a$ at $p$.

INPUT:
• $a$ – an element of self
• $p$ – a prime ideal or a prime number
• check – (default: True); check if $p$ is prime

REFERENCE:
[Kir2016]

EXAMPLES:

```python
sage: QQ.quadratic_defect(0, 7)
+Infinity
sage: QQ.quadratic_defect(5, 7)
0
sage: QQ.quadratic_defect(5, 2)
2
sage: QQ.quadratic_defect(5, 5)
1
```

random_element($num\_bound=None$, $den\_bound=None$, *args, **kwds)
Return a random element of $Q$.

Elements are constructed by randomly choosing integers for the numerator and denominator, not necessarily coprime.

INPUT:
• num_bound – a positive integer, specifying a bound on the absolute value of the numerator. If absent, no bound is enforced.
• den_bound – a positive integer, specifying a bound on the value of the denominator. If absent, the bound for the numerator will be reused.

Any extra positional or keyword arguments are passed through to `sage.rings.integer_ring.IntegerRing_class.random_element()`.

**EXAMPLES:**

```python
sage: QQ.random_element().parent() is QQ
True
sage: while QQ.random_element() != 0:
    ....:   pass
sage: while QQ.random_element() != -1/2:
    ....:   pass
```

In the following example, the resulting numbers range from -5/1 to 5/1 (both inclusive), while the smallest possible positive value is 1/10:

```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
```

Extra positional or keyword arguments are passed through:

```python
sage: QQ.random_element(distribution='1/n').parent() is QQ
True
sage: QQ.random_element(distribution='1/n').parent() is QQ
True
```

### range_by_height(start, end=None)

Range function for rational numbers, ordered by height.

Returns a Python generator for the list of rational numbers with heights in `range(start, end)`. Follows the same convention as Python `range()`, type `range?` for details.

See also `__iter__()`.

**EXAMPLES:**

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:

```
sage: list(QQ.range_by_height(3, 3))
[]
sage: list(QQ.range_by_height(10, 1))
[]
```

There are no rational numbers with height \( \leq 0 \):

```
sage: list(QQ.range_by_height(-10, 1))
[]
```

**relative_discriminant()**

Return the relative discriminant, which is 1.

EXAMPLES:

```
sage: QQ.relative_discriminant()
1
```

**residue_field\((p, check=True)\)**

Return the residue field of \( \mathbb{Q} \) at the prime \( p \), for consistency with other number fields.

**INPUT:**

- \( p \) – a prime integer.
- \( check \) (default True) – if True, check the primality of \( p \), else do not.

**OUTPUT:** The residue field at this prime.

EXAMPLES:

```
sage: QQ.residue_field(5)
Residue field of Integers modulo 5
sage: QQ.residue_field(next_prime(10^9))
Residue field of Integers modulo 1000000007
```

**selmer_generators\((S, m, proof=True, orders=False)\)**

Return generators of the group \( \mathbb{Q}(S, m) \).

**INPUT:**

- \( S \) – a set of primes
- \( m \) – a positive integer
- \( proof \) – ignored
- \( orders \) (default False) – if True, output two lists, the generators and their orders

**OUTPUT:**

A list of generators of \( \mathbb{Q}(S, m) \) (and, optionally, their orders in \( \mathbb{Q}^\times/(\mathbb{Q}^\times)^m \)). This is the subgroup of \( \mathbb{Q}^\times/(\mathbb{Q}^\times)^m \) consisting of elements \( a \) such that the valuation of \( a \) is divisible by \( m \) at all primes not in \( S \). It is equal to the group of \( S \)-units modulo \( m \)-th powers. The group \( \mathbb{Q}(S, m) \) 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:

`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:

```
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 github 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]
```

`selmer_space(S, p, proof=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, from_QSp, to_QSp where
• QSp is an abstract vector space over $F_p$ isomorphic to $\mathbb{Q}(S, p)$;
• QSp_gens is a list of elements of $\mathbb{Q}^*$ generating $\mathbb{Q}(S, p)$;
• 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 $\mathbb{Q}^*/(\mathbb{Q}^*)^p$;
• 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: QS2, QS2gens, fromQS2, toQS2 = QQ.selmer_space([], 2)  # needs sage.rings.number_field
sage: QS2  # needs sage.rings.number_field
Vector space of dimension 1 over Finite Field of size 2
sage: QS2gens  # 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
```

In general there is one generator for each $p \in S$, and an additional generator of $-1$ when $p = 2$:

```
sage: # needs sage.modules sage.rings.number_field
sage: QS2, QS2gens, fromQS2, toQS2 = QQ.selmer_space([5,7], 2)
sage: QS2  # needs sage.rings.number_field
Vector space of dimension 3 over Finite Field of size 2
sage: QS2gens
[5, 7, -1]
sage: toQS2(-7)
(0, 1, 1)
sage: fromQS2((0,1,1))
-7
```

The map fromQS2 is only well-defined modulo $p$’th powers (in this case, modulo squares):

```
sage: toQS2(-5/7)  # needs sage.modules sage.rings.number_field
(1, 1, 1)
sage: fromQS2((1,1,1))
```

(continues on next page)
The map toQS2 is not defined on all of $\mathbb{Q}^*$, only on those numbers which are squares away from 5 and 7:

```
sage: toQS2(210)  
Traceback (most recent call last):  
...  
ValueError: argument 210 should have valuations divisible by 2  
at all primes in [5, 7]
```

**signature()**

Return the signature of the rational field, which is $(1, 0)$, since there are 1 real and no complex embeddings.

**EXAMPLES:**

```
sage: QQ.signature()  
(1, 0)
```

**some_elements()**

Return some elements of $\mathbb{Q}$.

See `TestSuite()` for a typical use case.

**OUTPUT:** An iterator over 100 elements of $\mathbb{Q}$.

**EXAMPLES:**

```
sage: tuple(QQ.some_elements())  
(1/2, -1/2, 2, -2, 0, 1, -1, 42, 2/3, -2/3, 3/2, -3/2, 4/5, -4/5, 5/4, -5/4, 6/7, -6/7, 7/6, -7/6, 8/9, -8/9, 9/8, -9/8, 10/11, -10/11, 11/10, -11/10, 12/13, -12/13, 13/12, -13/12, 14/15, -14/15, 15/14, -15/14, 16/17, -16/17, 17/16, -17/16, 18/19, -18/19, 19/18, -19/18, 20/21, -20/21, 21/20, -21/20, 22/23, -22/23, 23/22, -23/22, 24/25, -24/25, 25/24, -25/24, ...)```

**valuation($p$)**

Return the discrete valuation with uniformizer $p$.

**EXAMPLES:**

```
sage: v = QQ.valuation(3)  
```

(continues on next page)
3-adic valuation
sage: v(1/3)
˓→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:

sage: QQ.zeta()
-1
sage: QQ.zeta(2)
-1
sage: QQ.zeta(1)
1
sage: QQ.zeta(3)
Traceback (most recent call last):
  ...
ValueError: no n-th root of unity in rational field

sage.rings.rational_field.frac(n, d)
Return the fraction n/d.

EXAMPLES:

sage: from sage.rings.rational_field import frac
sage: frac(1,2)
1/2

sage.rings.rational_field.is_RationalField(x)
Check to see if x is the rational field.

EXAMPLES:

sage: from sage.rings.rational_field import is_RationalField as is_RF
sage: is_RF(QQ)
True
sage: is_RF(ZZ)
False
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.

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:

```python
sage: sage.rings.rational.Q_to_Z(QQ, ZZ).section()
Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

class sage.rings.rational.Rational

Bases: FieldElement

A rational number.

Rational numbers are implemented using the GMP C library.

EXAMPLES:

```python
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):
...
```plaintext
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")
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

Conversion from fractions:
```sage:``` import fractions
```sage:``` f = fractions.Fraction(1r, 2r)
```sage:``` Rational(f)
1/2

Conversion from PARI:
```sage:``` Rational(pari('-939082/3992923'))
# → needs sage.libs.pari
-939082/3992923
```sage:``` Rational(pari('Pol([-1/2]]'))
#9595
# → needs sage.libs.pari
-1/2

Conversions from numpy:
```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

2.2. Rational Numbers

173
Conversions from gmpy2:

```
sage: from gmpy2 import *
sage: QQ(mpq('3/4'))
3/4
sage: QQ(mpz(42))
42
sage: Rational(mpq(2/3))
2/3
sage: Rational(mpz(5))
5
```

**absolute_norm()**

Return the norm from \( \mathbb{Q} \) 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
```

**additive_order()**

Return the additive order of \( \text{self} \).

OUTPUT: integer or infinity

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

**as_integer_ratio()**

Return the pair \((\text{self.numerator()}, \text{self.denominator()})\).

```
sage: x = -12/29
sage: x.as_integer_ratio()
(-12, 29)
```

**ceil()**

Return the ceiling of this rational number.

OUTPUT: Integer

If this rational number is an integer, this returns this number, otherwise it returns the floor of this number +1.

```
sage: n = 5/3; n.ceil()
2
sage: n = -17/19; n.ceil()
```

(continues on next page)
0

sage: n = -7/2; n.ceil()
-3
sage: n = 7/2; n.ceil()
4
sage: n = 10/2; n.ceil()
5

\texttt{charpoly}(\texttt{var}=’x’)

Return the characteristic polynomial of this rational number. This will always be just \texttt{var} - \texttt{self}; this is really here so that code written for number fields won’t crash when applied to rational numbers.

INPUT:

\begin{itemize}
\item \texttt{var} - a string
\end{itemize}

OUTPUT: Polynomial

EXAMPLES:

\begin{verbatim}
sage: (1/3).charpoly(’x’)
x - 1/3
\end{verbatim}

The default is \texttt{var}=’x’. (github issue \#20967):

\begin{verbatim}
sage: a = QQ(2); a.charpoly(’x’)
x - 2
\end{verbatim}

AUTHORS:

\begin{itemize}
\item Craig Citro
\end{itemize}

\texttt{conjugate}()

Return the complex conjugate of this rational number, which is the number itself.

EXAMPLES:

\begin{verbatim}
sage: n = 23/11
sage: n.conjugate()
23/11
\end{verbatim}

\texttt{content}(\texttt{other})

Return the content of \texttt{self} and \texttt{other}, i.e., the unique positive rational number \(c\) such that \(\texttt{self}/c\) and \(\texttt{other}/c\) are coprime integers.

\texttt{other} can be a rational number or a list of rational numbers.

EXAMPLES:

\begin{verbatim}
sage: a = 2/3
sage: a.content(2/3)
2/3
sage: a.content(1/5)
1/15
sage: a.content([2/5, 4/9])
2/45
\end{verbatim}
continued_fraction()

Return the continued fraction of that rational.

EXAMPLES:

```sage
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:

```sage
sage: (13/9).continued_fraction_list()
[1, 2, 4]
sage: 1 + 1/(2 + 1/4)
13/9
sage: (225/157).continued_fraction_list()
[1, 2, 3, 4, 5]
sage: 1 + 1/(2 + 1/(3 + 1/(4 + 1/5)))
225/157
sage: (fibonacci(20)/fibonacci(19)).continued_fraction_list() # needs sage.libs.pari
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
sage: (-1/3).continued_fraction_list()
[-1, 1, 2]
```

Check that the partial quotients of an integer $n$ is simply $[n]$:

```sage
sage: QQ(1).continued_fraction_list()
[1]
sage: QQ(0).continued_fraction_list()
[0]
sage: QQ(-1).continued_fraction_list()
[-1]
```

Hirzebruch-Jung continued fractions:
```sage
sage: (11/19).continued_fraction_list("hj")
[1, 3, 2, 3, 2]
sage: 1 - 1/(3 - 1/(2 - 1/(3 - 1/2)))
11/19
sage: (225/137).continued_fraction_list("hj")
[2, 3, 5, 10]
sage: 2 - 1/(3 - 1/(5 - 1/10))
225/137
sage: (-23/19).continued_fraction_list("hj")
[-1, 5, 4]
sage: -1 - 1/(5 - 1/4)
-23/19
```

### `denom()`

Return the denominator of this rational number. `denom()` is an alias of `denominator()`.

**EXAMPLES:**

```sage
sage: x = -5/11
sage: x.denominator()
11
sage: x = 9/3
sage: x.denominator()
1
sage: x = 5/13
sage: x.denom()
13
```

### `denominator()`

Return the denominator of this rational number. `denom()` is an alias of `denominator()`.

**EXAMPLES:**

```sage
sage: x = -5/11
sage: x.denominator()
11
sage: x = 9/3
sage: x.denominator()
1
sage: x = 5/13
sage: x.denom()
13
```

### `factor()`

Return the factorization of this rational number.

**OUTPUT:** Factorization

**EXAMPLES:**
Trying to factor 0 gives an arithmetic error:

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

**floor()**

Return the floor of this rational number as an integer.

**OUTPUT:** Integer

**EXAMPLES:**

```
sage: n = 5/3; n.floor()
1
sage: n = -17/19; n.floor()
-1
sage: n = -7/2; n.floor()
-4
sage: n = 7/2; n.floor()
3
sage: n = 10/2; n.floor()
5
```

**gamma**(prec=None)

Return the gamma function evaluated at self. This value is exact for integers and half-integers, and returns a symbolic value otherwise. For a numerical approximation, use keyword prec.

**EXAMPLES:**

```
sage: # 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)
```

This function accepts an optional precision argument:

```
sage: (1/3).gamma(prec=100)  # needs sage.rings.real_mpfr
2.6789385347077460272981674833
sage: (1/2).gamma(prec=100)  # needs sage.rings.real_mpfr
1.7724538509055160272981674833
```
global_height\((prec=None)\)
Return the absolute logarithmic height of this rational number.

**INPUT:**

- \(\text{prec} \text{ (int)}\) – desired floating point precision (default: default RealField precision).

**OUTPUT:**

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

**ALGORITHM:**
The height is the sum of the total archimedean and non-archimedean components, which is equal to \(\max(\log(n), \log(d))\) where \(n, d\) are the numerator and denominator of the rational number.

**EXAMPLES:**

```sage
sage: # needs sage.rings.real_mpfr
sage: a = QQ(6/25)
sage: a.global_height_arch() + a.global_height_non_arch()
3.21887582486820
sage: a.global_height()
3.21887582486820
sage: (1/a).global_height_arch()
1.42711635564015
sage: QQ(0).global_height()
0.000000000000000
sage: QQ(1).global_height()
0.000000000000000
```

global_height_arch\((prec=None)\)
Return the total archimedean component of the height of this rational number.

**INPUT:**

- \(\text{prec} \text{ (int)}\) – 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:**

```sage
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: QQ(0).global_height_arch()  # needs sage.rings.real_mpfr
0.000000000000000
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:**

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

**OUTPUT:**

(real) The total non-archimedean component of the height of this rational number.

**ALGORITHM:**

This is the sum of the local heights at all primes `p`, which may be computed without factorization as the log of the denominator.

**EXAMPLES:**

```python
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, 1.09861228866811, 0.000000000000000]
sage: 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 `self`, as an `Integer`.

**OUTPUT:** `Integer`

**EXAMPLES:**

```python
sage: a = 2/3
sage: a.height()
3
sage: a = 34/3
sage: a.height()
34
sage: a = -97/4
sage: a.height()
97
```

**AUTHORS:**

- Naqi Jaffery (2006-03-05): examples

**Note:** For the logarithmic height, use `global_height()`.

**imag**

Return the imaginary part of `self`, which is zero.

**EXAMPLES:**
\begin{verbatim}
sage: (1/239).imag()
0
\end{verbatim}

\textbf{is\_S\_integral}(\texttt{S}=[])

Determine if the rational number is \(S\)-integral.

\(x\) is \(S\)-integral if \(x.\text{valuation}(p)\geq 0\) for all \(p\) not in \(S\), i.e., the denominator of \(x\) is divisible only by the primes in \(S\).

\textbf{INPUT:}
- \(S\) – list or tuple of primes.

\textbf{OUTPUT:} bool

\textbf{Note:} Primality of the entries in \(S\) is not checked.

\textbf{EXAMPLES:}
\begin{verbatim}
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]
\end{verbatim}

\textbf{is\_S\_unit}(\texttt{S}=\texttt{None})

Determine if the rational number is an \(S\)-unit.

\(x\) is an \(S\)-unit if \(x.\text{valuation}(p)==0\) for all \(p\) not in \(S\), i.e., the numerator and denominator of \(x\) are divisible only by the primes in \(S\).

\textbf{INPUT:}
- \(S\) – list or tuple of primes.

\textbf{OUTPUT:} bool

\textbf{Note:} Primality of the entries in \(S\) is not checked.

\textbf{EXAMPLES:}
\begin{verbatim}
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]
\end{verbatim}

\textbf{is\_integer()}

Determine if a rational number is integral (i.e., is in \(\mathbb{Z}\)).

\textbf{OUTPUT:} bool

\textbf{EXAMPLES:}
sage: QQ(1/2).is_integral()
False
sage: QQ(4/4).is_integral()
True

**is_integral()**

Determine if a rational number is integral (i.e., is in \( \mathbb{Z} \)).

OUTPUT: bool

EXAMPLES:

```python
sage: QQ(1/2).is_integral()
False
sage: QQ(4/4).is_integral()
True
```

**is_norm(\( L, \) element=False, proof=True)**

Determine whether \( \text{self} \) is the norm of an element of \( L \).

INPUT:

- \( L \) – a number field
- \( \text{element} \) – (default: False) boolean whether to also output an element of which \( \text{self} \) is a norm
- \( \text{proof} \) – 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 \( \text{self} \) is the norm of an element of \( L \). If \( \text{element} \) 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 \( \text{self} == x.\text{norm()} \). Otherwise, \( x \) is None.

ALGORITHM:

Uses the PARI function pari:bnfisnorm. See _bnfisnorm().

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)
```

The number field doesn’t have to be defined by an integral polynomial:
... (continues on next page)
is_one()
Determine if a rational number is one.

OUTPUT: bool

EXAMPLES:

```
sage: QQ(1/2).is_one()
False
sage: QQ(4/4).is_one()
True
```

is_padic_square(p, check=True)
Determines whether this rational number is a square in $\mathbb{Q}_p$ (or in $R$ when $p = \text{infinity}$).

INPUT:
- $p$ - a prime number, or infinity
- check – (default: True); check if $p$ is prime

EXAMPLES:

```
sage: QQ(2).is_padic_square(7)
True
sage: QQ(98).is_padic_square(7)
True
sage: QQ(2).is_padic_square(5)
False
```

is_perfect_power(expected_value=False)
Return True if self is a perfect power.

INPUT:
- expected_value - (bool) whether or not this rational is expected be a perfect power. This does not affect the correctness of the output, only the runtime.

If expected_value is False (default) it will check the smallest of the numerator and denominator is a perfect power as a first step, which is often faster than checking if the quotient is a perfect power.

EXAMPLES:

```
sage: (4/9).is_perfect_power()
True
sage: (144/1).is_perfect_power()
True
sage: (4/3).is_perfect_power()
False
sage: (2/27).is_perfect_power()
False
```
sage: (4/27).is_perfect_power()
False
sage: (-1/25).is_perfect_power()
False
sage: (-1/27).is_perfect_power()
True
sage: (0/1).is_perfect_power()
True

The second parameter does not change the result, but may change the runtime.

sage: (-1/27).is_perfect_power(True)
True
sage: (-1/25).is_perfect_power(True)
False
sage: (2/27).is_perfect_power(True)
False
sage: (144/1).is_perfect_power(True)
True

This test makes sure we workaround a bug in GMP (see github issue #4612):

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

**is_rational()**

Return True since this is a rational number.

EXAMPLES:

sage: (3/4).is_rational()
True

**is_square()**

Return whether or not this rational number is a square.

OUTPUT: bool

EXAMPLES:

sage: x = 9/4
sage: x.is_square()
True
sage: x = (7/53)^100
sage: x.is_square()
True
sage: x = 4/3
sage: x.is_square()
False
sage: x = -1/4
sage: x.is_square()
False
**list()**

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

**OUTPUT:**

- list - the list [self]

**EXAMPLES:**

```python
sage: m = 5/3
sage: m.list()
[5/3]
```

**local_height(p, prec=None)**

Return the local height of this rational number at the prime \( p \).

**INPUT:**

- \( p \) – a prime number
- \( \text{prec} \) (int) – desired floating point precision (default: default \textit{RealField} precision).

**OUTPUT:**

(real) The local height of this rational number at the prime \( p \).

**EXAMPLES:**

```python
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
```

**local_height_arch(prec=None)**

Return the Archimedean local height of this rational number at the infinite place.

**INPUT:**

- \( \text{prec} \) (int) – desired floating point precision (default: default \textit{RealField} precision).

**OUTPUT:**

(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
0.000000000000000
sage: (1/a).local_height_arch()  # needs sage.rings.real_mpfr
1.42711635564015
sage: (1/a).local_height_arch(100)  # needs sage.rings.real_mpfr
```

(continues on next page)
\texttt{log}(m=None, prec=None)

Return the log of self.

**INPUT:**

- \texttt{m} – the base (default: natural log base e)
- \texttt{prec} – integer (optional); the precision in bits

**OUTPUT:**

When \texttt{prec} is not given, the log as an element in symbolic ring unless the logarithm is exact. Otherwise the log is a \texttt{RealField} approximation to \texttt{prec} bit precision.

**EXAMPLES:**

```
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)  # needs sage.symbolic
3

sage: log(QQ(125), 3)  # needs sage.symbolic
3*log(5)/log(3)

sage: QQ(8).log(1/2)  # needs sage.libs.pari
-3

sage: (1/8).log(1/2)  # needs sage.libs.pari
3

sage: log(QQ(8), 1/2)  # needs sage.libs.pari
3

sage: QQ(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: log(16/81, 27/8)  # needs sage.libs.pari
-4/3

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
```

**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: (1/3).minpoly()
x - 1/3
```

```
sage: (1/3).minpoly('y')
y - 1/3
```

**AUTHORS:**

- Craig Citro

**mod_ui**(n)

Return the remainder upon division of self by the unsigned long integer n.

**INPUT:**

- n - an unsigned long integer

**OUTPUT:** integer

**EXAMPLES:**

```
sage: (-4/17).mod_ui(3)
1
```

```
sage: (-4/17).mod_ui(17)
Traceback (most recent call last):
...
ArithmeticError: The inverse of 0 modulo 17 is not defined.
```

**multiplicative_order**()

Return the multiplicative order of self.

**OUTPUT:** Integer or infinity

**EXAMPLES:**

```
sage: QQ(1).multiplicative_order()
1
```

```
sage: QQ('1/-1').multiplicative_order()
2
```

```
sage: QQ(0).multiplicative_order()
+Infinity
```

```
sage: QQ('2/3').multiplicative_order()
```

(continues on next page)
norm()

Return the norm from \( \mathbb{Q} \) to \( \mathbb{Q} \) of \( x \) (which is just \( x \)). This was added for compatibility with \texttt{NumberField}.

**OUTPUT:**

- Rational - reference to self

**EXAMPLES:**

```sage
sage: (1/3).norm()
1/3
```

**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:**

```sage
sage: (25/4).nth_root(2)
5/2
sage: (125/8).nth_root(3)
5/2
sage: (-125/8).nth_root(3)
-5/2
sage: (25/4).nth_root(-2)
2/5
sage: (9/2).nth_root(2)
Traceback (most recent call last):
... ValueError: not a perfect 2nd power
sage: (-25/4).nth_root(2)
Traceback (most recent call last):
... ValueError: cannot take even root of negative number
```

\texttt{numer()}

Return the numerator of this rational number. \texttt{numer()} is an alias of \texttt{numerator()}.

**EXAMPLES:**

2.2. Rational Numbers
Sage: \( x = \frac{5}{11} \)
Sage: \( x.\text{numerator()} \)
5

Sage: \( x = \frac{9}{3} \)
Sage: \( x.\text{numerator()} \)
3

Sage: \( x = -\frac{5}{11} \)
Sage: \( x.\text{numerator()} \)
-5

\text{numerator()} \)

Return the numerator of this rational number. \( \text{numerator()} \) is an alias of \( \text{numerator()} \).

\text{EXAMPLES:}

Sage: \( x = \frac{5}{11} \)
Sage: \( x.\text{numerator()} \)
5

Sage: \( x = \frac{9}{3} \)
Sage: \( x.\text{numerator()} \)
3

Sage: \( x = -\frac{5}{11} \)
Sage: \( x.\text{numerator()} \)
-5

\text{ord}(p) \)

Return the power of \( p \) in the factorization of self.

\text{INPUT:}

• \( p \) - a prime number

\text{OUTPUT:}

(integer or infinity) \( \text{Infinity} \) if \( \text{self} \) is zero, otherwise the (positive or negative) integer \( e \) such that \( \text{self} = m * p^e \) with \( m \) coprime to \( p \).

\text{Note:} \quad \text{See also } \text{val_unit()} \text{ which returns the pair } (e, m). \quad \text{The function } \text{ord()} \text{ is an alias for } \text{valuation()}. 

\text{EXAMPLES:}

Sage: \( x = -\frac{5}{9} \)
Sage: \( x.\text{valuation}(5) \)
1
Sage: \( x.\text{ord}(5) \)
1
Sage: \( x.\text{valuation}(3) \)
-2

(continues on next page)
sage: x.valuation(2)
0

Some edge cases:

sage: (0/1).valuation(4)
+Infinity
sage: (7/16).valuation(4)
-2

period()

Return the period of the repeating part of the decimal expansion of this rational number.

ALGORITHM:

When a rational number $n/d$ with $(n, d) = 1$ is expanded, the period begins after $s$ terms and has length $t$, where $s$ and $t$ are the smallest numbers satisfying $10^s = 10^{s+t} \mod d$. In general if $d = 2^a5^b m$ 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

prime_to_S_part($S=\emptyset$)

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:**

```python
sage: QQ(3/4).prime_to_S_part()
3/4
sage: QQ(3/4).prime_to_S_part([2])
3
sage: QQ(-3/4).prime_to_S_part([3])
-1/4
sage: QQ(700/99).prime_to_S_part([2,3,5])
7/11
sage: QQ(-700/99).prime_to_S_part([2,3,5])
-7/11
sage: QQ(0).prime_to_S_part([2,3,5])
0
sage: QQ(-700/99).prime_to_S_part([])
-700/99
```

**real()**

Return the real part of `self`, which is `self`.

**EXAMPLES:**

```python
sage: (1/2).real()
1/2
```

**relative_norm()**

Return the norm from Q to Q of x (which is just x). This was added for compatibility with NumberFields.

**EXAMPLES:**

```python
sage: (6/5).relative_norm()
6/5
sage: QQ(7/5).relative_norm()
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
```

`sign()`
Return the sign of this rational number, which is -1, 0, or 1 depending on whether this number is negative, zero, or positive respectively.

OUTPUT: Integer

EXAMPLES:

```
sage: (2/3).sign()
1
sage: (0/3).sign()
0
sage: (-1/6).sign()
-1
```

`sqrt(prec=None, extend=True, all=False)`
The square root function.

INPUT:

• `prec` – integer (default: None): if None, returns an exact square root; otherwise returns a numerical square root if necessary, to the given bits of precision.

• `extend` – bool (default: True): if True, return a square root in an extension ring, if necessary. Otherwise, raise a ValueError if the square is not in the base ring.

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

EXAMPLES:
sage: x = 25/9
sage: x.sqrt()
5/3

sage: x = 64/4
sage: x.sqrt()
4

sage: x = 100/1
sage: x.sqrt(all=True)
[10, -10]

sage: x = 81/5
sage: x.sqrt()     # needs sage.symbolic
9*sqrt(1/5)

sage: x = -81/3
sage: 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=10)
0.82

sage: n.sqrt(prec=100)
0.81649658092772603273488489479024863305304008

sage: n.sqrt(prec=100)^2
0.66666666666666666666666666667

sage: n.sqrt(extend=False, all=True)
Traceback (most recent call last):
...
ValueError: square root of 2/3 not a rational number

sage: sqrt(-2/3, all=True)     # needs sage.symbolic
[0.816496580927726*0.816496580927726*I, -0.816496580927726*0.816496580927726*I]

sage: n.sqrt(extend=False, all=True)
sqrt(-2/3), -sqrt(-2/3)]

AUTHORS:
• Naqi Jaffery (2006-03-05): some examples

squarefree_part()
Return the square free part of $x$, i.e., an integer $z$ such that $x = zy^2$, for a perfect square $y^2$.

**EXAMPLES:**

```
sage: a = 1/2
code: sage: a.squarefree_part()
does not display any output.
sage: b = a/a.squarefree_part()  
code: sage: b, b.is_square()  
print: (1/4, True)
sage: a = 24/5

code: sage: a.squarefree_part()
does not display any output.
```

**str(base=10)**

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

**INPUT:**

- `base` – integer (default: 10); base must be between 2 and 36.

**OUTPUT:** string

**EXAMPLES:**

```
sage: (-4/17).str()
sage: '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()
print: [2, 17]
```

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

```
sage: (0/1).support()
Traceback (most recent call last):
...  
ArithmeticError: Support of 0 not defined.
```
trace()
Return the trace from $\mathbb{Q}$ to $\mathbb{Q}$ of $x$ (which is just $x$). This was added for compatibility with NumberFields.

OUTPUT:
• Rational - reference to self

EXAMPLES:

```
sage: (1/3).trace()
1/3
```

AUTHORS:
• Craig Citro

trunc()
Round this rational number to the nearest integer toward zero.

EXAMPLES:

```
sage: (5/3).trunc()
1
sage: (-5/3).trunc()
-1
sage: QQ(42).trunc()
42
sage: QQ(-42).trunc()
-42
```

val_unit($p$)
Return a pair: the $p$-adic valuation of self, and the $p$-adic unit of self, as a Rational.

We do not require the $p$ be prime, but it must be at least 2. For more documentation see Integer. val_unit().

INPUT:
• $p$ - a prime

OUTPUT:
• int - the $p$-adic valuation of this rational
• Rational - $p$-adic unit part of self

EXAMPLES:

```
sage: (-4/17).val_unit(2)
(2, -1/17)
sage: (-4/17).val_unit(17)
(-1, -4)
sage: (0/1).val_unit(17)
(+Infinity, 1)
```

AUTHORS:
• David Roe (2007-04-12)
valuation($p$)

Return the power of $p$ in the factorization of self.

**INPUT:**
- $p$ - a prime number

**OUTPUT:**
(integer or infinity) Infinity if self is zero, otherwise the (positive or negative) integer $e$ such that self = $m \times p^e$ 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
```

Some edge cases:

```sage
sage: (0/1).valuation(4)
+Infinity
sage: (7/16).valuation(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:**

```sage
sage: QQ.coerce_map_from(ZZ).is_surjective()
False
```

**section()**

Return a section of this morphism.

**EXAMPLES:**

```sage
sage: f = QQ.coerce_map_from(ZZ).section(); f
Generic map:
  From: Rational Field
  To:   Integer Ring
```
This map is a morphism in the category of sets with partial maps (see github issue #15618):

```
sage: f.parent()
Set of Morphisms from Rational Field to Integer Ring
in Category of sets with partial maps
```

class sage.rings.rational.int_to_Q

Bases: 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:

```
sage: from sage.rings.rational import integer_rational_power
sage: integer_rational_power(49, 1/2)
7
sage: integer_rational_power(27, 1/3)
3
sage: integer_rational_power(-27, 1/3) is None
True
sage: integer_rational_power(-27, 2/3) is None
True
sage: integer_rational_power(512, 7/9)
128
sage: integer_rational_power(27, 1/4) is None
True
sage: integer_rational_power(-16, 1/4) is None
True
sage: integer_rational_power(0, 7/9)
0
sage: integer_rational_power(1, 7/9)
1
sage: integer_rational_power(-1, 7/9) is None
True
sage: integer_rational_power(-1, 8/9) is None
True
sage: integer_rational_power(-1, 9/8) is None
True
```

TESTS (github issue #11228):
sage: integer_rational_power(-10, QQ(2))
100
sage: integer_rational_power(0, QQ(0))
1

sage.rings.rational.is_Rational(x)

Return True if x is of the Sage Rational type.

EXAMPLES:

sage: from sage.rings.rational import is_Rational
sage: is_Rational(2)
False
sage: is_Rational(2/1)
True
sage: is_Rational(int(2))
False
sage: is_Rational('5')
False

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)
sage: rational_power_parts(9/1000, 1/2)
(3/10, 1/10)
INDICES AND TABLES

- Index
- Module Index
- Search Page
a
sage.arith.functions, 77
sage.arith.misc, 82
sage.arith.multi_modular, 79
sage.arith.power, 78

r
sage.rings.bernmm, 65
sage.rings.bernoulli_mod_p, 66
sage.rings.factorint, 69
sage.rings.factorint_flint, 71
sage.rings.factorint_pari, 72
sage.rings.fast_arith, 73
sage.rings.integer, 13
sage.rings.integer_ring, 1
sage.rings.rational, 172
sage.rings.rational_field, 157
sage.rings.sum_of_squares, 74
Symbols
  `__pow__()` (sage.rings.integer.Integer method), 16

A
  absolute_degree() (sage.rings.integer_ring.IntegerRing_class method), 4
  absolute_degree() (sage.rings.rational_field.RationalField method), 159
  absolute_discriminant()
    (sage.rings.rational_field.RationalField method), 159
  absolute_norm()
    (sage.rings.rational.Rational method), 174
  absolute_polynomial()
    (sage.rings.rational_field.RationalField method), 159
  additive_order() (sage.rings.integer.Integer method), 17
  additive_order() (sage.rings.rational.Rational method), 175
  algdep()
    (in module sage.arith.misc), 93
  algebraic_closure()
    (sage.rings.rational_field.RationalField method), 159
  algebraic_dependency()
    (in module sage.arith.misc), 95
  arith_int
    (class in sage.rings.fast_arith), 73
  arith_llong
    (class in sage.rings.fast_arith), 73
  as_integer_ratio()
    (sage.rings.integer.Integer method), 17
  as_integer_ratio()
    (sage.rings.rational.Rational method), 174
  aurifeuillian()
    (in module sage.rings.factorint), 69
  automorphisms() (sage.rings.rational_field.RationalField method), 159

B
  balanced_digits()
    (sage.rings.integer.Integer method), 18
  bernmm_bern_modp()
    (in module sage.rings.bernmm), 65
  bernmm_bern_rat()
    (in module sage.rings.bernmm), 65
  bernoulli()
    (in module sage.arith.misc), 97
  bernoulli_modp()
    (in module sage.rings.bernoulli_modp), 66
  bernoulli_modp_single()
    (in module sage.rings.bernoulli_modp), 67
  binary() (sage.rings.integer.Integer method), 18
  binomial()
    (in module sage.arith.misc), 98
  binomial() (sage.rings.integer.Integer method), 19
  binomial_coefficients() (in module sage.arith.misc), 99
  bit_length() (sage.rings.integer.Integer method), 19
  bits()
    (sage.rings.integer.Integer method), 20

C
  carmichael_lambda()
    (in module sage.arith.misc), 100
  ceil()
    (sage.rings.integer.Integer method), 20
  ceil()
    (sage.rings.rational.Rational method), 174
  characteristic()
    (sage.rings.integer_ring.IntegerRing_class method), 5
  characteristic()
    (sage.rings.rational_field.RationalField method), 160
  charpoly()
    (sage.rings.rational.Rational method), 175
  class_number()
    (sage.rings.integer.Integer method), 20
  class_number()
    (sage.rings.rational_field.RationalField method), 160
  completion()
    (sage.rings.integer_ring.IntegerRing_class method), 5
  completion()
    (sage.rings.rational_field.RationalField method), 160
  complex_embedding()
    (sage.rings.rational_field.RationalField method), 160
  conjugate()
    (sage.rings.integer.Integer method), 21
  conjugate()
    (sage.rings.rational.Rational method), 175
  construction()
    (sage.rings.rational_field.RationalField method), 160
  content()
    (sage.rings.rational.Rational method), 175
  continuant()
    (in module sage.arith.misc), 102
  continued_fraction()
    (sage.rings.rational.Rational method), 175
  continued_fraction_list()
    (sage.rings.rational.Rational method), 176
### Index

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_height()</td>
<td>sage.rings.rational.Rational</td>
<td>178</td>
</tr>
<tr>
<td>global_height_arch()</td>
<td>sage.rings.rational.Rational</td>
<td>179</td>
</tr>
<tr>
<td>global_height_non_arch()</td>
<td>sage.rings.rational.Rational</td>
<td>179</td>
</tr>
<tr>
<td>height()</td>
<td>sage.rings.rational.Rational</td>
<td>180</td>
</tr>
<tr>
<td>hex()</td>
<td>sage.rings.integer.Integers</td>
<td>31</td>
</tr>
<tr>
<td>hilbert_conductor()</td>
<td>module sage.arith.misc</td>
<td>118</td>
</tr>
<tr>
<td>hilbert_conductor_inverse()</td>
<td>module sage.arith.misc</td>
<td>118</td>
</tr>
<tr>
<td>hilbert_symbol()</td>
<td>module sage.arith.misc</td>
<td>119</td>
</tr>
<tr>
<td>hilbert_symbol_negative_at_S()</td>
<td>module sage.arith.misc</td>
<td>162</td>
</tr>
<tr>
<td>imag()</td>
<td>sage.rings.integer.Integers</td>
<td>31</td>
</tr>
<tr>
<td>int_to_Z</td>
<td>class in sage.rings.rational</td>
<td>198</td>
</tr>
<tr>
<td>Integer</td>
<td>class in sage.rings.integer</td>
<td>15</td>
</tr>
<tr>
<td>integer_floor()</td>
<td>module sage.arith.misc</td>
<td>120</td>
</tr>
<tr>
<td>integer_rational_power()</td>
<td>module sage.rings.rational</td>
<td>198</td>
</tr>
<tr>
<td>integer_trunc()</td>
<td>module sage.arith.misc</td>
<td>121</td>
</tr>
<tr>
<td>IntegerRing()</td>
<td>module sage.rings.integer_ring</td>
<td>1</td>
</tr>
<tr>
<td>IntegerWrapper</td>
<td>class in sage.rings.integer</td>
<td>63</td>
</tr>
<tr>
<td>inverse_mod()</td>
<td>module sage.arith.misc</td>
<td>121</td>
</tr>
<tr>
<td>inverse_mod_int()</td>
<td>sage.rings.integer.Integers</td>
<td>73</td>
</tr>
<tr>
<td>inverse_mod_longlong()</td>
<td>sage.rings.fast_arith.arith_int</td>
<td>73</td>
</tr>
<tr>
<td>is_absolute()</td>
<td>sage.rings.rational_field.RationalField</td>
<td>163</td>
</tr>
<tr>
<td>is_discriminant()</td>
<td>sage.rings.integer.Integers</td>
<td>33</td>
</tr>
<tr>
<td>is_field()</td>
<td>sage.rings.integer_ring.Integers</td>
<td>6</td>
</tr>
<tr>
<td>is_fundamental_discriminant()</td>
<td>sage.rings.integer.Integers</td>
<td>33</td>
</tr>
<tr>
<td>is_Integer()</td>
<td>module sage.rings.integer</td>
<td>64</td>
</tr>
<tr>
<td>is_integer()</td>
<td>sage.rings.integer.Integers</td>
<td>33</td>
</tr>
<tr>
<td>is_integer()</td>
<td>sage.rings.rational.Rational</td>
<td>181</td>
</tr>
<tr>
<td>is_integral()</td>
<td>sage.rings.integer.Integers</td>
<td>13</td>
</tr>
<tr>
<td>is_integral()</td>
<td>sage.rings.rational.Rational</td>
<td>33</td>
</tr>
<tr>
<td>is_integral()</td>
<td>sage.rings.rational.Rational</td>
<td>182</td>
</tr>
<tr>
<td>is_integralell()</td>
<td>sage.rings.integer.Integers</td>
<td>6</td>
</tr>
<tr>
<td>is_irreducible()</td>
<td>sage.rings.integer.Integers</td>
<td>34</td>
</tr>
<tr>
<td>is_noetherian()</td>
<td>sage.rings.integer.Integers(RationalRing)</td>
<td>7</td>
</tr>
<tr>
<td>is_norm()</td>
<td>sage.rings.integer.Integers</td>
<td>34</td>
</tr>
<tr>
<td>is_nth_power()</td>
<td>sage.rings.rational.Rational</td>
<td>182</td>
</tr>
<tr>
<td>is_one()</td>
<td>sage.rings.integer.Integers</td>
<td>34</td>
</tr>
<tr>
<td>is_odd()</td>
<td>sage.rings.rational.Rational</td>
<td>184</td>
</tr>
<tr>
<td>is_perfect_power()</td>
<td>sage.rings.integer.Integers</td>
<td>35</td>
</tr>
<tr>
<td>is_perfect_power()</td>
<td>sage.rings.rational.Rational</td>
<td>184</td>
</tr>
<tr>
<td>is_power_of()</td>
<td>sage.rings.integer.Integers</td>
<td>35</td>
</tr>
<tr>
<td>is_power_of_two()</td>
<td>module sage.arith.misc</td>
<td>122</td>
</tr>
<tr>
<td>is_prime()</td>
<td>module sage.arith.misc</td>
<td>122</td>
</tr>
<tr>
<td>is_prime()</td>
<td>sage.rings.integer.Integers</td>
<td>36</td>
</tr>
<tr>
<td>is_prime_field()</td>
<td>sage.rings.rational_field.RationalField</td>
<td>163</td>
</tr>
<tr>
<td>is_prime_power()</td>
<td>module sage.arith/misc</td>
<td>123</td>
</tr>
<tr>
<td>is_prime_power()</td>
<td>sage.rings.integer.Integers</td>
<td>37</td>
</tr>
<tr>
<td>is_pseudoprime()</td>
<td>module sage.arith/misc</td>
<td>124</td>
</tr>
<tr>
<td>is_pseudoprime()</td>
<td>sage.rings.integer.Integers</td>
<td>39</td>
</tr>
<tr>
<td>is_pseudoprime_power()</td>
<td>module sage.arith/misc</td>
<td>124</td>
</tr>
<tr>
<td>is_pseudoprime_power()</td>
<td>sage.rings.integer.Integers</td>
<td>39</td>
</tr>
<tr>
<td>is_Rational()</td>
<td>module sage.rings.rational</td>
<td>199</td>
</tr>
<tr>
<td>is_Rational()</td>
<td>sage.rings.integer.Integers</td>
<td>40</td>
</tr>
<tr>
<td>is_RationalField()</td>
<td>module sage.rings.rational_field</td>
<td>181</td>
</tr>
<tr>
<td>is_S_integral()</td>
<td>sage.rings.rational.Rational</td>
<td>171</td>
</tr>
<tr>
<td>is_S_unit()</td>
<td>sage.rings.rational.Rational</td>
<td>181</td>
</tr>
<tr>
<td>is_square()</td>
<td>module sage.arith.misc</td>
<td>125</td>
</tr>
<tr>
<td>is_square()</td>
<td>sage.rings.integer.Integers</td>
<td>40</td>
</tr>
<tr>
<td>is_square()</td>
<td>sage.rings.rational.Rational</td>
<td>185</td>
</tr>
<tr>
<td>is_squarefree()</td>
<td>module sage.arith.misc</td>
<td>126</td>
</tr>
<tr>
<td>is_squarefree()</td>
<td>sage.rings.integer.Integers</td>
<td>181</td>
</tr>
</tbody>
</table>
O
oct() (sage.rings.integer.Integer method), 48
odd_part() (in module sage.arith.misc), 135
odd_part() (sage.rings.integer.Integer method), 49
ord() (sage.rings.integer.Integer method), 49
ord() (sage.rings.rational.Rational method), 190
order() (sage.rings.integer_ring.IntegerRing_class method), 7
order() (sage.rings.rational_field.RationalField method), 163
ordinal_str() (sage.rings.integer.Integer method), 49

P
p_primary_part() (sage.rings.integer.Integer method), 50
parameter() (sage.rings.integer_ring.IntegerRing_class method), 7
partial_product() (sage.arith.multi_modular.MultiModularBasis_base method), 80
perfect_power() (sage.rings.integer.Integer method), 50
period() (sage.rings.rational.Rational method), 191
places() (sage.rings.rational_field.RationalField method), 164
plot() (sage.arith.misc.Euler_Phi method), 87
plot() (sage.arith.misc.Moebius method), 90
plot() (sage.arith.misc.Sigma method), 91
polynomial() (sage.rings.rational_field.RationalField method), 164
popcount() (sage.rings.integer.Integer method), 51
power_basis() (sage.rings.rational_field.RationalField method), 164
power_mod() (in module sage.arith.misc), 136
powermod() (sage.rings.integer.Integer method), 51
precomputation_list() (sage.arith.multi_modular.MultiModularBasis_base method), 80
previous_prime() (in module sage.arith.misc), 136
previous_prime() (sage.rings.integer.Integer method), 52
previous_prime_power() (in module sage.arith.misc), 137
previous_prime_power() (sage.rings.integer.Integer method), 53
prime_divisors() (in module sage.arith.misc), 138
prime_divisors() (sage.rings.integer.Integer method), 53
prime_factors() (in module sage.arith.misc), 139
prime_factors() (sage.rings.integer.Integer method), 54
prime_powers() (in module sage.arith.misc), 139
prime_range() (in module sage.rings.fast_arith), 73
prime_to_m_part() (in module sage.arith.misc), 140

prime_to_m_part() (sage.rings.integer.Integer method), 54
prime_to_S_part() (sage.rings.rational.Rational method), 191
primes() (in module sage.arith.misc), 141
primes_first_n() (in module sage.arith.misc), 142
primes_of_bounded_norm_iter() (sage.rings.rational_field.RationalField method), 165
primitive_root() (in module sage.arith.misc), 142
prod() (sage.arith.multi_modular.MultiModularBasis_base method), 80

Python Enhancement Proposals
PEP 3127, 16

Q
Q_to_Z (class in sage.rings.rational), 172
quadratic_defect() (sage.rings.rational_field.RationalField method), 165
quadratic_residues() (in module sage.arith.misc), 143
quo_rem() (sage.rings.integer.Integer method), 55
quotient() (sage.rings.integer_ring.IntegerRing_class method), 7

R
radical() (in module sage.arith.misc), 144
random_element() (sage.rings.integer_ring.IntegerRing_class method), 8
random_element() (sage.rings.rational_field.RationalField method), 165
random_prime() (in module sage.arith.misc), 144
range() (sage.arith.misc.Moebius method), 90
range() (sage.rings.integer_ring.IntegerRing_class method), 10
range_by_height() (sage.rings.rational_field.RationalField method), 166
Rational (class in sage.rings.rational), 172
rational_power_parts() (in module sage.rings.rational), 199
rational_recon_int() (sage.rings.fast_arith.arith_int method), 73
rational_recon_longlong() (sage.rings.fast_arith.arith_llong method), 73
rational_reconstruction() (in module sage.arith.misc), 145
rational_reconstruction() (sage.rings.integer.Integer method), 55
RationalField (class in sage.rings.rational_field), 157
real() (sage.rings.integer.Integer method), 56
real() (sage.rings.rational.Rational method), 192
relative_discriminant() (sage.rings.rational_field.RationalField method), 172

Index
method), 167
relative_norm() (sage.rings.rational.Rational
method), 192
replace_prime() (sage.arith_multi_modular.MutableMultiModularBasis
method), 81
residue_field() (sage.rings.integer.IntegerRing_class
method), 10
residue_field() (sage.rings.rational_field.RationalField
method), 167
rising_factorial() (in module sage.arith.misc), 146
round() (sage.rings.integer.Integer method), 56
round() (sage.rings.rational.Rational method), 192
residue_field() (sage.rings.rational_field.RationalField
method), 10
residue_field() (sage.rings.rational_field.RationalField
method), 167
section() (sage.rings.rational.Z_to_Q method), 172
section() (sage.rings.rational.Q_to_Z method), 172
selmer_generators() (sage.rings.rational_field.RationalField
method), 167
selmer_group() (sage.rings.rational_field.RationalField
method), 168
selmer_group_iterator() (sage.rings.rational_field.RationalField
method), 168
selmer_space() (sage.rings.rational_field.RationalField
method), 168
Sigma (class in sage.arith.misc), 90
Sigma (class in sage.rings.integer.Integer method), 56
sign() (sage.rings.rational.Rational method), 193
signature() (sage.rings.rational_field.RationalField
method), 170
some_elements() (sage.rings.rational_field.RationalField
method), 170
sort_complex_numbers_for_display() (in module
date.arith.misc), 147
sqrt() (sage.rings.integer.Integer method), 56
sqrt() (sage.rings.rational.Rational method), 193
sqrtrem() (sage.rings.integer.Integer method), 57
squarefree_divisors() (in module sage.arith.misc), 148
squarefree_part() (sage.rings.integer.Integer method), 58
squarefree_part() (sage.rings.rational.Rational method), 194
str() (sage.rings.integer.Integer method), 58
str() (sage.rings.rational.Rational method), 195
subfactorial() (in module sage.arith.misc), 148
sum_of_k_squares() (in module sage.arith.misc), 149
support() (sage.rings.integer.Integer method), 59
support() (sage.rings.rational.Rational method), 195
test_bit() (sage.rings.integer.Integer method), 59
three_squares() (in module sage.arith.misc), 150
three_squares_pyx() (in module
dsage.rings.sum_of_squares), 75
trace() (sage.rings.rational.Rational method), 195
trailing_zero_bits() (sage.rings.integer.Integer method), 60
trial_division() (in module sage.arith.misc), 151
trial_division() (sage.rings.integer.Integer method), 60
trunc() (sage.rings.integer.Integer method), 61
trunc() (sage.rings.rational.Rational method), 196
two_squares() (in module sage.arith.misc), 152
two_squares_pyx() (in module
dsage.rings.sum_of_squares), 76
val_unit() (sage.rings.integer.Integer method), 61
val_unit() (sage.rings.rational.Rational method), 196
valuation() (in module sage.arith.misc), 153
valuation() (sage.rings.integer.Integer method), 62
valuation() (sage.rings.integer_ring.IntegerRing_class
method), 11
valuation() (sage.rings.rational.Rational method), 196
valuation() (sage.rings.rational_field.RationalField
method), 170
verify_bernoulli_mod_p() (in module sage.rings.bernoulli_mod_p), 68

X
XGCD() (in module sage.arith.misc), 91
xgcd() (in module sage.arith.misc), 154
xgcd() (sage.rings.integer.Integer method), 62
xgcd_int() (sage.rings.fast_arith.arith_int method), 73
xkcd() (in module sage.arith.misc), 155
xlcm() (in module sage.arith.misc), 156

Z
Z_to_Q (class in sage.rings.rational), 197
zeta() (sage.rings.integer_ring.IntegerRing_class method), 11
zeta() (sage.rings.rational_field.RationalField method), 171