
Fixed and Arbitrary Precision Numerical Fields

Release 10.6

The Sage Development Team

Jun 27, 2025

CONTENTS

1	Floating-Point Arithmetic	1
2	Interval Arithmetic	197
3	Exact Real Arithmetic	433
4	Indices and Tables	449
	Python Module Index	451
	Index	453

FLOATING-POINT ARITHMETIC

Sage supports arbitrary precision real (*RealField*) and complex fields (*ComplexField*). Sage also provides two optimized fixed precision fields for numerical computation, the real double (*RealDoubleField*) and complex double fields (*ComplexDoubleField*).

Real and complex double elements are optimized implementations that use the [GNU Scientific Library](#) for arithmetic and some special functions. Arbitrary precision real and complex numbers are implemented using the [MPFR](#) library, which builds on [GMP](#). In many cases, the [PARI](#) C-library is used to compute special functions when implementations aren't otherwise available.

1.1 Arbitrary precision floating point real numbers using GNU MPFR

AUTHORS:

- Kyle Schalm (2005-09)
- William Stein: bug fixes, examples, maintenance
- Didier Deshommes (2006-03-19): examples
- David Harvey (2006-09-20): compatibility with `Element._parent`
- William Stein (2006-10): default printing truncates to avoid base-2 rounding confusing (fix suggested by Bill Hart)
- Didier Deshommes: special constructor for QD numbers
- Paul Zimmermann (2008-01): added new functions from mpfr-2.3.0, replaced some, e.g., `sech = 1/cosh`, by their original mpfr version.
- Carl Witty (2008-02): define floating-point rank and associated functions; add some documentation
- Robert Bradshaw (2009-09): decimal literals, optimizations
- Jeroen Demeyer (2012-05-27): set the MPFR exponent range to the maximal possible value ([Issue #13033](#))
- Travis Scrimshaw (2012-11-02): Added doctests for full coverage
- Eviatar Bach (2013-06): Fixing numerical evaluation of `log_gamma`
- Vincent Klein (2017-06): `RealNumber` constructor support `gmpy2.mpfr`, `gmpy2.mpq` or `gmpy2.mpz` parameter. Add `__mpfr__` to class `RealNumber`.

This is a binding for the MPFR arbitrary-precision floating point library.

We define a class *RealField*, where each instance of *RealField* specifies a field of floating-point numbers with a specified precision and rounding mode. Individual floating-point numbers are of *RealNumber*.

In Sage (as in MPFR), floating-point numbers of precision p are of the form $sm2^{e-p}$, where $s \in \{-1, 1\}$, $2^{p-1} \leq m < 2^p$, and $-2^B + 1 \leq e \leq 2^B - 1$ where $B = 30$ on 32-bit systems and $B = 62$ on 64-bit systems; additionally, there are the special values `+0`, `-0`, `+infinity`, `-infinity` and `NaN` (which stands for Not-a-Number).

Operations in this module which are direct wrappers of MPFR functions are “correctly rounded”; we briefly describe what this means. Assume that you could perform the operation exactly, on real numbers, to get a result r . If this result can be represented as a floating-point number, then we return that number.

Otherwise, the result r is between two floating-point numbers. For the directed rounding modes (round to plus infinity, round to minus infinity, round to zero), we return the floating-point number in the indicated direction from r . For round to nearest, we return the floating-point number which is nearest to r .

This leaves one case unspecified: in round to nearest mode, what happens if r is exactly halfway between the two nearest floating-point numbers? In that case, we round to the number with an even mantissa (the mantissa is the number m in the representation above).

Consider the ordered set of floating-point numbers of precision p . (Here we identify $+0$ and -0 , and ignore NaN.) We can give a bijection between these floating-point numbers and a segment of the integers, where 0 maps to 0 and adjacent floating-point numbers map to adjacent integers. We call the integer corresponding to a given floating-point number the “floating-point rank” of the number. (This is not standard terminology; I just made it up.)

EXAMPLES:

A difficult conversion:

```
sage: RR(sys.maxsize)
9.22337203685478e18      # 64-bit
2.14748364700000e9      # 32-bit
```

```
>>> from sage.all import *
>>> RR(sys.maxsize)
9.22337203685478e18      # 64-bit
2.14748364700000e9      # 32-bit
```

```
class sage.rings.real_mpfr.QQtoRR
```

Bases: Map

```
class sage.rings.real_mpfr.RRtoRR
```

Bases: Map

```
section()
```

EXAMPLES:

```
sage: from sage.rings.real_mpfr import RRtoRR
sage: R10 = RealField(10)
sage: R100 = RealField(100)
sage: f = RRtoRR(R100, R10)
sage: f.section()
Generic map:
  From: Real Field with 10 bits of precision
  To:   Real Field with 100 bits of precision
```

```
>>> from sage.all import *
>>> from sage.rings.real_mpfr import RRtoRR
>>> R10 = RealField(Integer(10))
>>> R100 = RealField(Integer(100))
>>> f = RRtoRR(R100, R10)
>>> f.section()
Generic map:
```

(continues on next page)

(continued from previous page)

```
From: Real Field with 10 bits of precision
To:   Real Field with 100 bits of precision
```

```
sage.rings.real_mpfr.RealField(prec=53, sci_not=0, rnd='MPFR_RNDN')
```

`RealField(prec, sci_not, rnd):`

INPUT:

- `prec` – integer (default: 53); precision `prec` is the number of bits used to represent the mantissa of a floating-point number. The precision can be any integer between `mpfr_prec_min()` and `mpfr_prec_max()`. In the current implementation, `mpfr_prec_min()` is equal to 2.
- `sci_not` – boolean (default: False); if True, always display using scientific notation. If False, display using scientific notation only for very large or very small numbers.
- `rnd` – string; the rounding mode:
 - 'RNDN' – (default) round to nearest (ties go to the even number): Knuth says this is the best choice to prevent “floating point drift”
 - 'RNDD' – round towards minus infinity
 - 'RNDZ' – round towards zero
 - 'RNDU' – round towards plus infinity
 - 'RNDA' – round away from zero
 - 'RNDF' – faithful rounding (currently experimental; not guaranteed correct for every operation)
 - for specialized applications, the rounding mode can also be given as an integer value of type `mpfr_rnd_t`. However, the exact values are unspecified.

EXAMPLES:

```
sage: RealField(10)
Real Field with 10 bits of precision
sage: RealField()
Real Field with 53 bits of precision
sage: RealField(100000)
Real Field with 100000 bits of precision
```

```
>>> from sage.all import *
>>> RealField(Integer(10))
Real Field with 10 bits of precision
>>> RealField()
Real Field with 53 bits of precision
>>> RealField(Integer(100000))
Real Field with 100000 bits of precision
```

Here we show the effect of rounding:

```
sage: R17d = RealField(17, rnd='RNDD')
sage: a = R17d(1)/R17d(3); a.exact_rational()
87381/262144
sage: R17u = RealField(17, rnd='RNDU')
sage: a = R17u(1)/R17u(3); a.exact_rational()
43691/131072
```

```
>>> from sage.all import *
>>> R17d = RealField(Integer(17), rnd='RNDD')
>>> a = R17d(Integer(1))/R17d(Integer(3)); a.exact_rational()
87381/262144
>>> R17u = RealField(Integer(17), rnd='RNDU')
>>> a = R17u(Integer(1))/R17u(Integer(3)); a.exact_rational()
43691/131072
```

Note

The default precision is 53, since according to the MPFR manual: ‘mpfr should be able to exactly reproduce all computations with double-precision machine floating-point numbers (double type in C), except the default exponent range is much wider and subnormal numbers are not implemented.’

See also

- `sage.rings.real_mpfr`
- `sage.rings.real_arb.RealBallField` (real numbers with rigorous error bounds)

class `sage.rings.real_mpfr.RealField_class`

Bases: `RealField`

An approximation to the field of real numbers using floating point numbers with any specified precision. Answers derived from calculations in this approximation may differ from what they would be if those calculations were performed in the true field of real numbers. This is due to the rounding errors inherent to finite precision calculations.

See also

- `sage.rings.real_mpfr`
- `sage.rings.real_arb.RealBallField` (real numbers with rigorous error bounds)
- `sage.rings.complex_mpfr`

algebraic_closure()

Return the algebraic closure of `self`, i.e., the complex field with the same precision.

EXAMPLES:

```
sage: RR.algebraic_closure()
Complex Field with 53 bits of precision
sage: RR.algebraic_closure() is CC
True
sage: RealField(100, rnd='RNDD').algebraic_closure()
Complex Field with 100 bits of precision
sage: RealField(100).algebraic_closure()
Complex Field with 100 bits of precision
```



```

>>> from sage.all import *
>>> RR.algebraic_closure()
Complex Field with 53 bits of precision
>>> RR.algebraic_closure() is CC
True
>>> RealField(Integer(100), rnd='RNDD').algebraic_closure()
Complex Field with 100 bits of precision
>>> RealField(Integer(100)).algebraic_closure()
Complex Field with 100 bits of precision

```

catalan_constant()

Return Catalan's constant to the precision of this field.

EXAMPLES:

```

sage: RealField(100).catalan_constant()
0.91596559417721901505460351493

```

```

>>> from sage.all import *
>>> RealField(Integer(100)).catalan_constant()
0.91596559417721901505460351493

```

characteristic()

Return 0, since the field of real numbers has characteristic 0.

EXAMPLES:

```

sage: RealField(10).characteristic()
0

```

```

>>> from sage.all import *
>>> RealField(Integer(10)).characteristic()
0

```

complex_field()

Return complex field of the same precision.

EXAMPLES:

```

sage: RR.complex_field()
Complex Field with 53 bits of precision
sage: RR.complex_field() is CC
True
sage: RealField(100, rnd='RNDD').complex_field()
Complex Field with 100 bits of precision
sage: RealField(100).complex_field()
Complex Field with 100 bits of precision

```

```

>>> from sage.all import *
>>> RR.complex_field()
Complex Field with 53 bits of precision
>>> RR.complex_field() is CC
True

```

(continues on next page)

(continued from previous page)

```
>>> RealField(Integer(100), rnd='RNDD').complex_field()
Complex Field with 100 bits of precision
>>> RealField(Integer(100)).complex_field()
Complex Field with 100 bits of precision
```

construction()

Return the functorial construction of `self`, namely, completion of the rational numbers with respect to the prime at ∞ .

Also preserves other information that makes this field unique (e.g. precision, rounding, print mode).

EXAMPLES:

```
sage: R = RealField(100, rnd='RNDU')
sage: c, S = R.construction(); S
Rational Field
sage: R == c(S)
True
```

```
>>> from sage.all import *
>>> R = RealField(Integer(100), rnd='RNDU')
>>> c, S = R.construction(); S
Rational Field
>>> R == c(S)
True
```

euler_constant()

Return Euler's gamma constant to the precision of this field.

EXAMPLES:

```
sage: RealField(100).euler_constant()
0.57721566490153286060651209008
```

```
>>> from sage.all import *
>>> RealField(Integer(100)).euler_constant()
0.57721566490153286060651209008
```

factorial(n)

Return the factorial of the integer `n` as a real number.

EXAMPLES:

```
sage: RR.factorial(0)
1.000000000000000
sage: RR.factorial(1000000)
8.26393168833124e5565708
sage: RR.factorial(-1)
Traceback (most recent call last):
...
ArithmeticError: n must be nonnegative
```

Return the `i`-th generator of `self`.

[illegible]

```
>>> from sage.all import *
>>> R=RealField(Integer(100))
>>> R.gen(Integer(0))
1.00000000000000000000000000000000
>>> R.gen(Integer(1))
Traceback (most recent call last):
...
IndexError: self has only one generator
```

Return a tuple of generators.

```
sage: RR.gens()
(1.000000000000000,)
```

```
>>> from sage.all import *
>>> RR.gens()
(1.0000000000000000,)
```

Return `False`, since a real field (represented using finite precision) is not exact.

```
sage: RR.is_exact()
False
sage: RealField(100).is_exact()
False
```

```
>>> from sage.all import *
>>> RR.is_exact()
False
>>> RealField(Integer(100)).is_exact()
False
```

log2()

Return $\log(2)$ (i.e., the natural log of 2) to the precision of this field.

EXAMPLES:

```
sage: R=RealField(100)
sage: R.log2()
0.69314718055994530941723212146
sage: R(2).log()
0.69314718055994530941723212146
```

```
>>> from sage.all import *
>>> R=RealField(Integer(100))
>>> R.log2()
0.69314718055994530941723212146
>>> R(Integer(2)).log()
0.69314718055994530941723212146
```

name()

Return the name of `self`, which encodes the precision and rounding convention.

EXAMPLES:

```
sage: RR.name()
'RealField53_0'
sage: RealField(100, rnd='RNDU').name()
'RealField100_2'
```

```
>>> from sage.all import *
>>> RR.name()
'RealField53_0'
>>> RealField(Integer(100), rnd='RNDU').name()
'RealField100_2'
```

ngens()

Return the number of generators.

EXAMPLES:

```
sage: RR.ngens()
1
```

```
>>> from sage.all import *
>>> RR.ngens()
1
```

pi()

Return π to the precision of this field.

EXAMPLES:

```
sage: R = RealField(100)
sage: R.pi()
3.1415926535897932384626433833
sage: R.pi().sqrt()/2
0.88622692545275801364908374167
sage: R = RealField(150)
sage: R.pi().sqrt()/2
0.88622692545275801364908374167057259139877473
```

```
>>> from sage.all import *
>>> R = RealField(Integer(100))
>>> R.pi()
3.1415926535897932384626433833
>>> R.pi().sqrt()/Integer(2)
0.88622692545275801364908374167
>>> R = RealField(Integer(150))
>>> R.pi().sqrt()/Integer(2)
0.88622692545275801364908374167057259139877473
```

prec()

Return the precision of `self`.

EXAMPLES:

```
sage: RR.precision()
53
sage: RealField(20).precision()
20
```

```
>>> from sage.all import *
>>> RR.precision()
53
>>> RealField(Integer(20)).precision()
20
```

precision()

Return the precision of `self`.

EXAMPLES:

```
sage: RR.precision()
53
sage: RealField(20).precision()
20
```

```
>>> from sage.all import *
>>> RR.precision()
53
>>> RealField(Integer(20)).precision()
20
```

random_element (*min=-1, max=1, distribution=None*)

Return a uniformly distributed random number between `min` and `max` (default -1 to 1).

Warning

The argument `distribution` is ignored—the random number is from the uniform distribution.

EXAMPLES:

```
sage: r = RealField(100).random_element(-5, 10)
sage: r.parent() is RealField(100)
True
sage: -5 <= r <= 10
True
```

```
>>> from sage.all import *
>>> r = RealField(Integer(100)).random_element(-Integer(5), Integer(10))
>>> r.parent() is RealField(Integer(100))
True
>>> -Integer(5) <= r <= Integer(10)
True
```

rounding_mode ()

Return the rounding mode.

EXAMPLES:

```
sage: RR.rounding_mode()
'RNDN'
sage: RealField(20, rnd='RNDZ').rounding_mode()
'RNDZ'
sage: RealField(20, rnd='RNDU').rounding_mode()
'RNDU'
sage: RealField(20, rnd='RNDD').rounding_mode()
'RNDD'
```

```
>>> from sage.all import *
>>> RR.rounding_mode()
'RNDN'
>>> RealField(Integer(20), rnd='RNDZ').rounding_mode()
'RNDZ'
>>> RealField(Integer(20), rnd='RNDU').rounding_mode()
'RNDU'
>>> RealField(Integer(20), rnd='RNDD').rounding_mode()
'RNDD'
```

scientific_notation (*status=None*)

Set or return the scientific notation printing flag. If this flag is `True` then real numbers with this space as parent print using scientific notation.

INPUT:

- `status` – boolean optional flag

EXAMPLES:

```

sage: RR.scientific_notation()
False
sage: elt = RR(0.2512); elt
0.2512000000000000
sage: RR.scientific_notation(True)
sage: elt
2.512000000000000e-1
sage: RR.scientific_notation()
True
sage: RR.scientific_notation(False)
sage: elt
0.2512000000000000
sage: R = RealField(20, sci_not=1)
sage: R.scientific_notation()
True
sage: R(0.2512)
2.5120e-1

```

```

>>> from sage.all import *
>>> RR.scientific_notation()
False
>>> elt = RR(RealNumber('0.2512')); elt
0.2512000000000000
>>> RR.scientific_notation(True)
>>> elt
2.512000000000000e-1
>>> RR.scientific_notation()
True
>>> RR.scientific_notation(False)
>>> elt
0.2512000000000000
>>> R = RealField(Integer(20), sci_not=Integer(1))
>>> R.scientific_notation()
True
>>> R(RealNumber('0.2512'))
2.5120e-1

```

to_prec (*prec*)

Return the real field that is identical to `self`, except that it has the specified precision.

EXAMPLES:

```

sage: RR.to_prec(212)
Real Field with 212 bits of precision
sage: R = RealField(30, rnd="RNDZ")
sage: R.to_prec(300)
Real Field with 300 bits of precision and rounding RNDZ

```

```

>>> from sage.all import *
>>> RR.to_prec(Integer(212))
Real Field with 212 bits of precision

```

(continues on next page)

(continued from previous page)

```
>>> R = RealField(Integer(30), rnd="RNDZ")
>>> R.to_prec(Integer(300))
Real Field with 300 bits of precision and rounding RNDZ
```

zeta ($n=2$)

Return an n -th root of unity in the real field, if one exists, or raise a `ValueError` otherwise.

EXAMPLES:

```
sage: R = RealField()
sage: R.zeta()
-1.0000000000000000
sage: R.zeta(1)
1.0000000000000000
sage: R.zeta(5)
Traceback (most recent call last):
...
ValueError: No 5th root of unity in self
```

```
>>> from sage.all import *
>>> R = RealField()
>>> R.zeta()
-1.0000000000000000
>>> R.zeta(Integer(1))
1.0000000000000000
>>> R.zeta(Integer(5))
Traceback (most recent call last):
...
ValueError: No 5th root of unity in self
```

class `sage.rings.real_mpf.RealLiteral`

Bases: `RealNumber`

Real literals are created in parsing and provide a way to allow casting into higher precision rings.

base

literal

numerical_approx ($prec=None$, $digits=None$, $algorithm=None$)

Change the precision of `self` to `prec` bits or `digits` decimal digits.

INPUT:

- `prec` – precision in bits
- `digits` – precision in decimal digits (only used if `prec` is not given)
- `algorithm` – ignored for real numbers

If neither `prec` nor `digits` is given, the default precision is 53 bits (roughly 16 digits).

OUTPUT: a `RealNumber` with the given precision

EXAMPLES:


```
sage: (1.3).numerical_approx()
1.3000000000000000
sage: n(1.3, 120)
1.3000000000000000000000000000000000000000000000000
```

```
>>> from sage.all import *
>>> (RealNumber('1.3')).numerical_approx()
1.3000000000000000
>>> n(RealNumber('1.3'), Integer(120))
1.3000000000000000000000000000000000000000000000000
```

Compare with:

```
sage: RealField(120)(RR(13/10))
1.3000000000000000000000000000000000000000000000000
sage: n(RR(13/10), 120)
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 120 bits, use at most 53 bits
```

```
>>> from sage.all import *
>>> RealField(Integer(120))(RR(Integer(13)/Integer(10)))
1.3000000000000000000000000000000000000000000000000
>>> n(RR(Integer(13)/Integer(10)), Integer(120))
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 120 bits, use at most 53 bits
```

The result is a non-literal:

```
sage: type(1.3)
<class 'sage.rings.real_mpfr.RealLiteral'>
sage: type(n(1.3))
<class 'sage.rings.real_mpfr.RealNumber'>
```

```
>>> from sage.all import *
>>> type(RealNumber('1.3'))
<class 'sage.rings.real_mpfr.RealLiteral'>
>>> type(n(RealNumber('1.3')))
<class 'sage.rings.real_mpfr.RealNumber'>
```

class sage.rings.real_mpfr.**RealNumber**

Bases: `RingElement`

A floating point approximation to a real number using any specified precision. Answers derived from calculations with such approximations may differ from what they would be if those calculations were performed with true real numbers. This is due to the rounding errors inherent to finite precision calculations.

The approximation is printed to slightly fewer digits than its internal precision, in order to avoid confusing roundoff issues that occur because numbers are stored internally in binary.

agm (*other*)

Return the arithmetic-geometric mean of *self* and *other*.

The arithmetic-geometric mean is the common limit of the sequences u_n and v_n , where u_0 is `self`, v_0 is `other`, u_{n+1} is the arithmetic mean of u_n and v_n , and v_{n+1} is the geometric mean of u_n and v_n . If any operand is negative, the return value is `NaN`.

INPUT:

- `right` – another real number

OUTPUT: the AGM of `self` and `other`

EXAMPLES:

```
sage: a = 1.5
sage: b = 2.5
sage: a.agm(b)
1.96811775182478
sage: RealField(200)(a).agm(b)
1.9681177518247777389894630877503739489139488203685819712291
sage: a.agm(100)
28.1189391225320
```

```
>>> from sage.all import *
>>> a = RealNumber('1.5')
>>> b = RealNumber('2.5')
>>> a.agm(b)
1.96811775182478
>>> RealField(Integer(200))(a).agm(b)
1.9681177518247777389894630877503739489139488203685819712291
>>> a.agm(Integer(100))
28.1189391225320
```

The AGM always lies between the geometric and arithmetic mean:

```
sage: sqrt(a*b) < a.agm(b) < (a+b)/2
True
```

```
>>> from sage.all import *
>>> sqrt(a*b) < a.agm(b) < (a+b)/Integer(2)
True
```

It is, of course, symmetric:

```
sage: b.agm(a)
1.96811775182478
```

```
>>> from sage.all import *
>>> b.agm(a)
1.96811775182478
```

and satisfies the relation $AGM(ra, rb) = rAGM(a, b)$:

```
sage: (2*a).agm(2*b) / 2
1.96811775182478
sage: (3*a).agm(3*b) / 3
1.96811775182478
```

```
>>> from sage.all import *
>>> (Integer(2)*a).agm(Integer(2)*b) / Integer(2)
1.96811775182478
>>> (Integer(3)*a).agm(Integer(3)*b) / Integer(3)
1.96811775182478
```

It is also related to the elliptic integral

$$\int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - m \sin^2 \theta}}.$$

```
sage: m = (a-b)^2/(a+b)^2
sage: E = numerical_integral(1/sqrt(1-m*sin(x)^2), 0, RR.pi()/2)[0] #_
↪needs sage.symbolic
sage: RR.pi()/4 * (a+b)/E #_
↪needs sage.symbolic
1.96811775182478
```

```
>>> from sage.all import *
>>> m = (a-b)**Integer(2)/(a+b)**Integer(2)
>>> E = numerical_integral(Integer(1)/sqrt(Integer(1)-m*sin(x)**Integer(2)), #_
↪Integer(0), RR.pi()/Integer(2))[Integer(0)] # needs sage.symbolic
>>> RR.pi()/Integer(4) * (a+b)/E #_
↪ # needs sage.symbolic
1.96811775182478
```

algdep(*n*)

Return a polynomial of degree at most *n* which is approximately satisfied by this number.

Note

The resulting polynomial need not be irreducible, and indeed usually won't be if this number is a good approximation to an algebraic number of degree less than *n*.

ALGORITHM:

Uses the PARI C-library `pari:algdep` command.

EXAMPLES:

```
sage: r = sqrt(2.0); r
1.41421356237310
sage: r.algebraic_dependency(5)
x^2 - 2
```

```
>>> from sage.all import *
>>> r = sqrt(RealNumber('2.0')); r
1.41421356237310
>>> r.algebraic_dependency(Integer(5))
x^2 - 2
```

algebraic_dependency(*n*)

Return a polynomial of degree at most n which is approximately satisfied by this number.

Note

The resulting polynomial need not be irreducible, and indeed usually won't be if this number is a good approximation to an algebraic number of degree less than n .

ALGORITHM:

Uses the PARI C-library `pari:algdep` command.

EXAMPLES:

```
sage: r = sqrt(2.0); r
1.41421356237310
sage: r.algebraic_dependency(5)
x^2 - 2
```

```
>>> from sage.all import *
>>> r = sqrt(RealNumber('2.0')); r
1.41421356237310
>>> r.algebraic_dependency(Integer(5))
x^2 - 2
```

arccos()

Return the inverse cosine of `self`.

EXAMPLES:

```
sage: q = RR.pi()/3
sage: i = q.cos()
sage: i.arccos() == q
True
```

```
>>> from sage.all import *
>>> q = RR.pi()/Integer(3)
>>> i = q.cos()
>>> i.arccos() == q
True
```

arccosh()

Return the hyperbolic inverse cosine of `self`.

EXAMPLES:

```
sage: q = RR.pi()/2
sage: i = q.cosh(); i
2.50917847865806
sage: q == i.arccosh()
True
```

```
>>> from sage.all import *
>>> q = RR.pi()/Integer(2)
>>> i = q.cosh() ; i
2.50917847865806
>>> q == i.arccosh()
True
```

arccoth()

Return the inverse hyperbolic cotangent of `self`.

EXAMPLES:

```
sage: q = RR.pi()/5
sage: i = q.coth()
sage: i.arccoth() == q
True
```

```
>>> from sage.all import *
>>> q = RR.pi()/Integer(5)
>>> i = q.coth()
>>> i.arccoth() == q
True
```

arccsch()

Return the inverse hyperbolic cosecant of `self`.

EXAMPLES:

```
sage: i = RR.pi()/5
sage: q = i.csch()
sage: q.arccsch() == i
True
```

```
>>> from sage.all import *
>>> i = RR.pi()/Integer(5)
>>> q = i.csch()
>>> q.arccsch() == i
True
```

arcsech()

Return the inverse hyperbolic secant of `self`.

EXAMPLES:

```
sage: i = RR.pi()/3
sage: q = i.sech()
sage: q.arcsech() == i
True
```

```
>>> from sage.all import *
>>> i = RR.pi()/Integer(3)
>>> q = i.sech()
>>> q.arcsech() == i
True
```

arcsin()

Return the inverse sine of `self`.

EXAMPLES:

```
sage: q = RR.pi()/5
sage: i = q.sin()
sage: i.arcsin() == q
True
sage: i.arcsin() - q
0.0000000000000000
```

```
>>> from sage.all import *
>>> q = RR.pi()/Integer(5)
>>> i = q.sin()
>>> i.arcsin() == q
True
>>> i.arcsin() - q
0.0000000000000000
```

arcsinh()

Return the hyperbolic inverse sine of `self`.

EXAMPLES:

```
sage: q = RR.pi()/7
sage: i = q.sinh() ; i
0.464017630492991
sage: i.arcsinh() - q
0.0000000000000000
```

```
>>> from sage.all import *
>>> q = RR.pi()/Integer(7)
>>> i = q.sinh() ; i
0.464017630492991
>>> i.arcsinh() - q
0.0000000000000000
```

arctan()

Return the inverse tangent of `self`.

EXAMPLES:

```
sage: q = RR.pi()/5
sage: i = q.tan()
sage: i.arctan() == q
True
```

```
>>> from sage.all import *
>>> q = RR.pi()/Integer(5)
>>> i = q.tan()
>>> i.arctan() == q
True
```

arctanh()

Return the hyperbolic inverse tangent of `self`.

EXAMPLES:

```
sage: q = RR.pi()/7
sage: i = q.tanh() ; i
0.420911241048535
sage: i.arctanh() - q
0.000000000000000
```

```
>>> from sage.all import *
>>> q = RR.pi()/Integer(7)
>>> i = q.tanh() ; i
0.420911241048535
>>> i.arctanh() - q
0.000000000000000
```

as_integer_ratio()

Return a coprime pair of integers (a, b) such that `self` equals a / b exactly.

EXAMPLES:

```
sage: RR(0).as_integer_ratio()
(0, 1)
sage: RR(1/3).as_integer_ratio()
(6004799503160661, 18014398509481984)
sage: RR(37/16).as_integer_ratio()
(37, 16)
sage: RR(3^60).as_integer_ratio()
(42391158275216203520420085760, 1)
sage: RR('nan').as_integer_ratio()
Traceback (most recent call last):
...
ValueError: unable to convert NaN to a rational number
```

```
>>> from sage.all import *
>>> RR(Integer(0)).as_integer_ratio()
(0, 1)
>>> RR(Integer(1)/Integer(3)).as_integer_ratio()
(6004799503160661, 18014398509481984)
>>> RR(Integer(37)/Integer(16)).as_integer_ratio()
(37, 16)
>>> RR(Integer(3)**Integer(60)).as_integer_ratio()
(42391158275216203520420085760, 1)
>>> RR('nan').as_integer_ratio()
Traceback (most recent call last):
...
ValueError: unable to convert NaN to a rational number
```

This coincides with Python floats:

```
sage: pi = RR.pi()
sage: pi.as_integer_ratio()
```

(continues on next page)

(continued from previous page)

```
(884279719003555, 281474976710656)
sage: float(pi).as_integer_ratio() == pi.as_integer_ratio()
True
```

```
>>> from sage.all import *
>>> pi = RR.pi()
>>> pi.as_integer_ratio()
(884279719003555, 281474976710656)
>>> float(pi).as_integer_ratio() == pi.as_integer_ratio()
True
```

ceil()Return the ceiling of `self`.

EXAMPLES:

```
sage: (2.99).ceil()
3
sage: (2.00).ceil()
2
sage: (2.01).ceil()
3
```

```
>>> from sage.all import *
>>> (RealNumber('2.99')).ceil()
3
>>> (RealNumber('2.00')).ceil()
2
>>> (RealNumber('2.01')).ceil()
3
```

```
sage: ceil(10^16 * 1.0)
10000000000000000
sage: ceil(10^17 * 1.0)
100000000000000000
sage: ceil(RR(+infinity))
Traceback (most recent call last):
...
ValueError: Calling ceil() on infinity or NaN
```

```
>>> from sage.all import *
>>> ceil(Integer(10)**Integer(16) * RealNumber('1.0'))
10000000000000000
>>> ceil(Integer(10)**Integer(17) * RealNumber('1.0'))
100000000000000000
>>> ceil(RR(+infinity))
Traceback (most recent call last):
...
ValueError: Calling ceil() on infinity or NaN
```

ceiling()Return the ceiling of `self`.


```
>>> from sage.all import *
>>> ceil(Integer(10)**Integer(16) * RealNumber('1.0'))
10000000000000000
>>> ceil(Integer(10)**Integer(17) * RealNumber('1.0'))
100000000000000000
>>> ceil(RR(+infinity))
Traceback (most recent call last):
...
ValueError: Calling ceil() on infinity or NaN
```

`conjugate()`

Return the complex conjugate of this real number, which is the number itself.

EXAMPLES:

```
>>> from sage.all import *
>>> x = RealField(Integer(100)) (RealNumber('1.238'))
>>> x.conjugate()
1.2380000000000000000000000000000
```

cos ()

Return the cosine of `self`.

EXAMPLES:

```
sage: t=RR.pi()/2
sage: t.cos()
6.12323399573677e-17
```

```
>>> from sage.all import *
>>> t=RR.pi()/Integer(2)
>>> t.cos()
6.12323399573677e-17
```

cosh()

Return the hyperbolic cosine of `self`.

EXAMPLES:

```
sage: q = RR.pi()/12
sage: q.cosh()
1.03446564009551
```

```
>>> from sage.all import *
>>> q = RR.pi()/Integer(12)
>>> q.cosh()
1.03446564009551
```

cot()

Return the cotangent of `self`.

EXAMPLES:

```
sage: RealField(100)(2).cot()
-0.45765755436028576375027741043
```

```
>>> from sage.all import *
>>> RealField(Integer(100))(Integer(2)).cot()
-0.45765755436028576375027741043
```

coth()

Return the hyperbolic cotangent of `self`.

EXAMPLES:

```
sage: RealField(100)(2).coth()
1.0373147207275480958778097648
```

```
>>> from sage.all import *
>>> RealField(Integer(100))(Integer(2)).coth()
1.0373147207275480958778097648
```

csc()

Return the cosecant of `self`.

EXAMPLES:

```
sage: RealField(100)(2).csc()
1.0997501702946164667566973970
```

```
>>> from sage.all import *
>>> RealField(Integer(100))(Integer(2)).csc()
1.0997501702946164667566973970
```

csch()

Return the hyperbolic cosecant of `self`.

EXAMPLES:

```
sage: RealField(100)(2).csch()
0.27572056477178320775835148216
```

```
>>> from sage.all import *
>>> RealField(Integer(100))(Integer(2)).csch()
0.27572056477178320775835148216
```

cube_root()

Return the cubic root (defined over the real numbers) of `self`.

EXAMPLES:

```
sage: r = 125.0; r.cube_root()
5.000000000000000
sage: r = -119.0
sage: r.cube_root()^3 - r      # illustrates precision loss
-1.42108547152020e-14
```

```
>>> from sage.all import *
>>> r = RealNumber('125.0'); r.cube_root()
5.000000000000000
>>> r = -RealNumber('119.0')
>>> r.cube_root()**Integer(3) - r      # illustrates precision loss
-1.42108547152020e-14
```

eint()

Return the exponential integral of this number.

EXAMPLES:

```
sage: r = 1.0
sage: r.eint()
1.89511781635594
```

```
>>> from sage.all import *
>>> r = RealNumber('1.0')
>>> r.eint()
1.89511781635594
```

```
sage: r = -1.0
sage: r.eint()
-0.219383934395520
```

```
>>> from sage.all import *
>>> r = -RealNumber('1.0')
>>> r.eint()
-0.219383934395520
```

epsilon (*field=None*)

Return $\text{abs}(\text{self})$ divided by 2^b where b is the precision in bits of self . Equivalently, return $\text{abs}(\text{self})$ multiplied by the `ulp()` of 1.

This is a scale-invariant version of `ulp()` and it lies in $[u/2, u)$ where u is $\text{self.ulp}()$ (except in the case of zero or underflow).

INPUT:

- `field` – *RealField* used as parent of the result. If not specified, use `parent(self)`

OUTPUT:

```
field(self.abs() / 2^self.precision())
```

EXAMPLES:

```
sage: RR(2^53).epsilon()
1.000000000000000
sage: RR(0).epsilon()
0.000000000000000
sage: a = RR.pi()
sage: a.epsilon()
3.48786849800863e-16
sage: a.ulp()/2, a.ulp()
(2.22044604925031e-16, 4.44089209850063e-16)
sage: a / 2^a.precision()
3.48786849800863e-16
sage: (-a).epsilon()
3.48786849800863e-16
```

```
>>> from sage.all import *
>>> RR(Integer(2)**Integer(53)).epsilon()
1.000000000000000
>>> RR(Integer(0)).epsilon()
0.000000000000000
>>> a = RR.pi()
>>> a.epsilon()
3.48786849800863e-16
>>> a.ulp()/Integer(2), a.ulp()
(2.22044604925031e-16, 4.44089209850063e-16)
>>> a / Integer(2)**a.precision()
3.48786849800863e-16
>>> (-a).epsilon()
3.48786849800863e-16
```

We use a different field:

```
sage: a = RealField(256).pi()
sage: a.epsilon()
```

(continues on next page)

(continued from previous page)

```

2.
↪713132368784788677624750042896586252980746500631892201656843478528498954308e-
↪77
sage: e = a.epsilon(RealField(64))
sage: e
2.71313236878478868e-77
sage: parent(e)
Real Field with 64 bits of precision
sage: e = a.epsilon(QQ)
Traceback (most recent call last):
...
TypeError: field argument must be a RealField

```

```

>>> from sage.all import *
>>> a = RealField(Integer(256)).pi()
>>> a.epsilon()
2.
↪713132368784788677624750042896586252980746500631892201656843478528498954308e-
↪77
>>> e = a.epsilon(RealField(Integer(64)))
>>> e
2.71313236878478868e-77
>>> parent(e)
Real Field with 64 bits of precision
>>> e = a.epsilon(QQ)
Traceback (most recent call last):
...
TypeError: field argument must be a RealField

```

Special values:

```

sage: RR('nan').epsilon()
NaN
sage: parent(RR('nan').epsilon(RealField(42)))
Real Field with 42 bits of precision
sage: RR('+Inf').epsilon()
+infinity
sage: RR('-Inf').epsilon()
+infinity

```

```

>>> from sage.all import *
>>> RR('nan').epsilon()
NaN
>>> parent(RR('nan').epsilon(RealField(Integer(42))))
Real Field with 42 bits of precision
>>> RR('+Inf').epsilon()
+infinity
>>> RR('-Inf').epsilon()
+infinity

```

erf()Return the value of the error function on `self`.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).erf()
0.995322265018953
sage: R(6).erf()
1.000000000000000
```

```
>>> from sage.all import *
>>> R = RealField(Integer(53))
>>> R(Integer(2)).erf()
0.995322265018953
>>> R(Integer(6)).erf()
1.000000000000000
```

erfc()Return the value of the complementary error function on `self`, i.e., $1 - \text{erf}(\text{self})$.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).erfc()
0.00467773498104727
sage: R(6).erfc()
2.15197367124989e-17
```

```
>>> from sage.all import *
>>> R = RealField(Integer(53))
>>> R(Integer(2)).erfc()
0.00467773498104727
>>> R(Integer(6)).erfc()
2.15197367124989e-17
```

exact_rational()

Return the exact rational representation of this floating-point number.

EXAMPLES:

```
sage: RR(0).exact_rational()
0
sage: RR(1/3).exact_rational()
6004799503160661/18014398509481984
sage: RR(37/16).exact_rational()
37/16
sage: RR(3^60).exact_rational()
42391158275216203520420085760
sage: RR(3^60).exact_rational() - 3^60
6125652559
sage: RealField(5)(-pi).exact_rational()
↪needs sage.symbolic
-25/8
```

```
>>> from sage.all import *
>>> RR(Integer(0)).exact_rational()
```

(continues on next page)

(continued from previous page)

```

0
>>> RR(Integer(1)/Integer(3)).exact_rational()
6004799503160661/18014398509481984
>>> RR(Integer(37)/Integer(16)).exact_rational()
37/16
>>> RR(Integer(3)**Integer(60)).exact_rational()
42391158275216203520420085760
>>> RR(Integer(3)**Integer(60)).exact_rational() - Integer(3)**Integer(60)
6125652559
>>> RealField(Integer(5))(-pi).exact_rational()
↪      # needs sage.symbolic
-25/8

```

exp()Return e^{self} .

EXAMPLES:

```

sage: r = 0.0
sage: r.exp()
1.0000000000000000

```

```

>>> from sage.all import *
>>> r = RealNumber('0.0')
>>> r.exp()
1.0000000000000000

```

```

sage: r = 32.3
sage: a = r.exp(); a
1.06588847274864e14
sage: a.log()
32.300000000000000

```

```

>>> from sage.all import *
>>> r = RealNumber('32.3')
>>> a = r.exp(); a
1.06588847274864e14
>>> a.log()
32.300000000000000

```

```

sage: r = -32.3
sage: r.exp()
9.38184458849869e-15

```

```

>>> from sage.all import *
>>> r = -RealNumber('32.3')
>>> r.exp()
9.38184458849869e-15

```

exp10()Return 10^{self} .

EXAMPLES:

```
sage: r = 0.0
sage: r.exp10()
1.000000000000000
```

```
>>> from sage.all import *
>>> r = RealNumber('0.0')
>>> r.exp10()
1.000000000000000
```

```
sage: r = 32.0
sage: r.exp10()
1.000000000000000e32
```

```
>>> from sage.all import *
>>> r = RealNumber('32.0')
>>> r.exp10()
1.000000000000000e32
```

```
sage: r = -32.3
sage: r.exp10()
5.01187233627276e-33
```

```
>>> from sage.all import *
>>> r = -RealNumber('32.3')
>>> r.exp10()
5.01187233627276e-33
```

exp2()Return 2^{self} .

EXAMPLES:

```
sage: r = 0.0
sage: r.exp2()
1.000000000000000
```

```
>>> from sage.all import *
>>> r = RealNumber('0.0')
>>> r.exp2()
1.000000000000000
```

```
sage: r = 32.0
sage: r.exp2()
4.29496729600000e9
```

```
>>> from sage.all import *
>>> r = RealNumber('32.0')
>>> r.exp2()
4.29496729600000e9
```



```
sage: r = -32.3
sage: r.exp2()
1.89117248253021e-10
```

```
>>> from sage.all import *
>>> r = -RealNumber('32.3')
>>> r.exp2()
1.89117248253021e-10
```

expm1()

Return $e^{\text{self}} - 1$, avoiding cancellation near 0.

EXAMPLES:

```
sage: r = 1.0
sage: r.expm1()
1.71828182845905
```

```
>>> from sage.all import *
>>> r = RealNumber('1.0')
>>> r.expm1()
1.71828182845905
```

```
sage: r = 1e-16
sage: exp(r)-1
0.0000000000000000
sage: r.expm1()
1.0000000000000000e-16
```

```
>>> from sage.all import *
>>> r = RealNumber('1e-16')
>>> exp(r)-Integer(1)
0.0000000000000000
>>> r.expm1()
1.0000000000000000e-16
```

floor()

Return the floor of `self`.

EXAMPLES:

```
sage: R = RealField()
sage: (2.99).floor()
2
sage: (2.00).floor()
2
sage: floor(RR(-5/2))
-3
sage: floor(RR(+infinity))
Traceback (most recent call last):
...
ValueError: Calling floor() on infinity or NaN
```

```

>>> from sage.all import *
>>> R = RealField()
>>> (RealNumber('2.99')).floor()
2
>>> (RealNumber('2.00')).floor()
2
>>> floor(RR(-Integer(5)/Integer(2)))
-3
>>> floor(RR(+infinity))
Traceback (most recent call last):
...
ValueError: Calling floor() on infinity or NaN

```

fp_rank()

Return the floating-point rank of this number. That is, if you list the floating-point numbers of this precision in order, and number them starting with $0.0 \rightarrow 0$ and extending the list to positive and negative infinity, returns the number corresponding to this floating-point number.

EXAMPLES:

```

sage: RR(0).fp_rank()
0
sage: RR(0).nextabove().fp_rank()
1
sage: RR(0).nextbelow().nextbelow().fp_rank()
-2
sage: RR(1).fp_rank()
4835703278458516698824705      # 32-bit
20769187434139310514121985316880385 # 64-bit
sage: RR(-1).fp_rank()
-4835703278458516698824705      # 32-bit
-20769187434139310514121985316880385 # 64-bit
sage: RR(1).fp_rank() - RR(1).nextbelow().fp_rank()
1
sage: RR(-infinity).fp_rank()
-9671406552413433770278913      # 32-bit
-41538374868278621023740371006390273 # 64-bit
sage: RR(-infinity).fp_rank() - RR(-infinity).nextabove().fp_rank()
-1

```

```

>>> from sage.all import *
>>> RR(Integer(0)).fp_rank()
0
>>> RR(Integer(0)).nextabove().fp_rank()
1
>>> RR(Integer(0)).nextbelow().nextbelow().fp_rank()
-2
>>> RR(Integer(1)).fp_rank()
4835703278458516698824705      # 32-bit
20769187434139310514121985316880385 # 64-bit
>>> RR(-Integer(1)).fp_rank()
-4835703278458516698824705      # 32-bit
-20769187434139310514121985316880385 # 64-bit

```

(continues on next page)

(continued from previous page)

```

>>> RR(Integer(1)).fp_rank() - RR(Integer(1)).nextbelow().fp_rank()
1
>>> RR(-infinity).fp_rank()
-9671406552413433770278913          # 32-bit
-41538374868278621023740371006390273 # 64-bit
>>> RR(-infinity).fp_rank() - RR(-infinity).nextabove().fp_rank()
-1

```

fp_rank_delta(other)

Return the floating-point rank delta between `self` and `other`. That is, if the return value is positive, this is the number of times you have to call `.nextabove()` to get from `self` to `other`.

EXAMPLES:

```

sage: [x.fp_rank_delta(x.nextabove()) for x in #_
↳needs sage.symbolic
.....: (RR(-infinity), -1.0, 0.0, 1.0, RR(pi), RR(infinity))]
[1, 1, 1, 1, 1, 0]

```

```

>>> from sage.all import *
>>> [x.fp_rank_delta(x.nextabove()) for x in #_
↳needs sage.symbolic
...   (RR(-infinity), -RealNumber('1.0'), RealNumber('0.0'), RealNumber('1.0
↳'), RR(pi), RR(infinity))]
[1, 1, 1, 1, 1, 0]

```

In the 2-bit floating-point field, one subsegment of the floating-point numbers is: 1, 1.5, 2, 3, 4, 6, 8, 12, 16, 24, 32

```

sage: R2 = RealField(2)
sage: R2(1).fp_rank_delta(R2(2))
2
sage: R2(2).fp_rank_delta(R2(1))
-2
sage: R2(1).fp_rank_delta(R2(1048576))
40
sage: R2(24).fp_rank_delta(R2(4))
-5
sage: R2(-4).fp_rank_delta(R2(-24))
-5

```

```

>>> from sage.all import *
>>> R2 = RealField(Integer(2))
>>> R2(Integer(1)).fp_rank_delta(R2(Integer(2)))
2
>>> R2(Integer(2)).fp_rank_delta(R2(Integer(1)))
-2
>>> R2(Integer(1)).fp_rank_delta(R2(Integer(1048576)))
40
>>> R2(Integer(24)).fp_rank_delta(R2(Integer(4)))
-5
>>> R2(-Integer(4)).fp_rank_delta(R2(-Integer(24)))

```

(continues on next page)

(continued from previous page)

-5

There are lots of floating-point numbers around 0:

```
sage: R2(-1).fp_rank_delta(R2(1))
4294967298      # 32-bit
18446744073709551618 # 64-bit
```

```
>>> from sage.all import *
>>> R2(-Integer(1)).fp_rank_delta(R2(Integer(1)))
4294967298      # 32-bit
18446744073709551618 # 64-bit
```

frac()

Return a real number such that $\text{self} = \text{self.trunc()} + \text{self.frac()}$. The return value will also satisfy $-1 < \text{self.frac()} < 1$.

EXAMPLES:

```
sage: (2.99).frac()
0.9900000000000000
sage: (2.50).frac()
0.5000000000000000
sage: (-2.79).frac()
-0.7900000000000000
sage: (-2.79).trunc() + (-2.79).frac()
-2.7900000000000000
```

```
>>> from sage.all import *
>>> (RealNumber('2.99')).frac()
0.9900000000000000
>>> (RealNumber('2.50')).frac()
0.5000000000000000
>>> (-RealNumber('2.79')).frac()
-0.7900000000000000
>>> (-RealNumber('2.79')).trunc() + (-RealNumber('2.79')).frac()
-2.7900000000000000
```

gamma()

Return the value of the Euler gamma function on *self*.

EXAMPLES:

```
sage: R = RealField()
sage: R(6).gamma()
120.00000000000000
sage: R(1.5).gamma()
0.886226925452758
```

```
>>> from sage.all import *
>>> R = RealField()
>>> R(Integer(6)).gamma()
```

(continues on next page)

(continued from previous page)

```
120.00000000000000
>>> R(RealNumber('1.5')).gamma()
0.886226925452758
```

hex()

Return a hexadecimal floating-point representation of `self`, in the style of C99 hexadecimal floating-point constants.

EXAMPLES:

```
sage: RR(-1/3).hex()
'-0x5.5555555555554p-4'
sage: Reals(100)(123.456e789).hex()
'0xf.721008e90630c8da88f44dd2p+2624'
sage: (-0.).hex()
'-0x0p+0'
```

```
>>> from sage.all import *
>>> RR(-Integer(1)/Integer(3)).hex()
'-0x5.5555555555554p-4'
>>> Reals(Integer(100))(RealNumber('123.456e789')).hex()
'0xf.721008e90630c8da88f44dd2p+2624'
>>> (-RealNumber('0.')).hex()
'-0x0p+0'
```

```
sage: [(a.hex(), float(a).hex()) for a in [.5, 1., 2., 16.]]
[('0x8p-4', '0x1.0000000000000p-1'),
 ('0x1p+0', '0x1.0000000000000p+0'),
 ('0x2p+0', '0x1.0000000000000p+1'),
 ('0x1p+4', '0x1.0000000000000p+4')]
```

```
>>> from sage.all import *
>>> [(a.hex(), float(a).hex()) for a in [RealNumber('.5'), RealNumber('1. '),
RealNumber('2. '), RealNumber('16. ')]]
[('0x8p-4', '0x1.0000000000000p-1'),
 ('0x1p+0', '0x1.0000000000000p+0'),
 ('0x2p+0', '0x1.0000000000000p+1'),
 ('0x1p+4', '0x1.0000000000000p+4')]
```

Special values:

```
sage: [RR(s).hex() for s in ['+inf', '-inf', 'nan']]
['inf', '-inf', 'nan']
```

```
>>> from sage.all import *
>>> [RR(s).hex() for s in ['+inf', '-inf', 'nan']]
['inf', '-inf', 'nan']
```

imag()

Return the imaginary part of `self`.

(Since `self` is a real number, this simply returns exactly 0.)

EXAMPLES:

```
sage: RR.pi().imag()
0
sage: RealField(100)(2).imag()
0
```

```
>>> from sage.all import *
>>> RR.pi().imag()
0
>>> RealField(Integer(100))(Integer(2)).imag()
0
```

integer_part()

If in decimal this number is written `n.defg`, returns `n`.

OUTPUT: a Sage Integer

EXAMPLES:

```
sage: a = 119.41212
sage: a.integer_part()
119
sage: a = -123.4567
sage: a.integer_part()
-123
```

```
>>> from sage.all import *
>>> a = RealNumber('119.41212')
>>> a.integer_part()
119
>>> a = -RealNumber('123.4567')
>>> a.integer_part()
-123
```

A big number with no decimal point:

```
sage: a = RR(10^17); a
1.000000000000000e17
sage: a.integer_part()
100000000000000000
```

```
>>> from sage.all import *
>>> a = RR(Integer(10)**Integer(17)); a
1.000000000000000e17
>>> a.integer_part()
100000000000000000
```

is_NaN()

Return True if self is Not-a-Number NaN.

EXAMPLES:

```
sage: a = RR(0) / RR(0); a
NaN
sage: a.is_NaN()
True
```

```
>>> from sage.all import *
>>> a = RR(Integer(0)) / RR(Integer(0)); a
NaN
>>> a.is_NaN()
True
```

is_infinity()

Return True if self is ∞ and False otherwise.

EXAMPLES:

```
sage: a = RR('1.494') / RR(0); a
+infinity
sage: a.is_infinity()
True
sage: a = -RR('1.494') / RR(0); a
-infinity
sage: a.is_infinity()
True
sage: RR(1.5).is_infinity()
False
sage: RR('nan').is_infinity()
False
```

```
>>> from sage.all import *
>>> a = RR('1.494') / RR(Integer(0)); a
+infinity
>>> a.is_infinity()
True
>>> a = -RR('1.494') / RR(Integer(0)); a
-infinity
>>> a.is_infinity()
True
>>> RR(RealNumber('1.5')).is_infinity()
False
>>> RR('nan').is_infinity()
False
```

is_integer()

Return True if this number is a integer.

EXAMPLES:

```
sage: RR(1).is_integer()
True
sage: RR(0.1).is_integer()
False
```

```
>>> from sage.all import *
>>> RR(Integer(1)).is_integer()
True
>>> RR(RealNumber('0.1')).is_integer()
False
```

is_negative_infinity()

Return True if self is $-\infty$.

EXAMPLES:

```
sage: a = RR('1.494') / RR(0); a
+infinity
sage: a.is_negative_infinity()
False
sage: a = -RR('1.494') / RR(0); a
-infinity
sage: RR(1.5).is_negative_infinity()
False
sage: a.is_negative_infinity()
True
```

```
>>> from sage.all import *
>>> a = RR('1.494') / RR(Integer(0)); a
+infinity
>>> a.is_negative_infinity()
False
>>> a = -RR('1.494') / RR(Integer(0)); a
-infinity
>>> RR(RealNumber('1.5')).is_negative_infinity()
False
>>> a.is_negative_infinity()
True
```

is_positive_infinity()

Return True if self is $+\infty$.

EXAMPLES:

```
sage: a = RR('1.494') / RR(0); a
+infinity
sage: a.is_positive_infinity()
True
sage: a = -RR('1.494') / RR(0); a
-infinity
sage: RR(1.5).is_positive_infinity()
False
sage: a.is_positive_infinity()
False
```

```
>>> from sage.all import *
>>> a = RR('1.494') / RR(Integer(0)); a
+infinity
```

(continues on next page)

(continued from previous page)

```

>>> a.is_positive_infinity()
True
>>> a = -RR('1.494') / RR(Integer(0)); a
-infinity
>>> RR(RealNumber('1.5')).is_positive_infinity()
False
>>> a.is_positive_infinity()
False

```

is_real()

Return True if self is real (of course, this always returns True for a finite element of a real field).

EXAMPLES:

```

sage: RR(1).is_real()
True
sage: RR('-100').is_real()
True
sage: RR(NaN).is_real()
↪needs sage.symbolic
False

```

#

```

>>> from sage.all import *
>>> RR(Integer(1)).is_real()
True
>>> RR('-100').is_real()
True
>>> RR(NaN).is_real()
↪needs sage.symbolic
False

```

#

is_square()

Return whether or not this number is a square in this field. For the real numbers, this is True if and only if self is nonnegative.

EXAMPLES:

```

sage: r = 3.5
sage: r.is_square()
True
sage: r = 0.0
sage: r.is_square()
True
sage: r = -4.0
sage: r.is_square()
False

```

```

>>> from sage.all import *
>>> r = RealNumber('3.5')
>>> r.is_square()
True
>>> r = RealNumber('0.0')

```

(continues on next page)

(continued from previous page)

```
>>> r.is_square()
True
>>> r = -RealNumber('4.0')
>>> r.is_square()
False
```

is_unit()

Return True if `self` is a unit (has a multiplicative inverse) and False otherwise.

EXAMPLES:

```
sage: RR(1).is_unit()
True
sage: RR('0').is_unit()
False
sage: RR('-0').is_unit()
False
sage: RR('nan').is_unit()
False
sage: RR('inf').is_unit()
False
sage: RR('-inf').is_unit()
False
```

```
>>> from sage.all import *
>>> RR(Integer(1)).is_unit()
True
>>> RR('0').is_unit()
False
>>> RR('-0').is_unit()
False
>>> RR('nan').is_unit()
False
>>> RR('inf').is_unit()
False
>>> RR('-inf').is_unit()
False
```

j0()

Return the value of the Bessel J function of order 0 at `self`.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).j0()
0.223890779141236
```

```
>>> from sage.all import *
>>> R = RealField(Integer(53))
>>> R(Integer(2)).j0()
0.223890779141236
```

j1()

Return the value of the Bessel J function of order 1 at `self`.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).j1()
0.576724807756873
```

```
>>> from sage.all import *
>>> R = RealField(Integer(53))
>>> R(Integer(2)).j1()
0.576724807756873
```

`jn( $n$ )`

Return the value of the Bessel J function of order n at `self`.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).jn(3)
0.128943249474402
sage: R(2).jn(-17)
-2.65930780516787e-15
```

```
>>> from sage.all import *
>>> R = RealField(Integer(53))
>>> R(Integer(2)).jn(Integer(3))
0.128943249474402
>>> R(Integer(2)).jn(-Integer(17))
-2.65930780516787e-15
```

`log( $base=None$ )`

Return the logarithm of `self` to the `base`.

EXAMPLES:

```
sage: R = RealField()
sage: R(2).log()
0.693147180559945
sage: log(RR(2))
0.693147180559945
sage: log(RR(2), "e")
0.693147180559945
sage: log(RR(2), e)
0.693147180559945
↪needs sage.symbolic
```

#_

```
>>> from sage.all import *
>>> R = RealField()
>>> R(Integer(2)).log()
0.693147180559945
>>> log(RR(Integer(2)))
0.693147180559945
```

(continues on next page)

(continued from previous page)

```
>>> log(RR(Integer(2)), "e")
0.693147180559945
>>> log(RR(Integer(2)), e)
↳ # needs sage.symbolic
0.693147180559945
```

```
sage: r = R(-1); r.log()
3.14159265358979*I
sage: log(RR(-1), e)
↳ needs sage.symbolic
3.14159265358979*I
sage: r.log(2)
4.53236014182719*I
```

```
>>> from sage.all import *
>>> r = R(-Integer(1)); r.log()
3.14159265358979*I
>>> log(RR(-Integer(1)), e)
↳ # needs sage.symbolic
3.14159265358979*I
>>> r.log(Integer(2))
4.53236014182719*I
```

For the error value NaN (Not A Number), log will return NaN:

```
sage: r = R(NaN); r.log()
↳ needs sage.symbolic
NaN
```

```
>>> from sage.all import *
>>> r = R(NaN); r.log()
↳ needs sage.symbolic
NaN
```

log10()

Return log to the base 10 of self.

EXAMPLES:

```
sage: r = 16.0; r.log10()
1.20411998265592
sage: r.log() / log(10.0)
1.20411998265592
```

```
>>> from sage.all import *
>>> r = RealNumber('16.0'); r.log10()
1.20411998265592
>>> r.log() / log(RealNumber('10.0'))
1.20411998265592
```

```
sage: r = 39.9; r.log10()
1.60097289568675
```

```
>>> from sage.all import *
>>> r = RealNumber('39.9'); r.log10()
1.60097289568675
```

```
sage: r = 0.0
sage: r.log10()
-infinity
```

```
>>> from sage.all import *
>>> r = RealNumber('0.0')
>>> r.log10()
-infinity
```

```
sage: r = -1.0
sage: r.log10()
1.36437635384184*I
```

```
>>> from sage.all import *
>>> r = -RealNumber('1.0')
>>> r.log10()
1.36437635384184*I
```

log1p()

Return log base e of $1 + \text{self}$.

EXAMPLES:

```
sage: r = 15.0; r.log1p()
2.77258872223978
sage: (r+1).log()
2.77258872223978
```

```
>>> from sage.all import *
>>> r = RealNumber('15.0'); r.log1p()
2.77258872223978
>>> (r+Integer(1)).log()
2.77258872223978
```

For small values, this is more accurate than computing $\log(1 + \text{self})$ directly, as it avoids cancellation issues:

```
sage: r = 3e-10
sage: r.log1p()
2.9999999955000e-10
sage: (1+r).log()
3.00000024777111e-10
sage: r100 = RealField(100)(r)
sage: (1+r100).log()
2.999999995500000000978021372e-10
```

```
>>> from sage.all import *
>>> r = RealNumber('3e-10')
```

(continues on next page)

(continued from previous page)

```
>>> r.log1p()
2.99999999955000e-10
>>> (Integer(1)+r).log()
3.00000024777111e-10
>>> r100 = RealField(Integer(100))(r)
>>> (Integer(1)+r100).log()
2.9999999995500000000978021372e-10
```

```
sage: r = 38.9; r.log1p()
3.68637632389582
```

```
>>> from sage.all import *
>>> r = RealNumber('38.9'); r.log1p()
3.68637632389582
```

```
sage: r = -1.0
sage: r.log1p()
-infinity
```

```
>>> from sage.all import *
>>> r = -RealNumber('1.0')
>>> r.log1p()
-infinity
```

```
sage: r = -2.0
sage: r.log1p()
3.14159265358979*I
```

```
>>> from sage.all import *
>>> r = -RealNumber('2.0')
>>> r.log1p()
3.14159265358979*I
```

log2()

Return log to the base 2 of `self`.

EXAMPLES:

```
sage: r = 16.0
sage: r.log2()
4.000000000000000
```

```
>>> from sage.all import *
>>> r = RealNumber('16.0')
>>> r.log2()
4.000000000000000
```

```
sage: r = 31.9; r.log2()
4.99548451887751
```

```
>>> from sage.all import *
>>> r = RealNumber('31.9'); r.log2()
4.99548451887751
```

```
sage: r = 0.0
sage: r.log2()
-infinity
```

```
>>> from sage.all import *
>>> r = RealNumber('0.0')
>>> r.log2()
-infinity
```

```
sage: r = -3.0; r.log2()
1.58496250072116 + 4.53236014182719*I
```

```
>>> from sage.all import *
>>> r = -RealNumber('3.0'); r.log2()
1.58496250072116 + 4.53236014182719*I
```

`log_gamma()`

Return the principal branch of the log gamma of `self`.

Note that this is not in general equal to `log(gamma(self))` for negative input.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(6).log_gamma()
4.78749174278205
sage: R(1e10).log_gamma()
2.20258509288811e11
sage: log_gamma(-2.1)
1.53171380819509 - 9.42477796076938*I
sage: log(gamma(-1.1)) == log_gamma(-1.1)
False
```

```
>>> from sage.all import *
>>> R = RealField(Integer(53))
>>> R(Integer(6)).log_gamma()
4.78749174278205
>>> R(RealNumber('1e10')).log_gamma()
2.20258509288811e11
>>> log_gamma(-RealNumber('2.1'))
1.53171380819509 - 9.42477796076938*I
>>> log(gamma(-RealNumber('1.1'))) == log_gamma(-RealNumber('1.1'))
False
```

`multiplicative_order()`

Return the multiplicative order of `self`.

EXAMPLES:

```
sage: RR(1).multiplicative_order()
1
sage: RR(-1).multiplicative_order()
2
sage: RR(3).multiplicative_order()
+Infinity
```

```
>>> from sage.all import *
>>> RR(Integer(1)).multiplicative_order()
1
>>> RR(-Integer(1)).multiplicative_order()
2
>>> RR(Integer(3)).multiplicative_order()
+Infinity
```

nearby_rational (*max_error=None, max_denominator=None*)

Find a rational near to *self*. Exactly one of *max_error* or *max_denominator* must be specified.

If *max_error* is specified, then this returns the simplest rational in the range $[\text{self}-\text{max_error} \dots \text{self}+\text{max_error}]$. If *max_denominator* is specified, then this returns the rational closest to *self* with denominator at most *max_denominator*. (In case of ties, we pick the simpler rational.)

EXAMPLES:

```
sage: (0.333).nearby_rational(max_error=0.001)
1/3
sage: (0.333).nearby_rational(max_error=1)
0
sage: (-0.333).nearby_rational(max_error=0.0001)
-257/772
```

```
>>> from sage.all import *
>>> (RealNumber('0.333')).nearby_rational(max_error=RealNumber('0.001'))
1/3
>>> (RealNumber('0.333')).nearby_rational(max_error=Integer(1))
0
>>> (-RealNumber('0.333')).nearby_rational(max_error=RealNumber('0.0001'))
-257/772
```

```
sage: (0.333).nearby_rational(max_denominator=100)
1/3
sage: RR(1/3 + 1/1000000).nearby_rational(max_denominator=2999999)
777780/2333333
sage: RR(1/3 + 1/1000000).nearby_rational(max_denominator=3000000)
1000003/3000000
sage: (-0.333).nearby_rational(max_denominator=1000)
-333/1000
sage: RR(3/4).nearby_rational(max_denominator=2)
1
sage: # needs sage.symbolic
sage: RR(pi).nearby_rational(max_denominator=120)
355/113
```

(continues on next page)

(continued from previous page)

```

sage: RR(pi).nearby_rational(max_denominator=10000)
355/113
sage: RR(pi).nearby_rational(max_denominator=100000)
312689/99532
sage: RR(pi).nearby_rational(max_denominator=1)
3
sage: RR(-3.5).nearby_rational(max_denominator=1)
-3

```

```

>>> from sage.all import *
>>> (RealNumber('0.333')).nearby_rational(max_denominator=Integer(100))
1/3
>>> RR(Integer(1)/Integer(3) + Integer(1)/Integer(1000000)).nearby_
↪rational(max_denominator=Integer(2999999))
777780/2333333
>>> RR(Integer(1)/Integer(3) + Integer(1)/Integer(1000000)).nearby_
↪rational(max_denominator=Integer(3000000))
1000003/3000000
>>> (-RealNumber('0.333')).nearby_rational(max_denominator=Integer(1000))
-333/1000
>>> RR(Integer(3)/Integer(4)).nearby_rational(max_denominator=Integer(2))
1

>>> # needs sage.symbolic
>>> RR(pi).nearby_rational(max_denominator=Integer(120))
355/113
>>> RR(pi).nearby_rational(max_denominator=Integer(10000))
355/113
>>> RR(pi).nearby_rational(max_denominator=Integer(100000))
312689/99532
>>> RR(pi).nearby_rational(max_denominator=Integer(1))
3

>>> RR(-RealNumber('3.5')).nearby_rational(max_denominator=Integer(1))
-3

```

nextabove()Return the next floating-point number larger than *self*.

EXAMPLES:

```

sage: RR('-infinity').nextabove()
-2.09857871646739e323228496      # 32-bit
-5.87565378911159e1388255822130839282 # 64-bit
sage: RR(0).nextabove()
2.38256490488795e-323228497      # 32-bit
8.50969131174084e-1388255822130839284 # 64-bit
sage: RR('+infinity').nextabove()
+infinity
sage: RR(-sqrt(2)).str()
↪needs sage.symbolic

```

(continues on next page)

(continued from previous page)

```
'-1.4142135623730951'
sage: RR(-sqrt(2)).nextabove().str() #_
↪needs sage.symbolic
'-1.4142135623730949'
```

```
>>> from sage.all import *
>>> RR('-infinity').nextabove()
-2.09857871646739e323228496 # 32-bit
-5.87565378911159e1388255822130839282 # 64-bit
>>> RR(Integer(0)).nextabove()
2.38256490488795e-323228497 # 32-bit
8.50969131174084e-1388255822130839284 # 64-bit
>>> RR('+infinity').nextabove()
+infinity
>>> RR(-sqrt(Integer(2))).str() #_
↪ # needs sage.symbolic
'-1.4142135623730951'
>>> RR(-sqrt(Integer(2))).nextabove().str() #_
↪ # needs sage.symbolic
'-1.4142135623730949'
```

nextbelow()

Return the next floating-point number smaller than self.

EXAMPLES:

```
sage: RR('-infinity').nextbelow()
-infinity
sage: RR(0).nextbelow()
-2.38256490488795e-323228497 # 32-bit
-8.50969131174084e-1388255822130839284 # 64-bit
sage: RR('+infinity').nextbelow()
2.09857871646739e323228496 # 32-bit
5.87565378911159e1388255822130839282 # 64-bit
sage: RR(-sqrt(2)).str() #_
↪needs sage.symbolic
'-1.4142135623730951'
sage: RR(-sqrt(2)).nextbelow().str() #_
↪needs sage.symbolic
'-1.4142135623730954'
```

```
>>> from sage.all import *
>>> RR('-infinity').nextbelow()
-infinity
>>> RR(Integer(0)).nextbelow()
-2.38256490488795e-323228497 # 32-bit
-8.50969131174084e-1388255822130839284 # 64-bit
>>> RR('+infinity').nextbelow()
2.09857871646739e323228496 # 32-bit
5.87565378911159e1388255822130839282 # 64-bit
>>> RR(-sqrt(Integer(2))).str() #_
↪ # needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
'-1.4142135623730951'
>>> RR(-sqrt(Integer(2))).nextbelow().str()
↪      # needs sage.symbolic
'-1.4142135623730954'
```

nexttoward (*other*)

Return the floating-point number adjacent to *self* which is closer to *other*. If *self* or *other* is NaN, returns NaN; if *self* equals *other*, returns *self*.

EXAMPLES:

```
sage: (1.0).nexttoward(2).str()
'1.0000000000000002'
sage: (1.0).nexttoward(RR('-infinity')).str()
'0.9999999999999999'
sage: RR(infinity).nexttoward(0)
2.09857871646739e323228496      # 32-bit
5.87565378911159e1388255822130839282 # 64-bit
sage: RR(pi).str()
↪needs sage.symbolic
'3.1415926535897931'
sage: RR(pi).nexttoward(22/7).str()
↪needs sage.symbolic
'3.1415926535897936'
sage: RR(pi).nexttoward(21/7).str()
↪needs sage.symbolic
'3.1415926535897927'
```

```
>>> from sage.all import *
>>> (RealNumber('1.0')).nexttoward(Integer(2)).str()
'1.0000000000000002'
>>> (RealNumber('1.0')).nexttoward(RR('-infinity')).str()
'0.9999999999999999'
>>> RR(infinity).nexttoward(Integer(0))
2.09857871646739e323228496      # 32-bit
5.87565378911159e1388255822130839282 # 64-bit
>>> RR(pi).str()
↪needs sage.symbolic
'3.1415926535897931'
>>> RR(pi).nexttoward(Integer(22)/Integer(7)).str()
↪      # needs sage.symbolic
'3.1415926535897936'
>>> RR(pi).nexttoward(Integer(21)/Integer(7)).str()
↪      # needs sage.symbolic
'3.1415926535897927'
```

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

Return an *n*-th root of *self*.

INPUT:

- *n* – a positive number, rounded down to the nearest integer; note that *n* should be less than `sys.maxsize`
- *algorithm* – set this to 1 to call mpfr directly, set this to 2 to use interval arithmetic and logarithms,

or leave it at the default of 0 to choose the algorithm which is estimated to be faster

AUTHORS:

- Carl Witty (2007-10)

EXAMPLES:

```
sage: R = RealField()
sage: R(8).nth_root(3)
2.000000000000000
sage: R(8).nth_root(3.7)      # illustrate rounding down
2.000000000000000
sage: R(-8).nth_root(3)
-2.000000000000000
sage: R(0).nth_root(3)
0.000000000000000
sage: R(32).nth_root(-1)
Traceback (most recent call last):
...
ValueError: n must be positive
sage: R(32).nth_root(1.0)
32.000000000000000
sage: R(4).nth_root(4)
1.41421356237310
sage: R(4).nth_root(40)
1.03526492384138
sage: R(4).nth_root(400)
1.00347174850950
sage: R(4).nth_root(4000)
1.00034663365385
sage: R(4).nth_root(400000)
1.00000034657365
sage: R(-27).nth_root(3)
-3.000000000000000
sage: R(-4).nth_root(3999999)
-1.00000034657374
```

```
>>> from sage.all import *
>>> R = RealField()
>>> R(Integer(8)).nth_root(Integer(3))
2.000000000000000
>>> R(Integer(8)).nth_root(RealNumber('3.7'))      # illustrate rounding down
2.000000000000000
>>> R(-Integer(8)).nth_root(Integer(3))
-2.000000000000000
>>> R(Integer(0)).nth_root(Integer(3))
0.000000000000000
>>> R(Integer(32)).nth_root(-Integer(1))
Traceback (most recent call last):
...
ValueError: n must be positive
>>> R(Integer(32)).nth_root(RealNumber('1.0'))
32.000000000000000
>>> R(Integer(4)).nth_root(Integer(4))
```

(continues on next page)

(continued from previous page)

```

1.41421356237310
>>> R(Integer(4)).nth_root(Integer(40))
1.03526492384138
>>> R(Integer(4)).nth_root(Integer(400))
1.00347174850950
>>> R(Integer(4)).nth_root(Integer(4000))
1.00034663365385
>>> R(Integer(4)).nth_root(Integer(400000))
1.00000034657365
>>> R(-Integer(27)).nth_root(Integer(3))
-3.000000000000000
>>> R(-Integer(4)).nth_root(Integer(3999999))
-1.00000034657374

```

Note that for negative numbers, any even root throws an exception:

```

sage: R(-2).nth_root(6)
Traceback (most recent call last):
...
ValueError: taking an even root of a negative number

```

```

>>> from sage.all import *
>>> R(-Integer(2)).nth_root(Integer(6))
Traceback (most recent call last):
...
ValueError: taking an even root of a negative number

```

The n -th root of 0 is defined to be 0, for any n :

```

sage: R(0).nth_root(6)
0.000000000000000
sage: R(0).nth_root(7)
0.000000000000000

```

```

>>> from sage.all import *
>>> R(Integer(0)).nth_root(Integer(6))
0.000000000000000
>>> R(Integer(0)).nth_root(Integer(7))
0.000000000000000

```

prec()

Return the precision of `self`.

EXAMPLES:

```

sage: RR(1.0).precision()
53
sage: RealField(101)(-1).precision()
101

```

```

>>> from sage.all import *
>>> RR(RealNumber('1.0')).precision()

```

(continues on next page)

(continued from previous page)

```
53
>>> RealField(Integer(101))(-Integer(1)).precision()
101
```

```
precision()
```

Return the precision of `self`.

EXAMPLES:

```
sage: RR(1.0).precision()
53
sage: RealField(101)(-1).precision()
101
```

```
>>> from sage.all import *
>>> RR(RealNumber('1.0')).precision()
53
>>> RealField(Integer(101))(-Integer(1)).precision()
101
```

real()

Return the real part of `self`.

(Since `self` is a real number, this simply returns `self`.)

EXAMPLES:

[illegible][illegible]

round()

Round `self` to the nearest representable integer, rounding halfway cases away from zero.

Note

The rounding mode of the parent field does not affect the result.

EXAMPLES:

```
sage: RR(0.49).round()
0
sage: RR(0.5).round()
1
sage: RR(-0.49).round()
```

(continues on next page)

(continued from previous page)

```
0
sage: RR(-0.5).round()
-1
```

```
>>> from sage.all import *
>>> RR(RealNumber('0.49')).round()
0
>>> RR(RealNumber('0.5')).round()
1
>>> RR(-RealNumber('0.49')).round()
0
>>> RR(-RealNumber('0.5')).round()
-1
```

sec()

Return the secant of this number.

EXAMPLES:

```
sage: RealField(100)(2).sec()
-2.4029979617223809897546004014
```

```
>>> from sage.all import *
>>> RealField(Integer(100))(Integer(2)).sec()
-2.4029979617223809897546004014
```

sech()

Return the hyperbolic secant of `self`.

EXAMPLES:

```
sage: RealField(100)(2).sech()
0.26580222883407969212086273982
```

```
>>> from sage.all import *
>>> RealField(Integer(100))(Integer(2)).sech()
0.26580222883407969212086273982
```

sign()

Return +1 if `self` is positive, -1 if `self` is negative, and 0 if `self` is zero.

EXAMPLES:

```
sage: R=RealField(100)
sage: R(-2.4).sign()
-1
sage: R(2.4).sign()
1
sage: R(0).sign()
0
```

```

>>> from sage.all import *
>>> R=RealField(Integer(100))
>>> R(-RealNumber('2.4')).sign()
-1
>>> R(RealNumber('2.1')).sign()
1
>>> R(Integer(0)).sign()
0

```

`sign_mantissa_exponent()`

Return the sign, mantissa, and exponent of `self`.

In Sage (as in MPFR), floating-point numbers of precision p are of the form $sm2^{e-p}$, where $s \in \{-1, 1\}$, $2^{p-1} \leq m < 2^p$, and $-2^{30} + 1 \leq e \leq 2^{30} - 1$; plus the special values $+0$, -0 , $+\infty$, $-\infty$, and NaN (which stands for Not-a-Number).

This function returns s , m , and $e - p$. For the special values:

- $+0$ returns $(1, 0, 0)$ (analogous to IEEE-754; note that MPFR actually stores the exponent as “smallest exponent possible”)
- -0 returns $(-1, 0, 0)$ (analogous to IEEE-754; note that MPFR actually stores the exponent as “smallest exponent possible”)
- the return values for $+\infty$, $-\infty$, and NaN are not specified.

EXAMPLES:

```

sage: R = RealField(53)
sage: a = R(exp(1.0)); a
2.71828182845905
sage: sign, mantissa, exponent = R(exp(1.0)).sign_mantissa_exponent()
sage: sign, mantissa, exponent
(1, 6121026514868073, -51)
sage: sign*mantissa*(2**exponent) == a
True

```

```

>>> from sage.all import *
>>> R = RealField(Integer(53))
>>> a = R(exp(RealNumber('1.0'))); a
2.71828182845905
>>> sign, mantissa, exponent = R(exp(RealNumber('1.0'))).sign_mantissa_
↳ exponent()
>>> sign, mantissa, exponent
(1, 6121026514868073, -51)
>>> sign*mantissa*(Integer(2)**exponent) == a
True

```

The mantissa is always a nonnegative number (see [Issue #14448](#)):

```

sage: RR(-1).sign_mantissa_exponent()
(-1, 4503599627370496, -52)

```

```

>>> from sage.all import *
>>> RR(-Integer(1)).sign_mantissa_exponent()
(-1, 4503599627370496, -52)

```


We can also calculate this also using p -adic valuations:

```
sage: a = R(exp(1.0))
sage: b = a.exact_rational()
sage: valuation, unit = b.val_unit(2)
sage: (b/abs(b), unit, valuation)
(1, 6121026514868073, -51)
sage: a.sign_mantissa_exponent()
(1, 6121026514868073, -51)
```

```
>>> from sage.all import *
>>> a = R(exp(RealNumber('1.0')))
>>> b = a.exact_rational()
>>> valuation, unit = b.val_unit(Integer(2))
>>> (b/abs(b), unit, valuation)
(1, 6121026514868073, -51)
>>> a.sign_mantissa_exponent()
(1, 6121026514868073, -51)
```

`simplest_rational()`

Return the simplest rational which is equal to `self` (in the Sage sense). Recall that Sage defines the equality operator by coercing both sides to a single type and then comparing; thus, this finds the simplest rational which (when coerced to this RealField) is equal to `self`.

Given rationals a/b and c/d (both in lowest terms), the former is simpler if $b < d$ or if $b = d$ and $|a| < |c|$.

The effect of rounding modes is slightly counter-intuitive. Consider the case of round-toward-minus-infinity. This rounding is performed when coercing a rational to a floating-point number; so the `simplest_rational()` of a round-to-minus-infinity number will be either exactly equal to or slightly larger than the number.

EXAMPLES:

```
sage: RRd = RealField(53, rnd='RNDD')
sage: RRz = RealField(53, rnd='RNDZ')
sage: RRu = RealField(53, rnd='RNDU')
sage: RRa = RealField(53, rnd='RNDA')
sage: def check(x):
....:     rx = x.simplest_rational()
....:     assert x == rx
....:     return rx
sage: RRd(1/3) < RRu(1/3)
True
sage: check(RRd(1/3))
1/3
sage: check(RRu(1/3))
1/3
sage: check(RRz(1/3))
1/3
sage: check(RRa(1/3))
1/3
sage: check(RR(1/3))
1/3
sage: check(RRd(-1/3))
-1/3
sage: check(RRu(-1/3))
```

(continues on next page)

(continued from previous page)

```

-1/3
sage: check(RRz(-1/3))
-1/3
sage: check(RRa(-1/3))
-1/3
sage: check(RR(-1/3))
-1/3
sage: check(RealField(20)(pi)) #_
↪needs sage.symbolic
355/113
sage: check(RR(pi)) #_
↪needs sage.symbolic
245850922/78256779
sage: check(RR(2).sqrt())
131836323/93222358
sage: check(RR(1/2^210))
1/1645504557321205859467264516194506011931735427766374553794641921
sage: check(RR(2^210))
1645504557321205950811116849375918117252433820865891134852825088
sage: (RR(17).sqrt()).simplest_rational()^2 - 17
-1/348729667233025
sage: (RR(23).cube_root()).simplest_rational()^3 - 23
-1404915133/264743395842039084891584
sage: RRd5 = RealField(5, rnd='RNDD')
sage: RRu5 = RealField(5, rnd='RNDU')
sage: RR5 = RealField(5)
sage: below1 = RR5(1).nextbelow()
sage: check(RRd5(below1))
31/32
sage: check(RRu5(below1))
16/17
sage: check(below1)
21/22
sage: below1.exact_rational()
31/32
sage: above1 = RR5(1).nextabove()
sage: check(RRd5(above1))
10/9
sage: check(RRu5(above1))
17/16
sage: check(above1)
12/11
sage: above1.exact_rational()
17/16
sage: check(RR(1234))
1234
sage: check(RR5(1234))
1185
sage: check(RR5(1184))
1120
sage: RRd2 = RealField(2, rnd='RNDD')
sage: RRu2 = RealField(2, rnd='RNDU')

```

(continues on next page)

(continued from previous page)

```

sage: RR2 = RealField(2)
sage: check(RR2(8))
7
sage: check(RRd2(8))
8
sage: check(RRu2(8))
7
sage: check(RR2(13))
11
sage: check(RRd2(13))
12
sage: check(RRu2(13))
13
sage: check(RR2(16))
14
sage: check(RRd2(16))
16
sage: check(RRu2(16))
13
sage: check(RR2(24))
21
sage: check(RRu2(24))
17
sage: check(RR2(-24))
-21
sage: check(RRu2(-24))
-24

```

```

>>> from sage.all import *
>>> RRd = RealField(Integer(53), rnd='RNDD')
>>> RRz = RealField(Integer(53), rnd='RNDZ')
>>> RRu = RealField(Integer(53), rnd='RNDU')
>>> RRa = RealField(Integer(53), rnd='RNDA')
>>> def check(x):
...     rx = x.simplest_rational()
...     assert x == rx
...     return rx
>>> RRd(Integer(1)/Integer(3)) < RRu(Integer(1)/Integer(3))
True
>>> check(RRd(Integer(1)/Integer(3)))
1/3
>>> check(RRu(Integer(1)/Integer(3)))
1/3
>>> check(RRz(Integer(1)/Integer(3)))
1/3
>>> check(RRa(Integer(1)/Integer(3)))
1/3
>>> check(RR(Integer(1)/Integer(3)))
1/3
>>> check(RRd(-Integer(1)/Integer(3)))
-1/3
>>> check(RRu(-Integer(1)/Integer(3)))

```

(continues on next page)

(continued from previous page)

```

-1/3
>>> check(RRz(-Integer(1)/Integer(3)))
-1/3
>>> check(RRa(-Integer(1)/Integer(3)))
-1/3
>>> check(RR(-Integer(1)/Integer(3)))
-1/3
>>> check(RealField(Integer(20))(pi))
↳ # needs sage.symbolic
355/113
>>> check(RR(pi))
↳ needs sage.symbolic #
245850922/78256779
>>> check(RR(Integer(2)).sqrt())
131836323/93222358
>>> check(RR(Integer(1)/Integer(2)**Integer(210)))
1/1645504557321205859467264516194506011931735427766374553794641921
>>> check(RR(Integer(2)**Integer(210)))
1645504557321205950811116849375918117252433820865891134852825088
>>> (RR(Integer(17)).sqrt()).simplest_rational()*Integer(2) - Integer(17)
-1/348729667233025
>>> (RR(Integer(23)).cube_root()).simplest_rational()*Integer(3) -
↳ Integer(23)
-1404915133/264743395842039084891584
>>> RRd5 = RealField(Integer(5), rnd='RNDD')
>>> RRu5 = RealField(Integer(5), rnd='RNDU')
>>> RR5 = RealField(Integer(5))
>>> below1 = RR5(Integer(1)).nextbelow()
>>> check(RRd5(below1))
31/32
>>> check(RRu5(below1))
16/17
>>> check(below1)
21/22
>>> below1.exact_rational()
31/32
>>> above1 = RR5(Integer(1)).nextabove()
>>> check(RRd5(above1))
10/9
>>> check(RRu5(above1))
17/16
>>> check(above1)
12/11
>>> above1.exact_rational()
17/16
>>> check(RR(Integer(1234)))
1234
>>> check(RR5(Integer(1234)))
1185
>>> check(RR5(Integer(1184)))
1120
>>> RRd2 = RealField(Integer(2), rnd='RNDD')

```

(continues on next page)

(continued from previous page)

```

>>> RRu2 = RealField(Integer(2), rnd='RNDU')
>>> RR2 = RealField(Integer(2))
>>> check(RR2(Integer(8)))
7
>>> check(RRd2(Integer(8)))
8
>>> check(RRu2(Integer(8)))
7
>>> check(RR2(Integer(13)))
11
>>> check(RRd2(Integer(13)))
12
>>> check(RRu2(Integer(13)))
13
>>> check(RR2(Integer(16)))
14
>>> check(RRd2(Integer(16)))
16
>>> check(RRu2(Integer(16)))
13
>>> check(RR2(Integer(24)))
21
>>> check(RRu2(Integer(24)))
17
>>> check(RR2(-Integer(24)))
-21
>>> check(RRu2(-Integer(24)))
-24

```

sin()Return the sine of `self`.

EXAMPLES:

```

sage: R = RealField(100)
sage: R(2).sin()
0.90929742682568169539601986591

```

```

>>> from sage.all import *
>>> R = RealField(Integer(100))
>>> R(Integer(2)).sin()
0.90929742682568169539601986591

```

sincos()Return a pair consisting of the sine and cosine of `self`.

EXAMPLES:

```

sage: R = RealField()
sage: t = R.pi()/6
sage: t.sincos()
(0.5000000000000000, 0.866025403784439)

```

```
>>> from sage.all import *
>>> R = RealField()
>>> t = R.pi()/Integer(6)
>>> t.sincos()
(0.5000000000000000, 0.866025403784439)
```

sinh()

Return the hyperbolic sine of `self`.

EXAMPLES:

```
sage: q = RR.pi()/12
sage: q.sinh()
0.264800227602271
```

```
>>> from sage.all import *
>>> q = RR.pi()/Integer(12)
>>> q.sinh()
0.264800227602271
```

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

The square root function.

INPUT:

- `extend` – boolean (default: `True`); if `True`, return a square root in a complex field if necessary if `self` is negative. Otherwise raise a `ValueError`.
- `all` – boolean (default: `False`); if `True`, return a list of all square roots

EXAMPLES:

```
sage: r = -2.0
sage: r.sqrt()
1.41421356237310*I
```

```
>>> from sage.all import *
>>> r = -RealNumber('2.0')
>>> r.sqrt()
1.41421356237310*I
```

```
sage: r = 4.0
sage: r.sqrt()
2.0000000000000000
sage: r.sqrt()^2 == r
True
```

```
>>> from sage.all import *
>>> r = RealNumber('4.0')
>>> r.sqrt()
2.0000000000000000
>>> r.sqrt()**Integer(2) == r
True
```

```
sage: r = 4344
sage: r.sqrt()
↳needs sage.symbolic
2*sqrt(1086)
```

```
>>> from sage.all import *
>>> r = Integer(4344)
>>> r.sqrt()
↳needs sage.symbolic
2*sqrt(1086)
```

```
sage: r = 4344.0
sage: r.sqrt()^2 == r
True
sage: r.sqrt()^2 - r
0.0000000000000000
```

```
>>> from sage.all import *
>>> r = RealNumber('4344.0')
>>> r.sqrt()**Integer(2) == r
True
>>> r.sqrt()**Integer(2) - r
0.0000000000000000
```

```
sage: r = -2.0
sage: r.sqrt()
1.41421356237310*I
```

```
>>> from sage.all import *
>>> r = -RealNumber('2.0')
>>> r.sqrt()
1.41421356237310*I
```

str (*base=10, digits=0, no_sci=None, e=None, truncate=False, skip_zeroes=False*)

Return a string representation of `self`.

INPUT:

- `base` – (default: 10) base for output
- `digits` – (default: 0) number of digits to display; when `digits` is zero, choose this automatically
- `no_sci` – if 2, never print using scientific notation; if `True`, use scientific notation only for large or small numbers; if `False` always print with scientific notation; if `None` (the default), print how the parent prints.
- `e` – symbol used in scientific notation; defaults to 'e' for `base=10`, and '@' otherwise
- `truncate` – boolean (default: `False`); if `True`, round off the last digits in base-10 printing to lessen confusing base-2 roundoff issues. This flag may not be used in other bases or when `digits` is given.
- `skip_zeroes` – boolean (default: `False`); if `True`, skip trailing zeroes in mantissa

EXAMPLES:

[illegible]

```
>>> from sage.all import *
>>> a = Integer(61)/RealNumber('3.0'); a
20.33333333333333
>>> a.str()
```

(continues on next page)


```
'20.333333333333332'
>>> a.str(truncate=True)
'20.33333333333333'
>>> a.str(Integer(2))
'10100.01010101010101010101010101010101010101010101'
>>> a.str(no_sci=False)
'2.0333333333333332e1'
>>> a.str(Integer(16), no_sci=False)
'1.455555555555555@1'
>>> a.str(digits=Integer(5))
'20.333'
>>> a.str(Integer(2), digits=Integer(5))
'10100.'
>>> b = RealNumber('2.0')**Integer(99)
>>> b.str()
'6.3382530011411470e29'
>>> b.str(no_sci=False)
'6.3382530011411470e29'
>>> b.str(no_sci=True)
'6.3382530011411470e29'
>>> c = RealNumber('2.0')**Integer(100)
>>> c.str()
'1.2676506002282294e30'
>>> c.str(no_sci=False)
'1.2676506002282294e30'
>>> c.str(no_sci=True)
'1.2676506002282294e30'
>>> c.str(no_sci=Integer(2))
'12676506002282294000000000000000.'
>>> RealNumber('0.5')**Integer(53)
1.11022302462516e-16
>>> RealNumber('0.5')**Integer(54)
5.55111512312578e-17
>>> (RealNumber('0.01')).str()
'0.010000000000000000'
>>> (RealNumber('0.01')).str(skip_zeroes=True)
'0.01'
>>> (-RealNumber('10.042')).str()
'-10.042000000000000'
>>> (-RealNumber('10.042')).str(skip_zeroes=True)
'-10.042'
>>> (RealNumber('389.0')).str(skip_zeroes=True)
'389.'
```

```
sage: print((65536.0).str(base=2))  
1.0000000000000000000000000000000000000000000000000000000e16  
sage: print((65536.0).str(base=36))  
1ekg.00000000  
sage: print((65536.0).str(base=62))
```

1.1. Arbitrary precision floating point real numbers using GNU MPFR 61

(continued from previous page)

H32.0000000

```
>>> from sage.all import *
>>> print((RealNumber('65536.0')).str(base=Integer(2)))
1.0000000000000000000000000000000000000000000000000000000e16
>>> print((RealNumber('65536.0')).str(base=Integer(36)))
1ekg.00000000
>>> print((RealNumber('65536.0')).str(base=Integer(62)))
H32.0000000
```

String conversion respects rounding:

```
sage: x = -RR.pi()
sage: x.str(digits=1)
'-3.'
sage: y = RealField(53, rnd="RNDD")(x)
sage: y.str(digits=1)
'-4.'
sage: y = RealField(53, rnd="RNDU")(x)
sage: y.str(digits=1)
'-3.'
sage: y = RealField(53, rnd="RNDZ")(x)
sage: y.str(digits=1)
'-3.'
sage: y = RealField(53, rnd="RNDA")(x)
sage: y.str(digits=1)
'-4.'
```

```
>>> from sage.all import *
>>> x = -RR.pi()
>>> x.str(digits=Integer(1))
'-3.'
>>> y = RealField(Integer(53), rnd="RNDD")(x)
>>> y.str(digits=Integer(1))
'-4.'
>>> y = RealField(Integer(53), rnd="RNDU")(x)
>>> y.str(digits=Integer(1))
'-3.'
>>> y = RealField(Integer(53), rnd="RNDZ")(x)
>>> y.str(digits=Integer(1))
'-3.'
>>> y = RealField(Integer(53), rnd="RNDA")(x)
>>> y.str(digits=Integer(1))
'-4.'
```

Zero has the correct number of digits:

```
sage: zero = RR.zero()
sage: print(zero.str(digits=3))
0.00
sage: print(zero.str(digits=3, no_sci=False))
0.00e0
```

(continues on next page)

(continued from previous page)

```
sage: print(zero.str(digits=3, skip_zeroes=True))
0.
```

```
>>> from sage.all import *
>>> zero = RR.zero()
>>> print(zero.str(digits=Integer(3)))
0.00
>>> print(zero.str(digits=Integer(3), no_sci=False))
0.00e0
>>> print(zero.str(digits=Integer(3), skip_zeroes=True))
0.
```

The output always contains a decimal point, except when using scientific notation with exactly one digit:

```
sage: print((1e1).str(digits=1))
10.
sage: print((1e10).str(digits=1))
1e10
sage: print((1e-1).str(digits=1))
0.1
sage: print((1e-10).str(digits=1))
1e-10
sage: print((-1e1).str(digits=1))
-10.
sage: print((-1e10).str(digits=1))
-1e10
sage: print((-1e-1).str(digits=1))
-0.1
sage: print((-1e-10).str(digits=1))
-1e-10
```

```
>>> from sage.all import *
>>> print((RealNumber('1e1')).str(digits=Integer(1)))
10.
>>> print((RealNumber('1e10')).str(digits=Integer(1)))
1e10
>>> print((RealNumber('1e-1')).str(digits=Integer(1)))
0.1
>>> print((RealNumber('1e-10')).str(digits=Integer(1)))
1e-10
>>> print((-RealNumber('1e1')).str(digits=Integer(1)))
-10.
>>> print((-RealNumber('1e10')).str(digits=Integer(1)))
-1e10
>>> print((-RealNumber('1e-1')).str(digits=Integer(1)))
-0.1
>>> print((-RealNumber('1e-10')).str(digits=Integer(1)))
-1e-10
```

tan()

Return the tangent of self.

EXAMPLES:

```
sage: q = RR.pi()/3
sage: q.tan()
1.73205080756888
sage: q = RR.pi()/6
sage: q.tan()
0.577350269189626
```

```
>>> from sage.all import *
>>> q = RR.pi()/Integer(3)
>>> q.tan()
1.73205080756888
>>> q = RR.pi()/Integer(6)
>>> q.tan()
0.577350269189626
```

tanh()

Return the hyperbolic tangent of `self`.

EXAMPLES:

```
sage: q = RR.pi()/11
sage: q.tanh()
0.278079429295850
```

```
>>> from sage.all import *
>>> q = RR.pi()/Integer(11)
>>> q.tanh()
0.278079429295850
```

trunc()

Truncate `self`.

EXAMPLES:

```
sage: (2.99).trunc()
2
sage: (-0.00).trunc()
0
sage: (0.00).trunc()
0
```

```
>>> from sage.all import *
>>> (RealNumber('2.99')).trunc()
2
>>> (-RealNumber('0.00')).trunc()
0
>>> (RealNumber('0.00')).trunc()
0
```

ulp(*field=None*)

Return the unit of least precision of `self`, which is the weight of the least significant bit of `self`. This is always a strictly positive number. It is also the gap between this number and the closest number with larger absolute value that can be represented.

INPUT:

- `field` – *RealField* used as parent of the result; if not specified, use `parent(self)`

Note

The ulp of zero is defined as the smallest representable positive number. For extremely small numbers, underflow occurs and the output is also the smallest representable positive number (the rounding mode is ignored, this computation is done by rounding towards +infinity).

See also

`epsilon()` for a scale-invariant version of this.

EXAMPLES:

```
sage: a = 1.0
sage: a.ulp()
2.22044604925031e-16
sage: (-1.5).ulp()
2.22044604925031e-16
sage: a + a.ulp() == a
False
sage: a + a.ulp()/2 == a
True

sage: a = RealField(500).pi()
sage: b = a + a.ulp()
sage: (a+b)/2 in [a,b]
True
```

```
>>> from sage.all import *
>>> a = RealNumber('1.0')
>>> a.ulp()
2.22044604925031e-16
>>> (-RealNumber('1.5')).ulp()
2.22044604925031e-16
>>> a + a.ulp() == a
False
>>> a + a.ulp()/Integer(2) == a
True

>>> a = RealField(Integer(500)).pi()
>>> b = a + a.ulp()
>>> (a+b)/Integer(2) in [a,b]
True
```

The ulp of zero is the smallest nonzero number:

```
sage: a = RR(0).ulp()
sage: a
```

(continues on next page)

(continued from previous page)

```

2.38256490488795e-323228497          # 32-bit
8.50969131174084e-1388255822130839284 # 64-bit
sage: a.fp_rank()
1

```

```

>>> from sage.all import *
>>> a = RR(Integer(0)).ulp()
>>> a
2.38256490488795e-323228497          # 32-bit
8.50969131174084e-1388255822130839284 # 64-bit
>>> a.fp_rank()
1

```

The ulp of very small numbers results in underflow, so the smallest nonzero number is returned instead:

```

sage: a.ulp() == a
True

```

```

>>> from sage.all import *
>>> a.ulp() == a
True

```

We use a different field:

```

sage: a = RealField(256).pi()
sage: a.ulp()
3.
↪454467422037777850154540745120159828446400145774512554009481388067436721265e-
↪77
sage: e = a.ulp(RealField(64))
sage: e
3.45446742203777785e-77
sage: parent(e)
Real Field with 64 bits of precision
sage: e = a.ulp(QQ)
Traceback (most recent call last):
...
TypeError: field argument must be a RealField

```

```

>>> from sage.all import *
>>> a = RealField(Integer(256)).pi()
>>> a.ulp()
3.
↪454467422037777850154540745120159828446400145774512554009481388067436721265e-
↪77
>>> e = a.ulp(RealField(Integer(64)))
>>> e
3.45446742203777785e-77
>>> parent(e)
Real Field with 64 bits of precision
>>> e = a.ulp(QQ)
Traceback (most recent call last):

```

(continues on next page)

(continued from previous page)

```
...
TypeError: field argument must be a RealField
```

For infinity and NaN, we get back positive infinity and NaN:

```
sage: a = RR(infinity)
sage: a.ulp()
+infinity
sage: (-a).ulp()
+infinity
sage: a = RR('nan')
sage: a.ulp()
NaN
sage: parent(RR('nan').ulp(RealField(42)))
Real Field with 42 bits of precision
```

```
>>> from sage.all import *
>>> a = RR(infinity)
>>> a.ulp()
+infinity
>>> (-a).ulp()
+infinity
>>> a = RR('nan')
>>> a.ulp()
NaN
>>> parent(RR('nan').ulp(RealField(Integer(42))))
Real Field with 42 bits of precision
```

y0()

Return the value of the Bessel Y function of order 0 at `self`.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).y0()
0.510375672649745
```

```
>>> from sage.all import *
>>> R = RealField(Integer(53))
>>> R(Integer(2)).y0()
0.510375672649745
```

y1()

Return the value of the Bessel Y function of order 1 at `self`.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).y1()
-0.107032431540938
```

```
>>> from sage.all import *
>>> R = RealField(Integer(53))
>>> R(Integer(2)).y1()
-0.107032431540938
```

yn(*n*)

Return the value of the Bessel Y function of order n at `self`.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).yn(3)
-1.12778377684043
sage: R(2).yn(-17)
7.09038821729481e12
```

```
>>> from sage.all import *
>>> R = RealField(Integer(53))
>>> R(Integer(2)).yn(Integer(3))
-1.12778377684043
>>> R(Integer(2)).yn(-Integer(17))
7.09038821729481e12
```

zeta()

Return the Riemann zeta function evaluated at this real number.

Note

PARI is vastly more efficient at computing the Riemann zeta function. See the example below for how to use it.

EXAMPLES:

```
sage: R = RealField()
sage: R(2).zeta()
1.64493406684823
sage: R.pi()^2/6
1.64493406684823
sage: R(-2).zeta()
0.000000000000000
sage: R(1).zeta()
+infinity
```

```
>>> from sage.all import *
>>> R = RealField()
>>> R(Integer(2)).zeta()
1.64493406684823
>>> R.pi()**Integer(2)/Integer(6)
1.64493406684823
>>> R(-Integer(2)).zeta()
0.000000000000000
```

(continues on next page)

(continued from previous page)

```
>>> R(Integer(1)).zeta()
+infinity
```

Computing zeta using PARI is much more efficient in difficult cases. Here's how to compute zeta with at least a given precision:

```
sage: z = pari(2).zeta(precision=53); z #_
↳needs sage.libs.pari
1.64493406684823
sage: pari(2).zeta(precision=128).sage().prec() #_
↳needs sage.libs.pari
128
sage: pari(2).zeta(precision=65).sage().prec() #_
↳needs sage.libs.pari
128 # 64-bit
96 # 32-bit
```

```
>>> from sage.all import *
>>> z = pari(Integer(2)).zeta(precision=Integer(53)); z #_
↳ # needs sage.libs.pari
1.64493406684823
>>> pari(Integer(2)).zeta(precision=Integer(128)).sage().prec() #_
↳ # needs sage.libs.pari
128
>>> pari(Integer(2)).zeta(precision=Integer(65)).sage().prec() #_
↳ # needs sage.libs.pari
128 # 64-bit
96 # 32-bit
```

Note that the number of bits of precision in the constructor only effects the internal precision of the pari number, which is rounded up to the nearest multiple of 32 or 64. To increase the number of digits that gets displayed you must use `pari.set_real_precision`.

```
sage: type(z) #_
↳needs sage.libs.pari
<class 'cypari2.gen.Gen'>
sage: R(z) #_
↳needs sage.libs.pari
1.64493406684823
```

```
>>> from sage.all import *
>>> type(z) #_
↳needs sage.libs.pari
<class 'cypari2.gen.Gen'>
>>> R(z) #_
↳needs sage.libs.pari
1.64493406684823
```

```
class sage.rings.real_mpfr.ZZtoRR
```

```
Bases: Map
```

```
sage.rings.real_mpfr.create_RealNumber(s, base=10, pad=0, rnd='RNDN', min_prec=53)
```

Return the real number defined by the string `s` as an element of `RealField(prec=n)`, where `n` potentially has

INPUT:

- EXAMPLES:

[illegible]

```
sage: RealNumber("10101e2", base=2)
84.000000000000000
sage: RealNumber("deadbeef", base=16)
3.735928559000000e9
sage: RealNumber("deadbeefxxx", base=16)
Traceback (most recent call last):
...
TypeError: unable to convert 'deadbeefxxx' to a real number
sage: RealNumber("z", base=36)
35.000000000000000
```

(continued from previous page)

```

sage: RealNumber("AAA", base=37)
14070.000000000000
sage: RealNumber("aaa", base=37)
50652.000000000000
sage: RealNumber("3.4", base='foo')
Traceback (most recent call last):
...
TypeError: an integer is required
sage: RealNumber("3.4", base=63)
Traceback (most recent call last):
...
ValueError: base (=63) must be an integer between 2 and 62

```

```

>>> from sage.all import *
>>> RealNumber("10101e2", base=Integer(2))
84.000000000000000
>>> RealNumber("deadbeef", base=Integer(16))
3.735928559000000e9
>>> RealNumber("deadbeefxxx", base=Integer(16))
Traceback (most recent call last):
...
TypeError: unable to convert 'deadbeefxxx' to a real number
>>> RealNumber("z", base=Integer(36))
35.000000000000000
>>> RealNumber("AAA", base=Integer(37))
14070.000000000000
>>> RealNumber("aaa", base=Integer(37))
50652.000000000000
>>> RealNumber("3.4", base='foo')
Traceback (most recent call last):
...
TypeError: an integer is required
>>> RealNumber("3.4", base=Integer(63))
Traceback (most recent call last):
...
ValueError: base (=63) must be an integer between 2 and 62

```

The rounding mode is respected in all cases:

```

sage: RealNumber("1.5", rnd="RNDU").parent()
Real Field with 53 bits of precision and rounding RNDU
sage: RealNumber("1.5000000000000000000000000000000000000000000000000000000", rnd="RNDU").parent()
Real Field with 130 bits of precision and rounding RNDU

```

```

>>> from sage.all import *
>>> RealNumber("1.5", rnd="RNDU").parent()
Real Field with 53 bits of precision and rounding RNDU
>>> RealNumber("1.5000000000000000000000000000000000000000000000000000000", rnd="RNDU").parent()
Real Field with 130 bits of precision and rounding RNDU

```

```
class sage.rings.real_mpfr.double_toRR
```

Bases: Map

```
class sage.rings.real_mpfr.int_toRR
```

Bases: Map

```
sage.rings.real_mpfr.is_RealNumber(x)
```

Return True if x is of type *RealNumber*, meaning that it is an element of the MPFR real field with some precision.

EXAMPLES:

```
sage: from sage.rings.real_mpfr import is_RealNumber
sage: is_RealNumber(2.5)
doctest:warning...
DeprecationWarning: The function is_RealNumber is deprecated;
use 'isinstance(..., RealNumber)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
sage: is_RealNumber(float(2.3))
False
sage: is_RealNumber(RDF(2))
False
sage: is_RealNumber(pi)
↪needs sage.symbolic
False
```

```
>>> from sage.all import *
>>> from sage.rings.real_mpfr import is_RealNumber
>>> is_RealNumber(RealNumber('2.5'))
doctest:warning...
DeprecationWarning: The function is_RealNumber is deprecated;
use 'isinstance(..., RealNumber)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
>>> is_RealNumber(float(RealNumber('2.3')))
False
>>> is_RealNumber(RDF(Integer(2)))
False
>>> is_RealNumber(pi)
↪needs sage.symbolic
False
```

```
sage.rings.real_mpfr.mpfr_get_exp_max()
```

Return the current maximal exponent for MPFR numbers.

EXAMPLES:

```
sage: from sage.rings.real_mpfr import mpfr_get_exp_max
sage: mpfr_get_exp_max()
1073741823          # 32-bit
4611686018427387903 # 64-bit
sage: 0.5 << mpfr_get_exp_max()
1.04928935823369e323228496          # 32-bit
2.93782689455579e1388255822130839282 # 64-bit
sage: 0.5 << (mpfr_get_exp_max()+1)
+infinity
```

```

>>> from sage.all import *
>>> from sage.rings.real_mpfr import mpfr_get_exp_max
>>> mpfr_get_exp_max()
1073741823          # 32-bit
4611686018427387903 # 64-bit
>>> RealNumber('0.5') << mpfr_get_exp_max()
1.04928935823369e323228496          # 32-bit
2.93782689455579e1388255822130839282 # 64-bit
>>> RealNumber('0.5') << (mpfr_get_exp_max()+Integer(1))
+infinity

```

`sage.rings.real_mpfr.mpfr_get_exp_max_max()`

Get the maximal value allowed for `mpfr_set_exp_max()`.

EXAMPLES:

```

sage: from sage.rings.real_mpfr import mpfr_get_exp_max_max, mpfr_set_exp_max
sage: mpfr_get_exp_max_max()
1073741823          # 32-bit
4611686018427387903 # 64-bit

```

```

>>> from sage.all import *
>>> from sage.rings.real_mpfr import mpfr_get_exp_max_max, mpfr_set_exp_max
>>> mpfr_get_exp_max_max()
1073741823          # 32-bit
4611686018427387903 # 64-bit

```

This is really the maximal value allowed:

```

sage: mpfr_set_exp_max(mpfr_get_exp_max_max() + 1)
Traceback (most recent call last):
...
OverflowError: bad value for mpfr_set_exp_max()

```

```

>>> from sage.all import *
>>> mpfr_set_exp_max(mpfr_get_exp_max_max() + Integer(1))
Traceback (most recent call last):
...
OverflowError: bad value for mpfr_set_exp_max()

```

`sage.rings.real_mpfr.mpfr_get_exp_min()`

Return the current minimal exponent for MPFR numbers.

EXAMPLES:

```

sage: from sage.rings.real_mpfr import mpfr_get_exp_min
sage: mpfr_get_exp_min()
-1073741823          # 32-bit
-4611686018427387903 # 64-bit
sage: 0.5 >> (-mpfr_get_exp_min())
2.38256490488795e-323228497          # 32-bit
8.50969131174084e-1388255822130839284 # 64-bit
sage: 0.5 >> (-mpfr_get_exp_min()+1)
0.0000000000000000

```

```

>>> from sage.all import *
>>> from sage.rings.real_mpfr import mpfr_get_exp_min
>>> mpfr_get_exp_min()
-1073741823      # 32-bit
-4611686018427387903  # 64-bit
>>> RealNumber('0.5') >> (-mpfr_get_exp_min())
2.38256490488795e-323228497      # 32-bit
8.50969131174084e-1388255822130839284  # 64-bit
>>> RealNumber('0.5') >> (-mpfr_get_exp_min()+Integer(1))
0.0000000000000000

```

`sage.rings.real_mpfr.mpfr_get_exp_min_min()`

Get the minimal value allowed for `mpfr_set_exp_min()`.

EXAMPLES:

```

sage: from sage.rings.real_mpfr import mpfr_get_exp_min_min, mpfr_set_exp_min
sage: mpfr_get_exp_min_min()
-1073741823      # 32-bit
-4611686018427387903  # 64-bit

```

```

>>> from sage.all import *
>>> from sage.rings.real_mpfr import mpfr_get_exp_min_min, mpfr_set_exp_min
>>> mpfr_get_exp_min_min()
-1073741823      # 32-bit
-4611686018427387903  # 64-bit

```

This is really the minimal value allowed:

```

sage: mpfr_set_exp_min(mpfr_get_exp_min_min() - 1)
Traceback (most recent call last):
...
OverflowError: bad value for mpfr_set_exp_min()

```

```

>>> from sage.all import *
>>> mpfr_set_exp_min(mpfr_get_exp_min_min() - Integer(1))
Traceback (most recent call last):
...
OverflowError: bad value for mpfr_set_exp_min()

```

`sage.rings.real_mpfr.mpfr_prec_max()`

Return the mpfr variable MPFR_PREC_MAX.

EXAMPLES:

```

sage: from sage.rings.real_mpfr import mpfr_prec_max
sage: mpfr_prec_max()
2147483391      # 32-bit
922337203685477551  # 64-bit

sage: R = RealField(2^31-257); R
Real Field with 2147483391 bits of precision

```

(continues on next page)

(continued from previous page)

```
sage: R = RealField(2^31-256)
Traceback (most recent call last):
...
ValueError: prec (=...) must be >= 1 and <= ...
```

```
>>> from sage.all import *
>>> from sage.rings.real_mpfr import mpfr_prec_max
>>> mpfr_prec_max()
2147483391      # 32-bit
922337203685477551  # 64-bit

>>> R = RealField(Integer(2)**Integer(31)-Integer(257)); R
Real Field with 2147483391 bits of precision

>>> R = RealField(Integer(2)**Integer(31)-Integer(256))
Traceback (most recent call last):
...
ValueError: prec (=...) must be >= 1 and <= ...
```

sage.rings.real_mpfr.**mpfr_prec_min()**

Return the mpfr variable MPFR_PREC_MIN.

EXAMPLES:

```
sage: from sage.rings.real_mpfr import mpfr_prec_min
sage: mpfr_prec_min()
1
sage: R = RealField(2)
sage: R(2) + R(1)
3.0
sage: R(4) + R(1)
4.0

sage: R = RealField(0)
Traceback (most recent call last):
...
ValueError: prec (=0) must be >= 1 and <= ...
```

```
>>> from sage.all import *
>>> from sage.rings.real_mpfr import mpfr_prec_min
>>> mpfr_prec_min()
1
>>> R = RealField(Integer(2))
>>> R(Integer(2)) + R(Integer(1))
3.0
>>> R(Integer(4)) + R(Integer(1))
4.0

>>> R = RealField(Integer(0))
Traceback (most recent call last):
...
ValueError: prec (=0) must be >= 1 and <= ...
```

`sage.rings.real_mpfr.mpfr_set_exp_max(e)`

Set the maximal exponent for MPFR numbers.

EXAMPLES:

```
sage: from sage.rings.real_mpfr import mpfr_get_exp_max, mpfr_set_exp_max
sage: old = mpfr_get_exp_max()
sage: mpfr_set_exp_max(1000)
sage: 0.5 << 1000
5.35754303593134e300
sage: 0.5 << 1001
+infinity
sage: mpfr_set_exp_max(old)
sage: 0.5 << 1001
1.07150860718627e301
```

```
>>> from sage.all import *
>>> from sage.rings.real_mpfr import mpfr_get_exp_max, mpfr_set_exp_max
>>> old = mpfr_get_exp_max()
>>> mpfr_set_exp_max(Integer(1000))
>>> RealNumber('0.5') << Integer(1000)
5.35754303593134e300
>>> RealNumber('0.5') << Integer(1001)
+infinity
>>> mpfr_set_exp_max(old)
>>> RealNumber('0.5') << Integer(1001)
1.07150860718627e301
```

`sage.rings.real_mpfr.mpfr_set_exp_min(e)`

Set the minimal exponent for MPFR numbers.

EXAMPLES:

```
sage: from sage.rings.real_mpfr import mpfr_get_exp_min, mpfr_set_exp_min
sage: old = mpfr_get_exp_min()
sage: mpfr_set_exp_min(-1000)
sage: 0.5 >> 1000
4.66631809251609e-302
sage: 0.5 >> 1001
0.0000000000000000
sage: mpfr_set_exp_min(old)
sage: 0.5 >> 1001
2.33315904625805e-302
```

```
>>> from sage.all import *
>>> from sage.rings.real_mpfr import mpfr_get_exp_min, mpfr_set_exp_min
>>> old = mpfr_get_exp_min()
>>> mpfr_set_exp_min(-Integer(1000))
>>> RealNumber('0.5') >> Integer(1000)
4.66631809251609e-302
>>> RealNumber('0.5') >> Integer(1001)
0.0000000000000000
>>> mpfr_set_exp_min(old)
```

(continues on next page)


```
>>> RealNumber('0.5') >> Integer(1001)
2.33315904625805e-302
```

AUTHORS:

- William Stein (2006-01-26): complete rewrite
- Joel B. Mohler (2006-12-16): naive rewrite into pyrex
- William Stein(2007-01): rewrite of Mohler's rewrite
- Vincent Delecroix (2010-01): plot function
- Niles Johnson (2010-08): [Issue #3893](#): `random_element()` should pass on `*args` and `**kwargs`.
- Travis Scrimshaw (2012-10-18): Added documentation for full coverage
- Vincent Klein (2017-11-14) : add `__mpc__()` to class `ComplexNumber`. `ComplexNumber` constructor support `gmpy2.mpc` parameter.

Return the complex field with real and imaginary parts having *prec bits* of precision.

```
sage: ComplexField()
Complex Field with 53 bits of precision
sage: ComplexField(100)
Complex Field with 100 bits of precision
sage: ComplexField(100).base_ring()
Real Field with 100 bits of precision
sage: i = ComplexField(200).gen()
sage: i^2
-1.000000000000000000000000000000000000000000000000000000000000
```

[illegible]

- `ComplexField_class`
- `ComplexBallField` (complex numbers with rigorous error bounds)

```
class sage.rings.complex_mpf.ComplexField_class (prec=53)
```

Bases: `ComplexField`

An approximation to the field of complex numbers using floating point numbers with any specified precision. Answers derived from calculations in this approximation may differ from what they would be if those calculations were performed in the true field of complex numbers. This is due to the rounding errors inherent to finite precision calculations.

EXAMPLES:

```
sage: C = ComplexField(); C
Complex Field with 53 bits of precision
sage: Q = RationalField()
sage: C(1/3)
0.333333333333333
sage: C(1/3, 2)
0.333333333333333 + 2.00000000000000*I
sage: C(RR.pi())
3.14159265358979
sage: C(RR.log2(), RR.pi())
0.693147180559945 + 3.14159265358979*I
```

```
>>> from sage.all import *
>>> C = ComplexField(); C
Complex Field with 53 bits of precision
>>> Q = RationalField()
>>> C(Integer(1)/Integer(3))
0.333333333333333
>>> C(Integer(1)/Integer(3), Integer(2))
0.333333333333333 + 2.00000000000000*I
>>> C(RR.pi())
3.14159265358979
>>> C(RR.log2(), RR.pi())
0.693147180559945 + 3.14159265358979*I
```

We can also coerce rational numbers and integers into `C`, but coercing a polynomial will raise an exception:

```
sage: Q = RationalField()
sage: C(1/3)
0.333333333333333
sage: S = PolynomialRing(Q, 'x')
sage: C(S.gen())
Traceback (most recent call last):
...
TypeError: cannot convert nonconstant polynomial
```

```
>>> from sage.all import *
>>> Q = RationalField()
>>> C(Integer(1)/Integer(3))
0.333333333333333
>>> S = PolynomialRing(Q, 'x')
>>> C(S.gen())
Traceback (most recent call last):
...
```

(continues on next page)

```
TypeError: cannot convert nonconstant polynomial
```

(continues on next page)

(continued from previous page)

```

sage: CC.variable_name()
'I'
sage: CC == ComplexField(200)
True
sage: CC == ComplexField(53)
False
sage: CC == 1.1
False

```

```

>>> from sage.all import *
>>> CC = ComplexField(Integer(200))
>>> CC.is_field()
True
>>> CC.characteristic()
0
>>> CC.precision()
200
>>> CC.variable_name()
'I'
>>> CC == ComplexField(Integer(200))
True
>>> CC == ComplexField(Integer(53))
False
>>> CC == RealNumber('1.1')
False

```

See also

- `ComplexField()` (constructor)
- `ComplexBallField` (complex numbers with rigorous error bounds)
- `real_mpf`

algebraic_closure()

Return the algebraic closure of `self` (which is itself).

EXAMPLES:

```

sage: CC
Complex Field with 53 bits of precision
sage: CC.algebraic_closure()
Complex Field with 53 bits of precision
sage: CC = ComplexField(1000)
sage: CC.algebraic_closure() is CC
True

```

```

>>> from sage.all import *
>>> CC
Complex Field with 53 bits of precision
>>> CC.algebraic_closure()

```

(continues on next page)

(continued from previous page)

```
Complex Field with 53 bits of precision
>>> CC = ComplexField(Integer(1000))
>>> CC.algebraic_closure() is CC
True
```

characteristic()

Return the characteristic of $\mathbf{C}$, which is 0.

EXAMPLES:

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

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

construction()

Return the functorial construction of `self`, namely the algebraic closure of the real field with the same precision.

EXAMPLES:

```
sage: c, S = CC.construction(); S
Real Field with 53 bits of precision
sage: CC == c(S)
True
```

```
>>> from sage.all import *
>>> c, S = CC.construction(); S
Real Field with 53 bits of precision
>>> CC == c(S)
True
```

gen($n=0$)

Return the generator of the complex field.

EXAMPLES:

```
sage: ComplexField().gen(0)
1.000000000000000*I
```

```
>>> from sage.all import *
>>> ComplexField().gen(Integer(0))
1.000000000000000*I
```

is_exact()

Return whether or not this field is exact, which is always `False`.

EXAMPLES:

```
sage: ComplexField().is_exact()
False
```

```
>>> from sage.all import *
>>> ComplexField().is_exact()
False
```

ngens()

The number of generators of this complex field as an **R**-algebra.

There is one generator, namely $\sqrt{-1}$.

EXAMPLES:

```
sage: ComplexField().ngens()
1
```

```
>>> from sage.all import *
>>> ComplexField().ngens()
1
```

pi()

Return π as a complex number.

EXAMPLES:

```
sage: ComplexField().pi()
3.14159265358979
sage: ComplexField(100).pi()
3.1415926535897932384626433833
```

```
>>> from sage.all import *
>>> ComplexField().pi()
3.14159265358979
>>> ComplexField(Integer(100)).pi()
3.1415926535897932384626433833
```

prec()

Return the precision of this complex field.

EXAMPLES:

```
sage: ComplexField().prec()
53
sage: ComplexField(15).prec()
15
```

```
>>> from sage.all import *
>>> ComplexField().prec()
53
>>> ComplexField(Integer(15)).prec()
15
```

precision()

Return the precision of this complex field.

EXAMPLES:

```
sage: ComplexField().prec()
53
sage: ComplexField(15).prec()
15
```

```
>>> from sage.all import *
>>> ComplexField().prec()
53
>>> ComplexField(Integer(15)).prec()
15
```

random_element (*component_max=1, *args, **kws*)

Return a uniformly distributed random number inside a square centered on the origin (by default, the square $[-1, 1] \times [-1, 1]$).

Passes additional arguments and keywords to underlying real field.

EXAMPLES:

```
sage: CC.random_element().parent() is CC
True
sage: re, im = CC.random_element()
sage: -1 <= re <= 1, -1 <= im <= 1
(True, True)
sage: CC6 = ComplexField(6)
sage: CC6.random_element().parent() is CC6
True
sage: re, im = CC6.random_element(2^-20)
sage: -2^-20 <= re <= 2^-20, -2^-20 <= im <= 2^-20
(True, True)
sage: re, im = CC6.random_element(pi^20) #_
↪needs sage.symbolic
sage: bool(-pi^20 <= re <= pi^20), bool(-pi^20 <= im <= pi^20) #_
↪needs sage.symbolic
(True, True)
```

```
>>> from sage.all import *
>>> CC.random_element().parent() is CC
True
>>> re, im = CC.random_element()
>>> -Integer(1) <= re <= Integer(1), -Integer(1) <= im <= Integer(1)
(True, True)
>>> CC6 = ComplexField(Integer(6))
>>> CC6.random_element().parent() is CC6
True
>>> re, im = CC6.random_element(Integer(2)**-Integer(20))
>>> -Integer(2)**-Integer(20) <= re <= Integer(2)**-Integer(20), -
↪Integer(2)**-Integer(20) <= im <= Integer(2)**-Integer(20)
(True, True)
>>> re, im = CC6.random_element(pi**Integer(20)) #_
↪# needs sage.symbolic
>>> bool(-pi**Integer(20) <= re <= pi**Integer(20)), bool(-pi**Integer(20) <=
↪im <= pi**Integer(20)) # needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
(True, True)
```

Passes extra positional or keyword arguments through:

```
sage: CC.random_element(distribution='1/n').parent() is CC
True
```

```
>>> from sage.all import *
>>> CC.random_element(distribution='1/n').parent() is CC
True
```

scientific_notation (*status=None*)

Set or return the scientific notation printing flag.

If this flag is `True` then complex numbers with this space as parent print using scientific notation.

EXAMPLES:

```
sage: C = ComplexField()
sage: C((0.025, 2))
0.02500000000000000 + 2.000000000000000*I
sage: C.scientific_notation(True)
sage: C((0.025, 2))
2.500000000000000e-2 + 2.000000000000000e0*I
sage: C.scientific_notation(False)
sage: C((0.025, 2))
0.02500000000000000 + 2.000000000000000*I
```

```
>>> from sage.all import *
>>> C = ComplexField()
>>> C((RealNumber('0.025'), Integer(2)))
0.02500000000000000 + 2.000000000000000*I
>>> C.scientific_notation(True)
>>> C((RealNumber('0.025'), Integer(2)))
2.500000000000000e-2 + 2.000000000000000e0*I
>>> C.scientific_notation(False)
>>> C((RealNumber('0.025'), Integer(2)))
0.02500000000000000 + 2.000000000000000*I
```

to_prec (*prec*)

Return the complex field to the specified precision.

EXAMPLES:

```
sage: CC.to_prec(10)
Complex Field with 10 bits of precision
sage: CC.to_prec(100)
Complex Field with 100 bits of precision
```

```
>>> from sage.all import *
>>> CC.to_prec(Integer(10))
Complex Field with 10 bits of precision
```

(continues on next page)

(continued from previous page)

```
>>> CC.to_prec(Integer(100))
Complex Field with 100 bits of precision
```

zeta ($n=2$)Return a primitive n -th root of unity.

INPUT:

- n – integer (default: 2)

OUTPUT: a complex n -th root of unity

EXAMPLES:

```
sage: C = ComplexField()
sage: C.zeta(2)
-1.0000000000000000
sage: C.zeta(5)
0.309016994374947 + 0.951056516295154*I
```

```
>>> from sage.all import *
>>> C = ComplexField()
>>> C.zeta(Integer(2))
-1.0000000000000000
>>> C.zeta(Integer(5))
0.309016994374947 + 0.951056516295154*I
```

class sage.rings.complex_mpf.**ComplexNumber**Bases: `FieldElement`

A floating point approximation to a complex number using any specified precision. Answers derived from calculations with such approximations may differ from what they would be if those calculations were performed with true complex numbers. This is due to the rounding errors inherent to finite precision calculations.

EXAMPLES:

```
sage: I = CC.0
sage: b = 1.5 + 2.5*I
sage: loads(b.dumps()) == b
True
```

```
>>> from sage.all import *
>>> I = CC.gen(0)
>>> b = RealNumber('1.5') + RealNumber('2.5')*I
>>> loads(b.dumps()) == b
True
```

additive_order ()Return the additive order of `self`.

EXAMPLES:

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

(continues on next page)

(continued from previous page)

```
sage: CC.gen().additive_order()
+Infinity
```

```
>>> from sage.all import *
>>> CC(Integer(0)).additive_order()
1
>>> CC.gen().additive_order()
+Infinity
```

agm (*right*, *algorithm*='optimal')

Return the Arithmetic-Geometric Mean (AGM) of *self* and *right*.

INPUT:

- *right* – complex; another complex number
- *algorithm* – string (default: 'optimal'); the algorithm to use (see below)

OUTPUT:

(complex) A value of the AGM of *self* and *right*. Note that this is a multi-valued function, and the algorithm used affects the value returned, as follows:

- 'pari': Call the `pari:agm` function from the PARI library.
- 'optimal': Use the AGM sequence such that at each stage (a, b) is replaced by $(a_1, b_1) = ((a + b)/2, \pm\sqrt{ab})$ where the sign is chosen so that $|a_1 - b_1| \leq |a_1 + b_1|$, or equivalently $\Re(b_1/a_1) \geq 0$. The resulting limit is maximal among all possible values.
- 'principal': Use the AGM sequence such that at each stage (a, b) is replaced by $(a_1, b_1) = ((a + b)/2, \pm\sqrt{ab})$ where the sign is chosen so that $\Re(b_1) \geq 0$ (the so-called principal branch of the square root).

The values $AGM(a, 0)$, $AGM(0, a)$, and $AGM(a, -a)$ are all taken to be 0.

EXAMPLES:

```
sage: a = CC(1,1)
sage: b = CC(2,-1)
sage: a.agm(b)
1.62780548487271 + 0.136827548397369*I
sage: a.agm(b, algorithm='optimal')
1.62780548487271 + 0.136827548397369*I
sage: a.agm(b, algorithm='principal')
1.62780548487271 + 0.136827548397369*I
sage: a.agm(b, algorithm='pari')
↪needs sage.libs.pari
1.62780548487271 + 0.136827548397369*I
```

```
>>> from sage.all import *
>>> a = CC(Integer(1), Integer(1))
>>> b = CC(Integer(2), -Integer(1))
>>> a.agm(b)
1.62780548487271 + 0.136827548397369*I
>>> a.agm(b, algorithm='optimal')
1.62780548487271 + 0.136827548397369*I
```

(continues on next page)

(continued from previous page)

```

>>> a.agm(b, algorithm='principal')
1.62780548487271 + 0.136827548397369*I
>>> a.agm(b, algorithm='pari')                                     #_
↳needs sage.libs.pari
1.62780548487271 + 0.136827548397369*I

```

An example to show that the returned value depends on the algorithm parameter:

```

sage: a = CC(-0.95,-0.65)
sage: b = CC(0.683,0.747)
sage: a.agm(b, algorithm='optimal')
-0.371591652351761 + 0.319894660206830*I
sage: a.agm(b, algorithm='principal')
0.338175462986180 - 0.0135326969565405*I
sage: a.agm(b, algorithm='pari')                                     #_
↳needs sage.libs.pari
-0.371591652351761 + 0.319894660206830*I
sage: a.agm(b, algorithm='optimal').abs()
0.490319232466314
sage: a.agm(b, algorithm='principal').abs()
0.338446122230459
sage: a.agm(b, algorithm='pari').abs()                               #_
↳needs sage.libs.pari
0.490319232466314

```

```

>>> from sage.all import *
>>> a = CC(-RealNumber('0.95'),-RealNumber('0.65'))
>>> b = CC(RealNumber('0.683'),RealNumber('0.747'))
>>> a.agm(b, algorithm='optimal')
-0.371591652351761 + 0.319894660206830*I
>>> a.agm(b, algorithm='principal')
0.338175462986180 - 0.0135326969565405*I
>>> a.agm(b, algorithm='pari')                                     #_
↳needs sage.libs.pari
-0.371591652351761 + 0.319894660206830*I
>>> a.agm(b, algorithm='optimal').abs()
0.490319232466314
>>> a.agm(b, algorithm='principal').abs()
0.338446122230459
>>> a.agm(b, algorithm='pari').abs()                               #_
↳needs sage.libs.pari
0.490319232466314

```

algdep (*n*, ***kws*)

Return an irreducible polynomial of degree at most *n* which is approximately satisfied by this complex number.

ALGORITHM: Uses the PARI C-library `pari:algdep` command.

INPUT: Type algebraic_dependency? at the top level prompt.

All additional parameters are passed onto the top-level `algebraic_dependency()` command.

EXAMPLES:

```

sage: C = ComplexField()
sage: z = (1/2)*(1 + sqrt(3.0) *C.0); z
0.5000000000000000 + 0.866025403784439*I
sage: p = z.algebraic_dependency(5); p
x^2 - x + 1
sage: p(z)
1.11022302462516e-16

```

```

>>> from sage.all import *
>>> C = ComplexField()
>>> z = (Integer(1)/Integer(2))*(Integer(1) + sqrt(RealNumber('3.0')) *C.
↳gen(0)); z
0.5000000000000000 + 0.866025403784439*I
>>> p = z.algebraic_dependency(Integer(5)); p
x^2 - x + 1
>>> p(z)
1.11022302462516e-16

```

algebraic_dependency (*n*, ***kws*)

Return an irreducible polynomial of degree at most *n* which is approximately satisfied by this complex number.

ALGORITHM: Uses the PARI C-library `pari:algdep` command.

INPUT: Type `algebraic_dependency?` at the top level prompt.

All additional parameters are passed onto the top-level `algebraic_dependency()` command.

EXAMPLES:

```

sage: C = ComplexField()
sage: z = (1/2)*(1 + sqrt(3.0) *C.0); z
0.5000000000000000 + 0.866025403784439*I
sage: p = z.algebraic_dependency(5); p
x^2 - x + 1
sage: p(z)
1.11022302462516e-16

```

```

>>> from sage.all import *
>>> C = ComplexField()
>>> z = (Integer(1)/Integer(2))*(Integer(1) + sqrt(RealNumber('3.0')) *C.
↳gen(0)); z
0.5000000000000000 + 0.866025403784439*I
>>> p = z.algebraic_dependency(Integer(5)); p
x^2 - x + 1
>>> p(z)
1.11022302462516e-16

```

arccos ()

Return the arccosine of `self`.

EXAMPLES:

```
sage: (1+CC(I)).arccos()
↪needs sage.libs.pari
0.904556894302381 - 1.06127506190504*I
```

```
>>> from sage.all import *
>>> (Integer(1)+CC(I)).arccos()
↪ # needs sage.libs.pari
0.904556894302381 - 1.06127506190504*I
```

arccosh()

Return the hyperbolic arccosine of `self`.

EXAMPLES:

```
sage: (1+CC(I)).arccosh()
↪needs sage.libs.pari
1.06127506190504 + 0.904556894302381*I
```

```
>>> from sage.all import *
>>> (Integer(1)+CC(I)).arccosh()
↪ # needs sage.libs.pari
1.06127506190504 + 0.904556894302381*I
```

arccoth()

Return the hyperbolic arccotangent of `self`.

EXAMPLES:

```
sage: ComplexField(100)(1,1).arccoth()
↪needs sage.libs.pari
0.40235947810852509365018983331 - 0.55357435889704525150853273009*I
```

```
>>> from sage.all import *
>>> ComplexField(Integer(100))(Integer(1),Integer(1)).arccoth()
↪ # needs sage.libs.pari
0.40235947810852509365018983331 - 0.55357435889704525150853273009*I
```

arccsch()

Return the hyperbolic arccosecant of `self`.

EXAMPLES:

```
sage: ComplexField(100)(1,1).arccsch()
↪needs sage.libs.pari
0.53063753095251782601650945811 - 0.45227844715119068206365839783*I
```

```
>>> from sage.all import *
>>> ComplexField(Integer(100))(Integer(1),Integer(1)).arccsch()
↪ # needs sage.libs.pari
0.53063753095251782601650945811 - 0.45227844715119068206365839783*I
```

arcsech()

Return the hyperbolic arcsecant of `self`.

EXAMPLES:

```
sage: ComplexField(100)(1,1).arcsech() #_
↪needs sage.libs.pari
0.53063753095251782601650945811 - 1.1185178796437059371676632938*I
```

```
>>> from sage.all import *
>>> ComplexField(Integer(100))(Integer(1),Integer(1)).arcsech() _
↪ # needs sage.libs.pari
0.53063753095251782601650945811 - 1.1185178796437059371676632938*I
```

arcsin()

Return the arcsine of `self`.

EXAMPLES:

```
sage: (1+CC(I)).arcsin() #_
↪needs sage.libs.pari
0.666239432492515 + 1.06127506190504*I
```

```
>>> from sage.all import *
>>> (Integer(1)+CC(I)).arcsin() _
↪ # needs sage.libs.pari
0.666239432492515 + 1.06127506190504*I
```

arcsinh()

Return the hyperbolic arcsine of `self`.

EXAMPLES:

```
sage: (1+CC(I)).arcsinh() #_
↪needs sage.libs.pari
1.06127506190504 + 0.666239432492515*I
```

```
>>> from sage.all import *
>>> (Integer(1)+CC(I)).arcsinh() _
↪ # needs sage.libs.pari
1.06127506190504 + 0.666239432492515*I
```

arctan()

Return the arctangent of `self`.

EXAMPLES:

```
sage: (1+CC(I)).arctan() #_
↪needs sage.libs.pari
1.01722196789785 + 0.402359478108525*I
```

```
>>> from sage.all import *
>>> (Integer(1)+CC(I)).arctan() _
↪ # needs sage.libs.pari
1.01722196789785 + 0.402359478108525*I
```

arctanh()

Return the hyperbolic arctangent of `self`.

EXAMPLES:

```
sage: (1+CC(I)).arctanh()
↪needs sage.libs.pari
0.402359478108525 + 1.01722196789785*I
```

```
>>> from sage.all import *
>>> (Integer(1)+CC(I)).arctanh()
↪ # needs sage.libs.pari
0.402359478108525 + 1.01722196789785*I
```

arg()

See `argument()`.

EXAMPLES:

```
sage: i = CC.0
sage: (i^2).arg()
3.14159265358979
```

```
>>> from sage.all import *
>>> i = CC.gen(0)
>>> (i**Integer(2)).arg()
3.14159265358979
```

argument()

The argument (angle) of the complex number, normalized so that $-\pi < \theta \leq \pi$.

EXAMPLES:

```
sage: i = CC.0
sage: (i^2).argument()
3.14159265358979
sage: (1+i).argument()
0.785398163397448
sage: i.argument()
1.57079632679490
sage: (-i).argument()
-1.57079632679490
sage: (RR('-0.001') - i).argument()
-1.57179632646156
```

```
>>> from sage.all import *
>>> i = CC.gen(0)
>>> (i**Integer(2)).argument()
3.14159265358979
>>> (Integer(1)+i).argument()
0.785398163397448
>>> i.argument()
1.57079632679490
>>> (-i).argument()
```

(continues on next page)

(continued from previous page)

```
-1.57079632679490
>>> (RR('-0.001') - i).argument()
-1.57179632646156
```

conjugate()

Return the complex conjugate of this complex number.

EXAMPLES:

```
sage: i = CC.0
sage: (1+i).conjugate()
1.000000000000000 - 1.000000000000000*I
```

```
>>> from sage.all import *
>>> i = CC.gen(0)
>>> (Integer(1)+i).conjugate()
1.000000000000000 - 1.000000000000000*I
```

cos()

Return the cosine of `self`.

EXAMPLES:

```
sage: (1+CC(I)).cos()
0.833730025131149 - 0.988897705762865*I
```

```
>>> from sage.all import *
>>> (Integer(1)+CC(I)).cos()
0.833730025131149 - 0.988897705762865*I
```

cosh()

Return the hyperbolic cosine of `self`.

EXAMPLES:

```
sage: (1+CC(I)).cosh()
0.833730025131149 + 0.988897705762865*I
```

```
>>> from sage.all import *
>>> (Integer(1)+CC(I)).cosh()
0.833730025131149 + 0.988897705762865*I
```

cot()

Return the cotangent of `self`.

EXAMPLES:

```
sage: # needs sage.libs.pari
sage: (1+CC(I)).cot()
0.217621561854403 - 0.868014142895925*I
sage: i = ComplexField(200).0
sage: (1+i).cot()
0.21762156185440268136513424360523807352075436916785404091068 - 0.
```

(continues on next page)

(continued from previous page)

```

↪86801414289592494863584920891627388827343874994609327121115*I
sage: i = ComplexField(220).0
sage: (1+i).cot()
0.21762156185440268136513424360523807352075436916785404091068124239 - 0.
↪86801414289592494863584920891627388827343874994609327121115071646*I

```

```

>>> from sage.all import *
>>> # needs sage.libs.pari
>>> (Integer(1)+CC(I)).cot()
0.217621561854403 - 0.868014142895925*I
>>> i = ComplexField(Integer(200)).gen(0)
>>> (Integer(1)+i).cot()
0.21762156185440268136513424360523807352075436916785404091068 - 0.
↪86801414289592494863584920891627388827343874994609327121115*I
>>> i = ComplexField(Integer(220)).gen(0)
>>> (Integer(1)+i).cot()
0.21762156185440268136513424360523807352075436916785404091068124239 - 0.
↪86801414289592494863584920891627388827343874994609327121115071646*I

```

coth()Return the hyperbolic cotangent of `self`.

EXAMPLES:

```

sage: ComplexField(100)(1,1).coth() #_
↪needs sage.libs.pari
0.86801414289592494863584920892 - 0.21762156185440268136513424361*I

```

```

>>> from sage.all import *
>>> ComplexField(Integer(100))(Integer(1),Integer(1)).coth()
↪ # needs sage.libs.pari
0.86801414289592494863584920892 - 0.21762156185440268136513424361*I

```

csc()Return the cosecant of `self`.

EXAMPLES:

```

sage: ComplexField(100)(1,1).csc() #_
↪needs sage.libs.pari
0.62151801717042842123490780586 - 0.30393100162842645033448560451*I

```

```

>>> from sage.all import *
>>> ComplexField(Integer(100))(Integer(1),Integer(1)).csc()
↪ # needs sage.libs.pari
0.62151801717042842123490780586 - 0.30393100162842645033448560451*I

```

csch()Return the hyperbolic cosecant of `self`.

EXAMPLES:

```
sage: ComplexField(100)(1,1).csch()
↪needs sage.libs.pari
0.30393100162842645033448560451 - 0.62151801717042842123490780586*I
```

```
>>> from sage.all import *
>>> ComplexField(Integer(100))(Integer(1),Integer(1)).csch()
↪ # needs sage.libs.pari
0.30393100162842645033448560451 - 0.62151801717042842123490780586*I
```

dilog()

Return the complex dilogarithm of `self`.

The complex dilogarithm, or Spence's function, is defined by

$$Li_2(z) = - \int_0^z \frac{\log|1-\zeta|}{\zeta} d(\zeta) = \sum_{k=1}^{\infty} \frac{z^k}{k}$$

Note that the series definition can only be used for $|z| < 1$.

EXAMPLES:

```
sage: a = ComplexNumber(1,0)
sage: a.dilog()
↪needs sage.libs.pari
1.64493406684823
sage: float(pi^2/6)
↪needs sage.symbolic
1.6449340668482262
```

```
>>> from sage.all import *
>>> a = ComplexNumber(Integer(1),Integer(0))
>>> a.dilog()
↪needs sage.libs.pari
1.64493406684823
>>> float(pi**Integer(2)/Integer(6))
↪ # needs sage.symbolic
1.6449340668482262
```

```
sage: b = ComplexNumber(0,1)
sage: b.dilog()
↪needs sage.libs.pari
-0.205616758356028 + 0.915965594177219*I
```

```
>>> from sage.all import *
>>> b = ComplexNumber(Integer(0),Integer(1))
>>> b.dilog()
↪needs sage.libs.pari
-0.205616758356028 + 0.915965594177219*I
```

```
sage: c = ComplexNumber(0,0)
sage: c.dilog()
↪needs sage.libs.pari
0.0000000000000000
```

```
>>> from sage.all import *
>>> c = ComplexNumber(Integer(0), Integer(0))
>>> c.dilog() #_
↳needs sage.libs.pari
0.0000000000000000
```

eta (*omit_frac=False*)

Return the value of the Dedekind η function on `self`, intelligently computed using $\mathrm{SL}(2, \mathbb{Z})$ transformations.

The η function is

$$\eta(z) = e^{\pi iz/12} \prod_{n=1}^{\infty} (1 - e^{2\pi inz})$$

INPUT:

- `self` – element of the upper half plane (if not, raises a `ValueError`)
- `omit_frac` – boolean (default: `False`); if `True`, omit the $e^{\pi iz/12}$ factor

OUTPUT: a complex number

ALGORITHM: Uses the PARI C library.

EXAMPLES:

First we compute $\eta(1 + i)$:

```
sage: i = CC.0
sage: z = 1 + i; z.eta() #_
↳needs sage.libs.pari
0.742048775836565 + 0.198831370229911*I
```

```
>>> from sage.all import *
>>> i = CC.gen(0)
>>> z = Integer(1) + i; z.eta() #_
↳ # needs sage.libs.pari
0.742048775836565 + 0.198831370229911*I
```

We compute eta to low precision directly from the definition:

```
sage: pi = CC(pi) # otherwise we will get a symbolic result. #_
↳needs sage.symbolic
sage: exp(pi * i * z / 12) * prod(1 - exp(2*pi*i*n*z) #_
↳needs sage.libs.pari sage.symbolic
....: for n in range(1,10))
0.742048775836565 + 0.198831370229911*I
```

```
>>> from sage.all import *
>>> pi = CC(pi) # otherwise we will get a symbolic result. #_
↳needs sage.symbolic
>>> exp(pi * i * z / Integer(12)) * prod(Integer(1) -_
↳exp(Integer(2)*pi*i*n*z) # needs sage.libs.pari sage.
↳symbolic
....: for n in range(Integer(1), Integer(10)))
0.742048775836565 + 0.198831370229911*I
```

The optional argument allows us to omit the fractional part:

```
sage: z.eta(omit_frac=True) #_
↪needs sage.libs.pari
0.998129069925959
sage: prod(1 - exp(2*pi*i*n*z) for n in range(1,10)) #_
↪needs sage.libs.pari sage.symbolic
0.998129069925958 + 4.59099857829247e-19*I
```

```
>>> from sage.all import *
>>> z.eta(omit_frac=True) #_
↪needs sage.libs.pari
0.998129069925959
>>> prod(Integer(1) - exp(Integer(2)*pi*i*n*z) for n in range(Integer(1),
↪Integer(10))) # needs sage.libs.pari sage.symbolic
0.998129069925958 + 4.59099857829247e-19*I
```

We illustrate what happens when z is not in the upper half plane:

```
sage: z = CC(1)
sage: z.eta() #_
↪needs sage.libs.pari
Traceback (most recent call last):
...
ValueError: value must be in the upper half plane
```

```
>>> from sage.all import *
>>> z = CC(Integer(1))
>>> z.eta() #_
↪needs sage.libs.pari
Traceback (most recent call last):
...
ValueError: value must be in the upper half plane
```

You can also use functional notation:

```
sage: eta(1 + CC(I)) #_
↪needs sage.libs.pari
0.742048775836565 + 0.198831370229911*I
```

```
>>> from sage.all import *
>>> eta(Integer(1) + CC(I))
↪ # needs sage.libs.pari
0.742048775836565 + 0.198831370229911*I
```

`exp()`

Compute e^z or $\exp(z)$.

EXAMPLES:

```
sage: i = ComplexField(300).0
sage: z = 1 + i
sage: z.exp()
```

(continues on next page)

(continued from previous page)

```
1.4686939399158851571389675973266042613269567366290087227976756763109369658595
↪1213872272450 + 2.2873552871788423912081719067005018089555862566683556809386
↪5811410364716018934540926734485*I
```

```
>>> from sage.all import *
>>> i = ComplexField(Integer(300)).gen(0)
>>> z = Integer(1) + i
>>> z.exp()
1.4686939399158851571389675973266042613269567366290087227976756763109369658595
↪1213872272450 + 2.2873552871788423912081719067005018089555862566683556809386
↪5811410364716018934540926734485*I
```

gamma()

Return the Gamma function evaluated at this complex number.

EXAMPLES:

```
sage: i = ComplexField(30).0
sage: (1 + i).gamma()
↪needs sage.libs.pari
0.49801567 - 0.15494983*I
```

↪

```
>>> from sage.all import *
>>> i = ComplexField(Integer(30)).gen(0)
>>> (Integer(1) + i).gamma()
↪ # needs sage.libs.pari
0.49801567 - 0.15494983*I
```

↪

gamma_inc(t)

Return the incomplete Gamma function evaluated at this complex number.

EXAMPLES:

```
sage: # needs sage.libs.pari
sage: C, i = ComplexField(30).objgen()
sage: (1+i).gamma_inc(2 + 3*i) # abs tol 2e-10
0.0020969149 - 0.059981914*I
sage: (1+i).gamma_inc(5)
-0.0013781309 + 0.0065198200*I
sage: C(2).gamma_inc(1 + i)
0.70709210 - 0.42035364*I
sage: CC(2).gamma_inc(5)
0.0404276819945128
```

```
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> C, i = ComplexField(Integer(30)).objgen()
>>> (Integer(1)+i).gamma_inc(Integer(2) + Integer(3)*i) # abs tol 2e-10
0.0020969149 - 0.059981914*I
>>> (Integer(1)+i).gamma_inc(Integer(5))
-0.0013781309 + 0.0065198200*I
>>> C(Integer(2)).gamma_inc(Integer(1) + i)
```

(continues on next page)

(continued from previous page)

```
0.70709210 - 0.42035364*I
>>> CC(Integer(2)).gamma_inc(Integer(5))
0.0404276819945128
```

imag()

Return imaginary part of `self`.

EXAMPLES:

[illegible][illegible]

```
imag_part ()
```

Return imaginary part of `self`.

EXAMPLES:

[illegible][illegible]

is_NaN()

Check if `self` is not-a-number.

EXAMPLES:

```
sage: CC(1, 2).is_NaN()
False
sage: CC(NaN).is_NaN()
True
sage: CC(NaN, 2).log().is_NaN()
True
```

```
>>> from sage.all import *
>>> CC(Integer(1), Integer(2)).is_NaN()
False
>>> CC(NaN).is_NaN()
True
>>> CC(NaN, Integer(2)).log().is_NaN()
True
```

is_imaginary()

Return True if `self` is imaginary, i.e., has real part zero.

EXAMPLES:

```
sage: CC(1.23*i).is_imaginary()
True
sage: CC(1+i).is_imaginary()
False
```

```
>>> from sage.all import *
>>> CC(RealNumber('1.23')*i).is_imaginary()
True
>>> CC(Integer(1)+i).is_imaginary()
False
```

is_infinity()

Check if `self` is ∞ .

EXAMPLES:

```
sage: CC(1, 2).is_infinity()
False
sage: CC(0, oo).is_infinity()
True
```

```
>>> from sage.all import *
>>> CC(Integer(1), Integer(2)).is_infinity()
False
>>> CC(Integer(0), oo).is_infinity()
True
```

is_integer()

Return True if `self` is an integer.

EXAMPLES:

```
sage: CC(3).is_integer()
True
sage: CC(1,2).is_integer()
False
```

```
>>> from sage.all import *
>>> CC(Integer(3)).is_integer()
True
>>> CC(Integer(1), Integer(2)).is_integer()
False
```

is_negative_infinity()

Check if self is $-\infty$.

EXAMPLES:

```
sage: CC(1, 2).is_negative_infinity()
False
sage: CC(-oo, 0).is_negative_infinity()
True
sage: CC(0, -oo).is_negative_infinity()
False
```

```
>>> from sage.all import *
>>> CC(Integer(1), Integer(2)).is_negative_infinity()
False
>>> CC(-oo, Integer(0)).is_negative_infinity()
True
>>> CC(Integer(0), -oo).is_negative_infinity()
False
```

is_positive_infinity()

Check if self is $+\infty$.

EXAMPLES:

```
sage: CC(1, 2).is_positive_infinity()
False
sage: CC(oo, 0).is_positive_infinity()
True
sage: CC(0, oo).is_positive_infinity()
False
```

```
>>> from sage.all import *
>>> CC(Integer(1), Integer(2)).is_positive_infinity()
False
>>> CC(oo, Integer(0)).is_positive_infinity()
True
>>> CC(Integer(0), oo).is_positive_infinity()
False
```


is_real()

Return True if self is real, i.e., has imaginary part zero.

EXAMPLES:

```
sage: CC(1.23).is_real()
True
sage: CC(1+i).is_real()
False
```

```
>>> from sage.all import *
>>> CC(RealNumber('1.23')).is_real()
True
>>> CC(Integer(1)+i).is_real()
False
```

is_square()This function always returns true as $\mathbb{C}$ is algebraically closed.

EXAMPLES:

```
sage: a = ComplexNumber(2,1)
sage: a.is_square()
True
```

```
>>> from sage.all import *
>>> a = ComplexNumber(Integer(2), Integer(1))
>>> a.is_square()
True
```

 $\mathbb{C}$ is algebraically closed, hence every element is a square:

```
sage: b = ComplexNumber(5)
sage: b.is_square()
True
```

```
>>> from sage.all import *
>>> b = ComplexNumber(Integer(5))
>>> b.is_square()
True
```

log(base=None)Complex logarithm of z with branch chosen as follows: Write $z = \rho e^{i\theta}$ with $-\pi < \theta \leq \pi$. Then $\log(z) = \log(\rho) + i\theta$.**Warning**

Currently the real log is computed using floats, so there is potential precision loss.

EXAMPLES:

```
sage: a = ComplexNumber(2,1)
sage: a.log()
0.804718956217050 + 0.463647609000806*I
sage: log(a.abs())
0.804718956217050
sage: a.argument()
0.463647609000806
```

```
>>> from sage.all import *
>>> a = ComplexNumber(Integer(2),Integer(1))
>>> a.log()
0.804718956217050 + 0.463647609000806*I
>>> log(a.abs())
0.804718956217050
>>> a.argument()
0.463647609000806
```

```
sage: b = ComplexNumber(float(exp(42)),0)
sage: b.log() # abs tol 1e-12
41.99999999999971
```

```
>>> from sage.all import *
>>> b = ComplexNumber(float(exp(Integer(42))),Integer(0))
>>> b.log() # abs tol 1e-12
41.99999999999971
```

```
sage: c = ComplexNumber(-1,0)
sage: c.log()
3.14159265358979*I
```

```
>>> from sage.all import *
>>> c = ComplexNumber(-Integer(1),Integer(0))
>>> c.log()
3.14159265358979*I
```

The option of a base is included for compatibility with other logs:

```
sage: c = ComplexNumber(-1,0)
sage: c.log(2)
4.53236014182719*I
```

```
>>> from sage.all import *
>>> c = ComplexNumber(-Integer(1),Integer(0))
>>> c.log(Integer(2))
4.53236014182719*I
```

If either component (real or imaginary) of the complex number is NaN (not a number), log will return the complex NaN:

```
sage: c = ComplexNumber(NaN,2)
sage: c.log()
NaN + NaN*I
```

```
>>> from sage.all import *
>>> c = ComplexNumber(NaN, Integer(2))
>>> c.log()
NaN + NaN*I
```

multiplicative_order()

Return the multiplicative order of this complex number, if known, or raise a `NotImplementedError`.

EXAMPLES:

```
sage: C.<i> = ComplexField()
sage: i.multiplicative_order()
4
sage: C(1).multiplicative_order()
1
sage: C(-1).multiplicative_order()
2
sage: C(i^2).multiplicative_order()
2
sage: C(-i).multiplicative_order()
4
sage: C(2).multiplicative_order()
+Infinity
sage: w = (1+sqrt(-3.0))/2; w
0.5000000000000000 + 0.866025403784439*I
sage: abs(w)
1.0000000000000000
sage: w.multiplicative_order()
Traceback (most recent call last):
...
NotImplementedError: order of element not known
```

```
>>> from sage.all import *
>>> C = ComplexField(names=('i',)); (i,) = C._first_ngens(1)
>>> i.multiplicative_order()
4
>>> C(Integer(1)).multiplicative_order()
1
>>> C(-Integer(1)).multiplicative_order()
2
>>> C(i**Integer(2)).multiplicative_order()
2
>>> C(-i).multiplicative_order()
4
>>> C(Integer(2)).multiplicative_order()
+Infinity
>>> w = (Integer(1)+sqrt(-RealNumber('3.0')))/Integer(2); w
0.5000000000000000 + 0.866025403784439*I
>>> abs(w)
1.0000000000000000
>>> w.multiplicative_order()
Traceback (most recent call last):
...

```

(continues on next page)

(continued from previous page)

```
NotImplementedError: order of element not known
```

norm()

Return the norm of this complex number.

If $c = a + bi$ is a complex number, then the norm of c is defined as the product of c and its complex conjugate:

$$\text{norm}(c) = \text{norm}(a + bi) = c \cdot \bar{c} = a^2 + b^2.$$

The norm of a complex number is different from its absolute value. The absolute value of a complex number is defined to be the square root of its norm. A typical use of the complex norm is in the integral domain $\mathbf{Z}[i]$ of Gaussian integers, where the norm of each Gaussian integer $c = a + bi$ is defined as its complex norm.

See also

- `sage.misc.functional.norm()`
- `sage.rings.complex_double.ComplexDoubleElement.norm()`

EXAMPLES:

This indeed acts as the square function when the imaginary component of `self` is equal to zero:

```
sage: a = ComplexNumber(2,1)
sage: a.norm()
5.000000000000000
sage: b = ComplexNumber(4.2,0)
sage: b.norm()
17.640000000000000
sage: b^2
17.640000000000000
```

```
>>> from sage.all import *
>>> a = ComplexNumber(Integer(2), Integer(1))
>>> a.norm()
5.000000000000000
>>> b = ComplexNumber(RealNumber('4.2'), Integer(0))
>>> b.norm()
17.640000000000000
>>> b**Integer(2)
17.640000000000000
```

nth_root(*n*, *all=False*)

The n -th root function.

INPUT:

- `all` – boolean (default: `False`); if `True`, return a list of all n -th roots

EXAMPLES:

```
sage: a = CC(27)
sage: a.nth_root(3)
3.000000000000000
```

(continues on next page)

(continued from previous page)

```
sage: a.nth_root(3, all=True)
[3.000000000000000,
 -1.500000000000000 + 2.59807621135332*I,
 -1.500000000000000 - 2.59807621135332*I]
sage: a = ComplexField(20)(2,1)
sage: [r^7 for r in a.nth_root(7, all=True)]
[2.0000 + 1.0000*I, 2.0000 + 1.0000*I, 2.0000 + 1.0000*I, 2.0000 + 1.0000*I,
 2.0000 + 1.0000*I, 2.0000 + 1.0001*I, 2.0000 + 1.0001*I]
```

```
>>> from sage.all import *
>>> a = CC(Integer(27))
>>> a.nth_root(Integer(3))
3.000000000000000
>>> a.nth_root(Integer(3), all=True)
[3.000000000000000,
 -1.500000000000000 + 2.59807621135332*I,
 -1.500000000000000 - 2.59807621135332*I]
>>> a = ComplexField(Integer(20))(Integer(2), Integer(1))
>>> [r**Integer(7) for r in a.nth_root(Integer(7), all=True)]
[2.0000 + 1.0000*I, 2.0000 + 1.0000*I, 2.0000 + 1.0000*I, 2.0000 + 1.0000*I,
 2.0000 + 1.0000*I, 2.0000 + 1.0001*I, 2.0000 + 1.0001*I]
```

plot (**kargs)

Plots this complex number as a point in the plane.

The accepted options are the ones of `point2d()`. Type `point2d.options` to see all options.

Note

Just wraps the `sage.plot.point.point2d` method

EXAMPLES:

You can either use the indirect:

```
sage: z = CC(0,1)
sage: plot(z) #_
↳needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> z = CC(Integer(0), Integer(1))
>>> plot(z) #_
↳needs sage.plot
Graphics object consisting of 1 graphics primitive
```

or the more direct:

```
sage: z = CC(0,1)
sage: z.plot() #_
↳needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> z = CC(Integer(0),Integer(1))
>>> z.plot()
↳needs sage.plot
```

Graphics object consisting of 1 graphics primitive

```
prec()
```

Return precision of this complex number.

EXAMPLES:

```
sage: i = ComplexField(2000).0
sage: i.prec()
2000
```

```
>>> from sage.all import *
>>> i = ComplexField(Integer(2000)).gen(0)
>>> i.prec()
2000
```

real()

Return real part of `self`.

EXAMPLES:

[illegible][illegible]

```
real_part()
```

Return real part of `self`.

EXAMPLES:

```
sage: i = ComplexField(100).0  
sage: z = 2 + 3*i  
sage: x = z.real(); x  
2.000000000000000000000000000000
```

(continues on next page)

(continued from previous page)

```
sage: x.parent()
Real Field with 100 bits of precision
sage: z.real_part()
2.0000000000000000000000000000000
```

[illegible]

sec()

Return the secant of `self`.

EXAMPLES:

```
sage: ComplexField(100)(1,1).sec()
↳needs sage.libs.pari
0.49833703055518678521380589177 + 0.59108384172104504805039169297*I
```

```
>>> from sage.all import *
>>> ComplexField(Integer(100))(Integer(1),Integer(1)).sec()
↪ # needs sage.libs.pari
0.49833703055518678521380589177 + 0.59108384172104504805039169297*I
```

sech ()

Return the hyperbolic secant of `self`.

EXAMPLES:

```
sage: ComplexField(100)(1,1).sech()  
↳needs sage.libs.pari  
0.49833703055518678521380589177 - 0.59108384172104504805039169297*I
```

```
>>> from sage.all import *
>>> ComplexField(Integer(100))(Integer(1),Integer(1)).sech()
↪                                     # needs sage.libs.pari
0.49833703055518678521380589177 - 0.59108384172104504805039169297*I
```

sin()

Return the sine of `self`.

EXAMPLES:

```
sage: (1+CC(I)).sin()
1.29845758141598 + 0.634963914784736*I
```

```
>>> from sage.all import *
>>> (Integer(1)+CC(I)).sin()
1.29845758141598 + 0.634963914784736*I
```

sinh()

Return the hyperbolic sine of `self`.

EXAMPLES:

```
sage: (1+CC(I)).sinh()
0.634963914784736 + 1.29845758141598*I
```

```
>>> from sage.all import *
>>> (Integer(1)+CC(I)).sinh()
0.634963914784736 + 1.29845758141598*I
```

sqrt (*all=False*)

The square root function, taking the branch cut to be the negative real axis.

INPUT:

- `all` – boolean (default: `False`); if `True`, return a list of all square roots

EXAMPLES:

```
sage: C.<i> = ComplexField(30)
sage: i.sqrt()
0.70710678 + 0.70710678*I
sage: (1+i).sqrt()
1.0986841 + 0.45508986*I
sage: (C(-1)).sqrt()
1.0000000*I
sage: (1 + 1e-100*i).sqrt()^2
1.0000000 + 1.0000000e-100*I
sage: i = ComplexField(200).0
sage: i.sqrt()
0.70710678118654752440084436210484903928483593768847403658834 + 0.
↪ 70710678118654752440084436210484903928483593768847403658834*I
```

```
>>> from sage.all import *
>>> C = ComplexField(Integer(30), names=('i',)); (i,) = C._first_ngens(1)
>>> i.sqrt()
0.70710678 + 0.70710678*I
>>> (Integer(1)+i).sqrt()
1.0986841 + 0.45508986*I
>>> (C(-Integer(1))).sqrt()
1.0000000*I
>>> (Integer(1) + RealNumber('1e-100')*i).sqrt()**Integer(2)
1.0000000 + 1.0000000e-100*I
>>> i = ComplexField(Integer(200)).gen(0)
>>> i.sqrt()
0.70710678118654752440084436210484903928483593768847403658834 + 0.
↪ 70710678118654752440084436210484903928483593768847403658834*I
```


`str` (*base=10, istr='I', **kws*)

Return a string representation of `self`.

INPUT:

- `base` – (default: 10) base for output
- `istr` – (default: `I`) string representation of the complex unit
- `**kws` – other arguments to pass to the `str()` method of the real numbers in the real and imaginary parts

EXAMPLES:

```
sage: # needs sage.symbolic
sage: a = CC(pi + I*e); a
3.14159265358979 + 2.71828182845905*I
sage: a.str(truncate=True)
'3.14159265358979 + 2.71828182845905*I'
sage: a.str()
'3.1415926535897931 + 2.7182818284590451*I'
sage: a.str(base=2)
'11.001001000011111101101010100010001000010110100011000 + 10.
↪101101111110000101010001011000101000101001101101001*I'
sage: CC(0.5 + 0.625*I).str(base=2)
'0.100000000000000000000000000000000000000000000000000 + 0.
↪101000000000000000000000000000000000000000000000000*I'
sage: a.str(base=16)
'3.243f6a8885a30 + 2.b7e151628aed2*I'
sage: a.str(base=36)
'3.53i5ab8p5fc + 2.puw5nggjf8f*I'

sage: CC(0)
0.0000000000000000
sage: CC.0.str(istr='%i')
'1.0000000000000000*%i'
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> a = CC(pi + I*e); a
3.14159265358979 + 2.71828182845905*I
>>> a.str(truncate=True)
'3.14159265358979 + 2.71828182845905*I'
>>> a.str()
'3.1415926535897931 + 2.7182818284590451*I'
>>> a.str(base=Integer(2))
'11.001001000011111101101010100010001000010110100011000 + 10.
↪101101111110000101010001011000101000101001101101001*I'
>>> CC(RealNumber('0.5') + RealNumber('0.625')*I).str(base=Integer(2))
'0.100000000000000000000000000000000000000000000000000 + 0.
↪101000000000000000000000000000000000000000000000000*I'
>>> a.str(base=Integer(16))
'3.243f6a8885a30 + 2.b7e151628aed2*I'
>>> a.str(base=Integer(36))
'3.53i5ab8p5fc + 2.puw5nggjf8f*I'
```

(continues on next page)

(continued from previous page)

```
>>> CC(Integer(0))
0.0000000000000000
>>> CC.gen(0).str(istr='%i')
'1.0000000000000000*%i'
```

tan()

Return the tangent of self.

EXAMPLES:

```
sage: (1+CC(I)).tan()
0.271752585319512 + 1.08392332733869*I
```

```
>>> from sage.all import *
>>> (Integer(1)+CC(I)).tan()
0.271752585319512 + 1.08392332733869*I
```

tanh()

Return the hyperbolic tangent of self.

EXAMPLES:

```
sage: (1+CC(I)).tanh()
1.08392332733869 + 0.271752585319512*I
```

```
>>> from sage.all import *
>>> (Integer(1)+CC(I)).tanh()
1.08392332733869 + 0.271752585319512*I
```

zeta()

Return the Riemann zeta function evaluated at this complex number.

EXAMPLES:

```
sage: i = ComplexField(30).gen()
sage: z = 1 + i
sage: z.zeta() #_
↪needs sage.libs.pari
0.58215806 - 0.92684856*I
sage: zeta(z) #_
↪needs sage.libs.pari
0.58215806 - 0.92684856*I

sage: CC(1).zeta()
Infinity
```

```
>>> from sage.all import *
>>> i = ComplexField(Integer(30)).gen()
>>> z = Integer(1) + i
>>> z.zeta() #_
↪needs sage.libs.pari
```

(continues on next page)

(continued from previous page)

```

0.58215806 - 0.92684856*I
>>> zeta(z)
↳needs sage.libs.pari
0.58215806 - 0.92684856*I

>>> CC(Integer(1)).zeta()
Infinity

```

```
class sage.rings.complex_mpfr.RRtoCC
```

Bases: Map

EXAMPLES:

```

sage: from sage.rings.complex_mpfr import RRtoCC
sage: RRtoCC(RR, CC)
Natural map:
  From: Real Field with 53 bits of precision
  To:   Complex Field with 53 bits of precision

```

```

>>> from sage.all import *
>>> from sage.rings.complex_mpfr import RRtoCC
>>> RRtoCC(RR, CC)
Natural map:
  From: Real Field with 53 bits of precision
  To:   Complex Field with 53 bits of precision

```

```
sage.rings.complex_mpfr.cmp_abs(a, b)
```

Return -1 , 0 , or 1 according to whether $|a|$ is less than, equal to, or greater than $|b|$.

Optimized for non-close numbers, where the ordering can be determined by examining exponents.

EXAMPLES:

```

sage: from sage.rings.complex_mpfr import cmp_abs
sage: cmp_abs(CC(5), CC(1))
1
sage: cmp_abs(CC(5), CC(4))
1
sage: cmp_abs(CC(5), CC(5))
0
sage: cmp_abs(CC(5), CC(6))
-1
sage: cmp_abs(CC(5), CC(100))
-1
sage: cmp_abs(CC(-100), CC(1))
1
sage: cmp_abs(CC(-100), CC(100))
0
sage: cmp_abs(CC(-100), CC(1000))
-1
sage: cmp_abs(CC(1,1), CC(1))
1
sage: cmp_abs(CC(1,1), CC(2))

```

(continues on next page)

(continued from previous page)

```

-1
sage: cmp_abs(CC(1,1), CC(1,0.99999))
1
sage: cmp_abs(CC(1,1), CC(1,-1))
0
sage: cmp_abs(CC(0), CC(1))
-1
sage: cmp_abs(CC(1), CC(0))
1
sage: cmp_abs(CC(0), CC(0))
0
sage: cmp_abs(CC(2,1), CC(1,2))
0

```

```

>>> from sage.all import *
>>> from sage.rings.complex_mpfr import cmp_abs
>>> cmp_abs(CC(Integer(5)), CC(Integer(1)))
1
>>> cmp_abs(CC(Integer(5)), CC(Integer(4)))
1
>>> cmp_abs(CC(Integer(5)), CC(Integer(5)))
0
>>> cmp_abs(CC(Integer(5)), CC(Integer(6)))
-1
>>> cmp_abs(CC(Integer(5)), CC(Integer(100)))
-1
>>> cmp_abs(CC(-Integer(100)), CC(Integer(1)))
1
>>> cmp_abs(CC(-Integer(100)), CC(Integer(100)))
0
>>> cmp_abs(CC(-Integer(100)), CC(Integer(1000)))
-1
>>> cmp_abs(CC(Integer(1), Integer(1)), CC(Integer(1)))
1
>>> cmp_abs(CC(Integer(1), Integer(1)), CC(Integer(2)))
-1
>>> cmp_abs(CC(Integer(1), Integer(1)), CC(Integer(1), RealNumber('0.99999')))
1
>>> cmp_abs(CC(Integer(1), Integer(1)), CC(Integer(1), -Integer(1)))
0
>>> cmp_abs(CC(Integer(0)), CC(Integer(1)))
-1
>>> cmp_abs(CC(Integer(1)), CC(Integer(0)))
1
>>> cmp_abs(CC(Integer(0)), CC(Integer(0)))
0
>>> cmp_abs(CC(Integer(2), Integer(1)), CC(Integer(1), Integer(2)))
0

```

sage.rings.complex_mpfr.**create_ComplexNumber**(s_real, s_imag=None, pad=0, min_prec=53)

Return the complex number defined by the strings s_real and s_imag as an element of Complex-Field(prec=n), where n potentially has slightly more (controlled by pad) bits than given by s.

- `s_real` – string that defines a real number (or something whose string representation defines a number)
- `s_imag` – string that defines a real number (or something whose string representation defines a number)
- `pad` – integer at least 0
- `min_prec` – number will have at least this many bits of precision, no matter what

[illegible]

```
>>> from sage.all import *  
>>> ComplexNumber('2.3')  
2.300000000000000  
>>> ComplexNumber('2.3','1.1')  
2.300000000000000 + 1.100000000000000*I  
>>> ComplexNumber(Integer(10))  
10.000000000000000  
>>> ComplexNumber(Integer(10),Integer(10))  
10.000000000000000 + 10.000000000000000*I  
>>> ComplexNumber(RealNumber('1.000000000000000000000000000000'),Integer(2))  
1.0000000000000000000000000000000 + 2.000000000000000000000000000000*I  
>>> ComplexNumber(Integer(1),RealNumber('2.000000000000000000000000'))  
1.0000000000000000000000000 + 2.000000000000000000000000*I
```

```
sage: sage.rings.complex_mpr.create_ComplexNumber(s_real=2,s_imag=1)
2.000000000000000 + 1.000000000000000*I
```

```
>>> from sage.all import *
>>> sage.rings.complex_mpr.create_ComplexNumber(s_real=Integer(2), s_
↪ imag=Integer(1))
2.000000000000000 + 1.000000000000000*I
```

Return `True` if `x` is a complex number. In particular, if `x` is of the `ComplexNumber` type.

```
sage: from sage.rings.complex_mpf_r import is_ComplexNumber
sage: a = ComplexNumber(1, 2); a
1.000000000000000 + 2.000000000000000*I
```

(continues on next page)

(continued from previous page)

```

sage: is_ComplexNumber(a)
doctest:warning...
DeprecationWarning: The function is_ComplexNumber is deprecated;
use 'isinstance(..., ComplexNumber)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
sage: b = ComplexNumber(1); b
1.0000000000000000
sage: is_ComplexNumber(b)
True

```

```

>>> from sage.all import *
>>> from sage.rings.complex_mpfr import is_ComplexNumber
>>> a = ComplexNumber(Integer(1), Integer(2)); a
1.0000000000000000 + 2.0000000000000000*I
>>> is_ComplexNumber(a)
doctest:warning...
DeprecationWarning: The function is_ComplexNumber is deprecated;
use 'isinstance(..., ComplexNumber)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
>>> b = ComplexNumber(Integer(1)); b
1.0000000000000000
>>> is_ComplexNumber(b)
True

```

Note that the global element `I` is a number field element, of type `sage.rings.number_field.number_field_element_quadratic.NumberFieldElement_gaussian`, while elements of the class `ComplexField_class` are of type `ComplexNumber`:

```

sage: # needs sage.symbolic
sage: c = 1 + 2*I
sage: is_ComplexNumber(c)
False
sage: d = CC(1 + 2*I)
sage: is_ComplexNumber(d)
True

```

```

>>> from sage.all import *
>>> # needs sage.symbolic
>>> c = Integer(1) + Integer(2)*I
>>> is_ComplexNumber(c)
False
>>> d = CC(Integer(1) + Integer(2)*I)
>>> is_ComplexNumber(d)
True

```

`sage.rings.complex_mpfr.late_import()`

Import the objects/modules after build (when needed).

`sage.rings.complex_mpfr.make_ComplexNumber0(fld, mult_order, real, imag)`

Create a complex number for pickling.

EXAMPLES:

```
sage: a = CC(1 + I)
sage: loads(dumps(a)) == a # indirect doctest
True
```

```
>>> from sage.all import *
>>> a = CC(Integer(1) + I)
>>> loads(dumps(a)) == a # indirect doctest
True
```

`sage.rings.complex_mpfr.set_global_complex_round_mode(n)`

Set the global complex rounding mode.

Warning

Do not call this function explicitly. The default rounding mode is $n = 0$.

EXAMPLES:

```
sage: sage.rings.complex_mpfr.set_global_complex_round_mode(0)
```

```
>>> from sage.all import *
>>> sage.rings.complex_mpfr.set_global_complex_round_mode(Integer(0))
```

1.3 Arbitrary precision floating point complex numbers using GNU MPC

This is a binding for the arbitrary-precision floating point library [GNU MPC](#).

We define a class `MPCComplexField`, each instance of which specifies a field of floating-point complex numbers with a specified precision shared by the real and imaginary part and a rounding mode stating the rounding mode directions specific to real and imaginary parts.

Individual floating-point numbers are of class `MPCComplexNumber`.

For floating-point representation and rounding mode description see the documentation for the `sage.rings.real_mpfr`.

AUTHORS:

- Philippe Theveny (2008-10-13): initial version, adapted from `sage.rings.real_mpfr` and `sage.rings.complex_mpfr`.
- Alex Ghitza (2008-11): cache, generators, random element, and many doctests.
- Yann Laigle-Chapuy (2010-01): improves compatibility with CC, updates.
- Jeroen Demeyer (2012-02): reformat documentation, make MPC a standard package.
- Travis Scrimshaw (2012-10-18): Added doctests for full coverage.
- Vincent Klein (2017-11-15) : add `__mpc__()` to class `MPCComplexNumber`. `MPCComplexNumber` constructor support `gmpy2.mpz`, `gmpy2.mpq`, `gmpy2.mpfr` and `gmpy2.mpc` parameters.

EXAMPLES:

```
sage: MPC = MPComplexField(42)
sage: a = MPC(12, '15.64E+32'); a
12.0000000000 + 1.56400000000e33*I
sage: a * a * a * a
5.98338564121e132 - 1.83633318912e101*I
sage: a + 1
13.0000000000 + 1.56400000000e33*I
sage: a / 3
4.00000000000 + 5.21333333333e32*I
sage: MPC("infinity + NaN *I")
+infinity + NaN*I
```

```
>>> from sage.all import *
>>> MPC = MPCComplexField(Integer(42))
>>> a = MPC(Integer(12), '15.64E+32'); a
12.0000000000 + 1.564000000000e33*I
>>> a * a * a * a
5.98338564121e132 - 1.83633318912e101*I
>>> a + Integer(1)
13.0000000000 + 1.564000000000e33*I
>>> a / Integer(3)
4.00000000000 + 5.21333333333e32*I
>>> MPC("infinity + NaN *I")
+infinity + NaN*I
```

```
class sage.rings.complex_mpc.CCtoMPC
```

Bases: Map

```
class sage.rings.complex_mpc.INTEGERTOMPC
```

Bases: Map

```
sage.rings.complex_mpc.MPComplexField(prec=53, rnd='RNDNN', names=None)
```

Return the complex field with real and imaginary parts having *prec bits* of precision.

EXAMPLES:

[illegible]

```
>>> from sage.all import *
>>> MPCComplexField()
Complex Field with 53 bits of precision
>>> MPCComplexField(Integer(100))
Complex Field with 100 bits of precision
>>> MPCComplexField(Integer(100)).base_ring()
Real Field with 100 bits of precision
```

(continues on next page)


```
>>> i = MPComplexField(Integer(200)).gen()
>>> i**Integer(2)
-1.0000000000000000000000000000000000000000000000000000000
```

INPUT:

- Rounding mode is of the form 'RND_{xy}' where x and y are the rounding mode for respectively the real and imaginary parts and are one of:

- For example, 'RNDZU' indicates to round the real part towards zero, and the imaginary part towards plus infinity.

EXAMPLES:

```
>>> from sage.all import *
>>> MPCComplexField(Integer(17))
Complex Field with 17 bits of precision
>>> MPCComplexField()
Complex Field with 53 bits of precision
>>> MPCComplexField(Integer(1042), 'RNDDZ')
Complex Field with 1042 bits of precision and rounding RNDDZ
```

ALGORITHMS: Computations are done using the MPC library.

Return 0, since the field of complex numbers has characteristic 0.

EXAMPLES:

117

```
>>> from sage.all import *
>>> MPComplexField(Integer(42)).characteristic()
0
```

gen ($n=0$)

Return the generator of this complex field over its real subfield.

EXAMPLES:

```
sage: MPComplexField(34).gen()
1.00000000*I
```

```
>>> from sage.all import *
>>> MPComplexField(Integer(34)).gen()
1.00000000*I
```

is_exact ()

Return whether or not this field is exact, which is always False.

EXAMPLES:

```
sage: MPComplexField(42).is_exact()
False
```

```
>>> from sage.all import *
>>> MPComplexField(Integer(42)).is_exact()
False
```

name ()

Return the name of the complex field.

EXAMPLES:

```
sage: C = MPComplexField(10, 'RNDNZ'); C.name()
'MPComplexField10_RNDNZ'
```

```
>>> from sage.all import *
>>> C = MPComplexField(Integer(10), 'RNDNZ'); C.name()
'MPComplexField10_RNDNZ'
```

ngens ()

Return 1, the number of generators of this complex field over its real subfield.

EXAMPLES:

```
sage: MPComplexField(34).ngens()
1
```

```
>>> from sage.all import *
>>> MPComplexField(Integer(34)).ngens()
1
```

prec()

Return the precision of this field of complex numbers.

EXAMPLES:

```
sage: MPComplexField().prec()
53
sage: MPComplexField(22).prec()
22
```

```
>>> from sage.all import *
>>> MPComplexField().prec()
53
>>> MPComplexField(Integer(22)).prec()
22
```

random_element(min=0, max=1)

Return a random complex number, uniformly distributed with real and imaginary parts between min and max (default 0 to 1).

EXAMPLES:

```
sage: MPComplexField(100).random_element(-5, 10) # random
1.9305310520925994224072377281 + 0.94745292506956219710477444855*I
sage: MPComplexField(10).random_element() # random
0.12 + 0.23*I
```

```
>>> from sage.all import *
>>> MPComplexField(Integer(100)).random_element(-Integer(5), Integer(10)) # random
1.9305310520925994224072377281 + 0.94745292506956219710477444855*I
>>> MPComplexField(Integer(10)).random_element() # random
0.12 + 0.23*I
```

rounding_mode()

Return rounding modes used for each part of a complex number.

EXAMPLES:

```
sage: MPComplexField().rounding_mode()
'RNDNN'
sage: MPComplexField(rnd='RNDZU').rounding_mode()
'RNDZU'
```

```
>>> from sage.all import *
>>> MPComplexField().rounding_mode()
'RNDNN'
>>> MPComplexField(rnd='RNDZU').rounding_mode()
'RNDZU'
```

rounding_mode_imag()

Return rounding mode used for the imaginary part of complex number.

EXAMPLES:

```
sage: MPCComplexField(rnd='RNDZU').rounding_mode_imag()
'RNDU'
```

```
>>> from sage.all import *
>>> MPCComplexField(rnd='RNDZU').rounding_mode_imag()
'RNDU'
```

rounding_mode_real()

Return rounding mode used for the real part of complex number.

EXAMPLES:

```
sage: MPCComplexField(rnd='RNDZU').rounding_mode_real()
'RNDZ'
```

```
>>> from sage.all import *
>>> MPCComplexField(rnd='RNDZU').rounding_mode_real()
'RNDZ'
```

class sage.rings.complex_mpc.MPCComplexNumber

Bases: `FieldElement`

A floating point approximation to a complex number using any specified precision common to both real and imaginary part.

agm(*right*, *algorithm*='optimal')

Return the algebro-geometric mean of *self* and *right*.

EXAMPLES:

```
sage: MPC = MPCComplexField()
sage: u = MPC(1, 4)
sage: v = MPC(-2, 5)
sage: u.agm(v, algorithm='pari')
-0.410522769709397 + 4.60061063922097*I
sage: u.agm(v, algorithm='principal')
1.24010691168158 - 0.472193567796433*I
sage: u.agm(v, algorithm='optimal')
-0.410522769709397 + 4.60061063922097*I
```

```
>>> from sage.all import *
>>> MPC = MPCComplexField()
>>> u = MPC(Integer(1), Integer(4))
>>> v = MPC(-Integer(2), Integer(5))
>>> u.agm(v, algorithm='pari')
-0.410522769709397 + 4.60061063922097*I
>>> u.agm(v, algorithm='principal')
1.24010691168158 - 0.472193567796433*I
>>> u.agm(v, algorithm='optimal')
-0.410522769709397 + 4.60061063922097*I
```

algdep(*n*, ***kws*)

Return an irreducible polynomial of degree at most *n* which is approximately satisfied by this complex number.

ALGORITHM: Uses the PARI C-library `pari:algdep` command.

INPUT: Type `algebraic_dependency?` at the top level prompt.

All additional parameters are passed onto the top-level `algebraic_dependency` command.

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: z = (1/2)*(1 + sqrt(3.0) * MPC.0); z
0.5000000000000000 + 0.866025403784439*I
sage: p = z.algebraic_dependency(5)
sage: p
x^2 - x + 1
sage: p(z)
1.11022302462516e-16
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> z = (Integer(1)/Integer(2))*(Integer(1) + sqrt(RealNumber('3.0')) * MPC.
↳gen(0)); z
0.5000000000000000 + 0.866025403784439*I
>>> p = z.algebraic_dependency(Integer(5))
>>> p
x^2 - x + 1
>>> p(z)
1.11022302462516e-16
```

`algebraic_dependency` (*n*, ***kws*)

Return an irreducible polynomial of degree at most *n* which is approximately satisfied by this complex number.

ALGORITHM: Uses the PARI C-library `pari:algdep` command.

INPUT: Type `algebraic_dependency?` at the top level prompt.

All additional parameters are passed onto the top-level `algebraic_dependency` command.

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: z = (1/2)*(1 + sqrt(3.0) * MPC.0); z
0.5000000000000000 + 0.866025403784439*I
sage: p = z.algebraic_dependency(5)
sage: p
x^2 - x + 1
sage: p(z)
1.11022302462516e-16
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> z = (Integer(1)/Integer(2))*(Integer(1) + sqrt(RealNumber('3.0')) * MPC.
↳gen(0)); z
0.5000000000000000 + 0.866025403784439*I
>>> p = z.algebraic_dependency(Integer(5))
>>> p
```

(continues on next page)

(continued from previous page)

```
x^2 - x + 1
>>> p(z)
1.11022302462516e-16
```

arccos()

Return the arccosine of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: u = MPC(2, 4)
sage: arccos(u)
1.11692611683177 - 2.19857302792094*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> u = MPC(Integer(2), Integer(4))
>>> arccos(u)
1.11692611683177 - 2.19857302792094*I
```

arccosh()

Return the hyperbolic arccos of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: u = MPC(2, 4)
sage: arccosh(u)
2.19857302792094 + 1.11692611683177*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> u = MPC(Integer(2), Integer(4))
>>> arccosh(u)
2.19857302792094 + 1.11692611683177*I
```

arccoth()

Return the hyperbolic arccotangent of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField(100)
sage: MPC(1,1).arccoth()
0.40235947810852509365018983331 - 0.55357435889704525150853273009*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField(Integer(100))
>>> MPC(Integer(1), Integer(1)).arccoth()
0.40235947810852509365018983331 - 0.55357435889704525150853273009*I
```

arccsch()

Return the hyperbolic arcsine of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField(100)
sage: MPC(1,1).arccsch()
0.53063753095251782601650945811 - 0.45227844715119068206365839783*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField(Integer(100))
>>> MPC(Integer(1), Integer(1)).arccsch()
0.53063753095251782601650945811 - 0.45227844715119068206365839783*I
```

arcsech()

Return the hyperbolic arcsecant of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField(100)
sage: MPC(1,1).arcsech()
0.53063753095251782601650945811 - 1.1185178796437059371676632938*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField(Integer(100))
>>> MPC(Integer(1), Integer(1)).arcsech()
0.53063753095251782601650945811 - 1.1185178796437059371676632938*I
```

arcsin()

Return the arcsine of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: u = MPC(2, 4)
sage: arcsin(u)
0.453870209963122 + 2.19857302792094*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> u = MPC(Integer(2), Integer(4))
>>> arcsin(u)
0.453870209963122 + 2.19857302792094*I
```

arcsinh()

Return the hyperbolic arcsine of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: u = MPC(2, 4)
sage: arcsinh(u)
2.18358521656456 + 1.09692154883014*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> u = MPC(Integer(2), Integer(4))
>>> arcsinh(u)
2.18358521656456 + 1.09692154883014*I
```

arctan()

Return the arctangent of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: u = MPC(-2, 4)
sage: arctan(u)
-1.46704821357730 + 0.200586618131234*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> u = MPC(-Integer(2), Integer(4))
>>> arctan(u)
-1.46704821357730 + 0.200586618131234*I
```

arctanh()

Return the hyperbolic arctangent of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: u = MPC(2, 4)
sage: arctanh(u)
0.0964156202029962 + 1.37153510396169*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> u = MPC(Integer(2), Integer(4))
>>> arctanh(u)
0.0964156202029962 + 1.37153510396169*I
```

argument()

The argument (angle) of the complex number, normalized so that $-\pi < \theta \leq \pi$.

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: i = MPC.0
sage: (i^2).argument()
3.14159265358979
sage: (1+i).argument()
0.785398163397448
sage: i.argument()
1.57079632679490
sage: (-i).argument()
-1.57079632679490
sage: (RR('-0.001') - i).argument()
-1.57179632646156
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> i = MPC.gen(0)
>>> (i**Integer(2)).argument()
```

(continues on next page)

(continued from previous page)

```

3.14159265358979
>>> (Integer(1)+i).argument()
0.785398163397448
>>> i.argument()
1.57079632679490
>>> (-i).argument()
-1.57079632679490
>>> (RR('-0.001') - i).argument()
-1.57179632646156

```

conjugate()

Return the complex conjugate of this complex number:

$$\text{conjugate}(a + ib) = a - ib.$$

EXAMPLES:

```

sage: MPC = MPComplexField()
sage: i = MPC(0, 1)
sage: (1+i).conjugate()
1.000000000000000 - 1.000000000000000*I

```

```

>>> from sage.all import *
>>> MPC = MPComplexField()
>>> i = MPC(Integer(0), Integer(1))
>>> (Integer(1)+i).conjugate()
1.000000000000000 - 1.000000000000000*I

```

cos()

Return the cosine of this complex number:

$$\cos(a + ib) = \cos a \cosh b - i \sin a \sinh b.$$

EXAMPLES:

```

sage: MPC = MPComplexField()
sage: u = MPC(2, 4)
sage: cos(u)
-11.3642347064011 - 24.8146514856342*I

```

```

>>> from sage.all import *
>>> MPC = MPComplexField()
>>> u = MPC(Integer(2), Integer(4))
>>> cos(u)
-11.3642347064011 - 24.8146514856342*I

```

cosh()

Return the hyperbolic cosine of this complex number:

$$\cosh(a + ib) = \cosh a \cos b + i \sinh a \sin b.$$

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: u = MPC(2, 4)
sage: cosh(u)
-2.45913521391738 - 2.74481700679215*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> u = MPC(Integer(2), Integer(4))
>>> cosh(u)
-2.45913521391738 - 2.74481700679215*I
```

cot()

Return the cotangent of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField(53)
sage: (1+MPC(I)).cot()
0.217621561854403 - 0.868014142895925*I
sage: i = MPComplexField(200).0
sage: (1+i).cot()
0.21762156185440268136513424360523807352075436916785404091068 - 0.
↪86801414289592494863584920891627388827343874994609327121115*I
sage: i = MPComplexField(220).0
sage: (1+i).cot()
0.21762156185440268136513424360523807352075436916785404091068124239 - 0.
↪86801414289592494863584920891627388827343874994609327121115071646*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField(Integer(53))
>>> (Integer(1)+MPC(I)).cot()
0.217621561854403 - 0.868014142895925*I
>>> i = MPComplexField(Integer(200)).gen(0)
>>> (Integer(1)+i).cot()
0.21762156185440268136513424360523807352075436916785404091068 - 0.
↪86801414289592494863584920891627388827343874994609327121115*I
>>> i = MPComplexField(Integer(220)).gen(0)
>>> (Integer(1)+i).cot()
0.21762156185440268136513424360523807352075436916785404091068124239 - 0.
↪86801414289592494863584920891627388827343874994609327121115071646*I
```

coth()

Return the hyperbolic cotangent of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField(100)
sage: MPC(1,1).coth()
0.86801414289592494863584920892 - 0.21762156185440268136513424361*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField(Integer(100))
```

(continues on next page)

(continued from previous page)

```
>>> MPC(Integer(1), Integer(1)).coth()
0.86801414289592494863584920892 - 0.21762156185440268136513424361*I
```

csc()

Return the cosecant of this complex number.

EXAMPLES:

```
sage: MPC = MPCComplexField(100)
sage: MPC(1,1).csc()
0.62151801717042842123490780586 - 0.30393100162842645033448560451*I
```

```
>>> from sage.all import *
>>> MPC = MPCComplexField(Integer(100))
>>> MPC(Integer(1), Integer(1)).csc()
0.62151801717042842123490780586 - 0.30393100162842645033448560451*I
```

csch()

Return the hyperbolic cosecant of this complex number.

EXAMPLES:

```
sage: MPC = MPCComplexField(100)
sage: MPC(1,1).csch()
0.30393100162842645033448560451 - 0.62151801717042842123490780586*I
```

```
>>> from sage.all import *
>>> MPC = MPCComplexField(Integer(100))
>>> MPC(Integer(1), Integer(1)).csch()
0.30393100162842645033448560451 - 0.62151801717042842123490780586*I
```

dilog()Return the complex dilogarithm of `self`.

The complex dilogarithm, or Spence's function, is defined by

$$Li_2(z) = - \int_0^z \frac{\log|1-\zeta|}{\zeta} d(\zeta) = \sum_{k=1}^{\infty} \frac{z^k}{k^2}.$$

Note that the series definition can only be used for $|z| < 1$.

EXAMPLES:

```
sage: MPC = MPCComplexField()
sage: a = MPC(1,0)
sage: a.dilog()
1.64493406684823
sage: float(pi^2/6)
1.6449340668482262
```

#

```
>>> from sage.all import *
>>> MPC = MPCComplexField()
```

(continues on next page)

(continued from previous page)

```
>>> a = MPC(Integer(1), Integer(0))
>>> a.dilog()
1.64493406684823
>>> float(pi**Integer(2)/Integer(6))
↩ # needs sage.symbolic
1.6449340668482262
```

```
sage: b = MPC(0, 1)
sage: b.dilog()
-0.205616758356028 + 0.915965594177219*I
```

```
>>> from sage.all import *
>>> b = MPC(Integer(0), Integer(1))
>>> b.dilog()
-0.205616758356028 + 0.915965594177219*I
```

```
sage: c = MPC(0, 0)
sage: c.dilog()
0
```

```
>>> from sage.all import *
>>> c = MPC(Integer(0), Integer(0))
>>> c.dilog()
0
```

eta (*omit_frac=False*)Return the value of the Dedekind η function on `self`, intelligently computed using $\mathrm{SL}(2, \mathbf{Z})$ transformations.The η function is

$$\eta(z) = e^{\pi iz/12} \prod_{n=1}^{\infty} (1 - e^{2\pi inz})$$

INPUT:

- `self` – element of the upper half plane (if not, raises a `ValueError`)
- `omit_frac` – boolean (default: `False`); if `True`, omit the $e^{\pi iz/12}$ factor

OUTPUT: a complex number

ALGORITHM: Uses the PARI C library.

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: i = MPC.0
sage: z = 1+i; z.eta()
0.742048775836565 + 0.198831370229911*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> i = MPC.gen(0)
>>> z = Integer(1)+i; z.eta()
0.742048775836565 + 0.198831370229911*I
```

exp()

Return the exponential of this complex number:

$$\exp(a + ib) = \exp(a)(\cos b + i \sin b).$$

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: u = MPC(2, 4)
sage: exp(u)
-4.82980938326939 - 5.59205609364098*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> u = MPC(Integer(2), Integer(4))
>>> exp(u)
-4.82980938326939 - 5.59205609364098*I
```

gamma()

Return the Gamma function evaluated at this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField(30)
sage: i = MPC.0
sage: (1+i).gamma()
0.49801567 - 0.15494983*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField(Integer(30))
>>> i = MPC.gen(0)
>>> (Integer(1)+i).gamma()
0.49801567 - 0.15494983*I
```

gamma_inc(t)

Return the incomplete Gamma function evaluated at this complex number.

EXAMPLES:

```
sage: C, i = MPComplexField(30).objgen()
sage: (1+i).gamma_inc(2 + 3*i) # abs tol 2e-10
0.0020969149 - 0.059981914*I
sage: (1+i).gamma_inc(5)
-0.0013781309 + 0.0065198200*I
sage: C(2).gamma_inc(1 + i)
0.70709210 - 0.42035364*I
```

```
>>> from sage.all import *
>>> C, i = MPComplexField(Integer(30)).objgen()
>>> (Integer(1)+i).gamma_inc(Integer(2) + Integer(3)*i) # abs tol 2e-10
0.0020969149 - 0.059981914*I
>>> (Integer(1)+i).gamma_inc(Integer(5))
-0.0013781309 + 0.0065198200*I
>>> C(Integer(2)).gamma_inc(Integer(1) + i)
0.70709210 - 0.42035364*I
```

EXAMPLES:

EXAMPLES:

EXAMPLES:

```
>>> from sage.all import *
>>> C200 = MPCComplexField(Integer(200))
>>> C200(RealNumber('1.23')).is_real()
True
```

(continued from previous page)

```
>>> C200(Integer(1)+i).is_real()
False
```

is_square()

This function always returns true as $\mathbf{C}$ is algebraically closed.

EXAMPLES:

```
sage: C200 = MPComplexField(200)
sage: a = C200(2,1)
sage: a.is_square()
True
```

```
>>> from sage.all import *
>>> C200 = MPComplexField(Integer(200))
>>> a = C200(Integer(2), Integer(1))
>>> a.is_square()
True
```

$\mathbf{C}$ is algebraically closed, hence every element is a square:

```
sage: b = C200(5)
sage: b.is_square()
True
```

```
>>> from sage.all import *
>>> b = C200(Integer(5))
>>> b.is_square()
True
```

log()

Return the logarithm of this complex number with the branch cut on the negative real axis:

$$\log(z) = \log|z| + i \arg(z).$$

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: u = MPC(2, 4)
sage: log(u)
1.49786613677700 + 1.10714871779409*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> u = MPC(Integer(2), Integer(4))
>>> log(u)
1.49786613677700 + 1.10714871779409*I
```

norm()

Return the norm of a complex number, rounded with the rounding mode of the real part. The norm is the square of the absolute value:

$$\text{norm}(a + ib) = a^2 + b^2.$$

OUTPUT:

A floating-point number in the real field of the real part (same precision, same rounding mode).

EXAMPLES:

This indeed acts as the square function when the imaginary component of `self` is equal to zero:

```
sage: MPC = MPComplexField()
sage: a = MPC(2,1)
sage: a.norm()
5.000000000000000
sage: b = MPC(4.2,0)
sage: b.norm()
17.640000000000000
sage: b^2
17.640000000000000
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> a = MPC(Integer(2), Integer(1))
>>> a.norm()
5.000000000000000
>>> b = MPC(RealNumber('4.2'), Integer(0))
>>> b.norm()
17.640000000000000
>>> b**Integer(2)
17.640000000000000
```

nth_root (*n*, *all*=False)

The *n*-th root function.

INPUT:

- *all* – boolean (default: False); if True, return a list of all *n*-th roots

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: a = MPC(27)
sage: a.nth_root(3)
3.000000000000000
sage: a.nth_root(3, all=True)
[3.000000000000000, -1.500000000000000 + 2.59807621135332*I, -1.500000000000000 -
↪ 2.59807621135332*I]
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> a = MPC(Integer(27))
>>> a.nth_root(Integer(3))
3.000000000000000
>>> a.nth_root(Integer(3), all=True)
[3.000000000000000, -1.500000000000000 + 2.59807621135332*I, -1.500000000000000 -
↪ 2.59807621135332*I]
```

prec ()

EXAMPLES:

```
sage: i = MPCComplexField(2000).0
sage: i.prec()
2000
```

```
>>> from sage.all import *
>>> i = MPComplexField(Integer(2000)).gen(0)
>>> i.prec()
2000
```

real()

Return the real part of `self`.

EXAMPLES:

[illegible][illegible]

sec()

Return the secant of this complex number.

EXAMPLES:

```
sage: MPC = MPCComplexField(100)
sage: MPC(1, 1).sec()
0.49833703055518678521380589177 + 0.59108384172104504805039169297*I
```

```
>>> from sage.all import *
>>> MPC = MPCComplexField(Integer(100))
>>> MPC(Integer(1),Integer(1)).sec()
0.49833703055518678521380589177 + 0.59108384172104504805039169297*I
```

sech()

Return the hyperbolic secant of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField(100)
sage: MPC(1,1).sech()
0.49833703055518678521380589177 - 0.59108384172104504805039169297*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField(Integer(100))
>>> MPC(Integer(1), Integer(1)).sech()
0.49833703055518678521380589177 - 0.59108384172104504805039169297*I
```

sin()

Return the sine of this complex number:

$$\sin(a + ib) = \sin a \cosh b + i \cos a \sinh b.$$

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: u = MPC(2, 4)
sage: sin(u)
24.8313058489464 - 11.3566127112182*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> u = MPC(Integer(2), Integer(4))
>>> sin(u)
24.8313058489464 - 11.3566127112182*I
```

sinh()

Return the hyperbolic sine of this complex number:

$$\sinh(a + ib) = \sinh a \cos b + i \cosh a \sin b.$$

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: u = MPC(2, 4)
sage: sinh(u)
-2.37067416935200 - 2.84723908684883*I
```

```
>>> from sage.all import *
>>> MPC = MPComplexField()
>>> u = MPC(Integer(2), Integer(4))
>>> sinh(u)
-2.37067416935200 - 2.84723908684883*I
```

sqr()

Return the square of a complex number:

$$(a + ib)^2 = (a^2 - b^2) + 2iab.$$

EXAMPLES:

$$\sqrt{z} = \sqrt{|z|}(\cos(\arg(z)/2) + i \sin(\arg(z)/2)).$$

EXAMPLES:

```
sage: C = MPComplexField()
sage: a = C(24, 10)
sage: a.sqrt()
5.000000000000000 + 1.000000000000000*I
```

```
>>> from sage.all import *
>>> C = MPCComplexField()
>>> a = C(Integer(24), Integer(10))
>>> a.sqrt()
5.000000000000000 + 1.000000000000000*I
```

```
str (base=10, **kws)
```

Return a string of `self`.

INPUT:

- `base` – (default: 10) base for output
- `**kwargs` – other arguments to pass to the `str()` method of the real numbers in the real and imaginary parts

EXAMPLES:

[illegible]

```

>>> from sage.all import *
>>> MPC = MPComplexField(Integer(64))
>>> z = MPC(-Integer(4), Integer(3))/Integer(7)
>>> z.str()
'-0.571428571428571428564 + 0.428571428571428571436*I'
>>> z.str(Integer(16))
'-0.92492492492492490 + 0.6db6db6db6db6db70*I'
>>> z.str(truncate=True)
'-0.571428571428571429 + 0.428571428571428571*I'
>>> z.str(Integer(2))
'-0.1001001001001001001001001001001001001001001001001001001001001001001 + 0.
0110110110110110110110110110110110110110110110110110110110110110110*I'

```

tan()

Return the tangent of this complex number:

$$\tan(a + ib) = (\sin 2a + i \sinh 2b) / (\cos 2a + \cosh 2b).$$

EXAMPLES:

```

sage: MPC = MPComplexField()
sage: u = MPC(-2, 4)
sage: tan(u)
0.000507980623470039 + 1.00043851320205*I

```

```

>>> from sage.all import *
>>> MPC = MPComplexField()
>>> u = MPC(-Integer(2), Integer(4))
>>> tan(u)
0.000507980623470039 + 1.00043851320205*I

```

tanh()

Return the hyperbolic tangent of this complex number:

$$\tanh(a + ib) = (\sinh 2a + i \sin 2b) / (\cosh 2a + \cos 2b).$$

EXAMPLES:

```

sage: MPC = MPComplexField()
sage: u = MPC(2, 4)
sage: tanh(u)
1.00468231219024 + 0.0364233692474037*I

```

```

>>> from sage.all import *
>>> MPC = MPComplexField()
>>> u = MPC(Integer(2), Integer(4))
>>> tanh(u)
1.00468231219024 + 0.0364233692474037*I

```

zeta()

Return the Riemann zeta function evaluated at this complex number.

EXAMPLES:

```
sage: i = MPCComplexField(30).gen()
sage: z = 1 + i
sage: z.zeta()
0.58215806 - 0.92684856*I
```

```
>>> from sage.all import *
>>> i = MPCComplexField(Integer(30)).gen()
>>> z = Integer(1) + i
>>> z.zeta()
0.58215806 - 0.92684856*I
```

class sage.rings.complex_mpc.MPCToMPC

Bases: Map

section()

EXAMPLES:

```
sage: from sage.rings.complex_mpc import *
sage: C10 = MPCComplexField(10)
sage: C100 = MPCComplexField(100)
sage: f = MPCToMPC(C100, C10)
sage: f.section()
Generic map:
  From: Complex Field with 10 bits of precision
  To:   Complex Field with 100 bits of precision
```

```
>>> from sage.all import *
>>> from sage.rings.complex_mpc import *
>>> C10 = MPCComplexField(Integer(10))
>>> C100 = MPCComplexField(Integer(100))
>>> f = MPCToMPC(C100, C10)
>>> f.section()
Generic map:
  From: Complex Field with 10 bits of precision
  To:   Complex Field with 100 bits of precision
```

class sage.rings.complex_mpc.MPFRtoMPC

Bases: Map

sage.rings.complex_mpc.late_import()

Import the objects/modules after build (when needed).

sage.rings.complex_mpc.split_complex_string(string, base=10)

Split and return in that order the real and imaginary parts of a complex in a string.

This is an internal function.

EXAMPLES:

```
sage: sage.rings.complex_mpc.split_complex_string('123.456e789')
('123.456e789', None)
sage: sage.rings.complex_mpc.split_complex_string('123.456e789*I')
(None, '123.456e789')
sage: sage.rings.complex_mpc.split_complex_string('123.+456e789*I')
```

(continues on next page)

(continued from previous page)

```
('123.', '+456e789')
sage: sage.rings.complex_mpc.split_complex_string('123.456e789', base=2)
(None, None)
```

```
>>> from sage.all import *
>>> sage.rings.complex_mpc.split_complex_string('123.456e789')
('123.456e789', None)
>>> sage.rings.complex_mpc.split_complex_string('123.456e789*I')
(None, '123.456e789')
>>> sage.rings.complex_mpc.split_complex_string('123.+456e789*I')
('123.', '+456e789')
>>> sage.rings.complex_mpc.split_complex_string('123.456e789', base=Integer(2))
(None, None)
```

1.4 Double precision floating point real numbers

EXAMPLES:

We create the real double vector space of dimension 3:

```
sage: V = RDF^3; V #_
↳needs sage.modules
Vector space of dimension 3 over Real Double Field
```

```
>>> from sage.all import *
>>> V = RDF**Integer(3); V
↳ # needs sage.modules
Vector space of dimension 3 over Real Double Field
```

Notice that this space is unique:

```
sage: V is RDF^3 #_
↳needs sage.modules
True
sage: V is FreeModule(RDF, 3) #_
↳needs sage.modules
True
sage: V is VectorSpace(RDF, 3) #_
↳needs sage.modules
True
```

```
>>> from sage.all import *
>>> V is RDF**Integer(3)
↳ # needs sage.modules
True
>>> V is FreeModule(RDF, Integer(3))
↳ # needs sage.modules
True
>>> V is VectorSpace(RDF, Integer(3))
↳ # needs sage.modules
True
```

Also, you can instantly create a space of large dimension:

```
sage: V = RDF^10000 #_
↪needs sage.modules
```

```
>>> from sage.all import *
>>> V = RDF**Integer(10000)
↪ # needs sage.modules
```

class sage.rings.real_double.RealDoubleElement

Bases: FieldElement

An approximation to a real number using double precision floating point numbers. Answers derived from calculations with such approximations may differ from what they would be if those calculations were performed with true real numbers. This is due to the rounding errors inherent to finite precision calculations.

NaN()

Return Not-a-Number NaN.

EXAMPLES:

```
sage: RDF.NaN()
NaN
```

```
>>> from sage.all import *
>>> RDF.NaN()
NaN
```

abs()

Return the absolute value of *self*.

EXAMPLES:

```
sage: RDF(1e10).abs()
10000000000.0
sage: RDF(-1e10).abs()
10000000000.0
```

```
>>> from sage.all import *
>>> RDF(RealNumber('1e10')).abs()
10000000000.0
>>> RDF(-RealNumber('1e10')).abs()
10000000000.0
```

agm(*other*)

Return the arithmetic-geometric mean of *self* and *other*. The arithmetic-geometric mean is the common limit of the sequences u_n and v_n , where u_0 is *self*, v_0 is *other*, u_{n+1} is the arithmetic mean of u_n and v_n , and v_{n+1} is the geometric mean of u_n and v_n . If any operand is negative, the return value is NaN.

EXAMPLES:

```
sage: a = RDF(1.5)
sage: b = RDF(2.3)
sage: a.agm(b)
1.8786484558146697
```

```
>>> from sage.all import *
>>> a = RDF(RealNumber('1.5'))
>>> b = RDF(RealNumber('2.3'))
>>> a.agm(b)
1.8786484558146697
```

The arithmetic-geometric mean always lies between the geometric and arithmetic mean:

```
sage: sqrt(a*b) < a.agm(b) < (a+b)/2
True
```

```
>>> from sage.all import *
>>> sqrt(a*b) < a.agm(b) < (a+b)/Integer(2)
True
```

algdep(n)

Return a polynomial of degree at most n which is approximately satisfied by this number.

Note

The resulting polynomial need not be irreducible, and indeed usually won't be if this number is a good approximation to an algebraic number of degree less than n .

ALGORITHM:

Uses the PARI C-library `pari:algdep` command.

EXAMPLES:

```
sage: r = sqrt(RDF(2)); r
1.4142135623730951
sage: r.algebraic_dependency(5)
↪needs sage.libs.pari
x^2 - 2
```

```
>>> from sage.all import *
>>> r = sqrt(RDF(Integer(2))); r
1.4142135623730951
>>> r.algebraic_dependency(Integer(5))
↪ # needs sage.libs.pari
x^2 - 2
```

algebraic_dependency(n)

Return a polynomial of degree at most n which is approximately satisfied by this number.

Note

The resulting polynomial need not be irreducible, and indeed usually won't be if this number is a good approximation to an algebraic number of degree less than n .

ALGORITHM:

Uses the PARI C-library `pari:algdep` command.

EXAMPLES:

```
sage: r = sqrt(RDF(2)); r
1.4142135623730951
sage: r.algebraic_dependency(5)
↪needs sage.libs.pari
x^2 - 2
```

```
>>> from sage.all import *
>>> r = sqrt(RDF(Integer(2))); r
1.4142135623730951
>>> r.algebraic_dependency(Integer(5))
↪      # needs sage.libs.pari
x^2 - 2
```

as_integer_ratio()

Return a coprime pair of integers (a, b) such that self equals a / b exactly.

EXAMPLES:

```
sage: RDF(0).as_integer_ratio()
(0, 1)
sage: RDF(1/3).as_integer_ratio()
(6004799503160661, 18014398509481984)
sage: RDF(37/16).as_integer_ratio()
(37, 16)
sage: RDF(3^60).as_integer_ratio()
(42391158275216203520420085760, 1)
```

```
>>> from sage.all import *
>>> RDF(Integer(0)).as_integer_ratio()
(0, 1)
>>> RDF(Integer(1)/Integer(3)).as_integer_ratio()
(6004799503160661, 18014398509481984)
>>> RDF(Integer(37)/Integer(16)).as_integer_ratio()
(37, 16)
>>> RDF(Integer(3)**Integer(60)).as_integer_ratio()
(42391158275216203520420085760, 1)
```

ceil()

Return the ceiling of self.

EXAMPLES:

```
sage: RDF(2.99).ceil()
3
sage: RDF(2.00).ceil()
2
sage: RDF(-5/2).ceil()
-2
```

```
>>> from sage.all import *
>>> RDF(RealNumber('2.99')).ceil()
3
>>> RDF(RealNumber('2.00')).ceil()
2
>>> RDF(-Integer(5)/Integer(2)).ceil()
-2
```

ceiling()

Return the ceiling of *self*.

EXAMPLES:

```
sage: RDF(2.99).ceil()
3
sage: RDF(2.00).ceil()
2
sage: RDF(-5/2).ceil()
-2
```

```
>>> from sage.all import *
>>> RDF(RealNumber('2.99')).ceil()
3
>>> RDF(RealNumber('2.00')).ceil()
2
>>> RDF(-Integer(5)/Integer(2)).ceil()
-2
```

conjugate()

Return the complex conjugate of this real number, which is the real number itself.

EXAMPLES:

```
sage: RDF(4).conjugate()
4.0
```

```
>>> from sage.all import *
>>> RDF(Integer(4)).conjugate()
4.0
```

cube_root()

Return the cubic root (defined over the real numbers) of *self*.

EXAMPLES:

```
sage: r = RDF(125.0); r.cube_root()
5.000000000000001
sage: r = RDF(-119.0)
sage: r.cube_root()^3 - r # rel tol 1
-1.4210854715202004e-14
```

```
>>> from sage.all import *
>>> r = RDF(RealNumber('125.0')); r.cube_root()
```

(continues on next page)

(continued from previous page)

```

5.0000000000000001
>>> r = RDF(-RealNumber('119.0'))
>>> r.cube_root()**Integer(3) - r # rel tol 1
-1.4210854715202004e-14

```

floor()

Return the floor of `self`.

EXAMPLES:

```

sage: RDF(2.99).floor()
2
sage: RDF(2.00).floor()
2
sage: RDF(-5/2).floor()
-3

```

```

>>> from sage.all import *
>>> RDF(RealNumber('2.99')).floor()
2
>>> RDF(RealNumber('2.00')).floor()
2
>>> RDF(-Integer(5)/Integer(2)).floor()
-3

```

frac()

Return a real number in $(-1, 1)$. It satisfies the relation: $x = x.\text{trunc}() + x.\text{frac}()$

EXAMPLES:

```

sage: RDF(2.99).frac()
0.9900000000000002
sage: RDF(2.50).frac()
0.5
sage: RDF(-2.79).frac()
-0.79

```

```

>>> from sage.all import *
>>> RDF(RealNumber('2.99')).frac()
0.9900000000000002
>>> RDF(RealNumber('2.50')).frac()
0.5
>>> RDF(-RealNumber('2.79')).frac()
-0.79

```

imag()

Return the imaginary part of this number, which is zero.

EXAMPLES:

```

sage: a = RDF(3)
sage: a.imag()
0.0

```

```
>>> from sage.all import *
>>> a = RDF(Integer(3))
>>> a.imag()
0.0
```

integer_part()

If in decimal this number is written $n.d\text{ef}g$, returns n .

EXAMPLES:

```
sage: r = RDF('-1.6')
sage: a = r.integer_part(); a
-1
sage: type(a)
<class 'sage.rings.integer.Integer'>
sage: r = RDF(0.0/0.0)
sage: a = r.integer_part()
Traceback (most recent call last):
...
TypeError: Attempt to get integer part of NaN
```

```
>>> from sage.all import *
>>> r = RDF('-1.6')
>>> a = r.integer_part(); a
-1
>>> type(a)
<class 'sage.rings.integer.Integer'>
>>> r = RDF(RealNumber('0.0')/RealNumber('0.0'))
>>> a = r.integer_part()
Traceback (most recent call last):
...
TypeError: Attempt to get integer part of NaN
```

is_NaN()

Check if self is NaN.

EXAMPLES:

```
sage: RDF(1).is_NaN()
False
sage: a = RDF(0)/RDF(0)
sage: a.is_NaN()
True
```

```
>>> from sage.all import *
>>> RDF(Integer(1)).is_NaN()
False
>>> a = RDF(Integer(0))/RDF(Integer(0))
>>> a.is_NaN()
True
```

is_infinity()

Check if self is ∞ .

EXAMPLES:

```
sage: a = RDF(2); b = RDF(0)
sage: (a/b).is_infinity()
True
sage: (b/a).is_infinity()
False
```

```
>>> from sage.all import *
>>> a = RDF(Integer(2)); b = RDF(Integer(0))
>>> (a/b).is_infinity()
True
>>> (b/a).is_infinity()
False
```

is_integer()

Return True if this number is a integer.

EXAMPLES:

```
sage: RDF(3.5).is_integer()
False
sage: RDF(3).is_integer()
True
```

```
>>> from sage.all import *
>>> RDF(RealNumber('3.5')).is_integer()
False
>>> RDF(Integer(3)).is_integer()
True
```

is_negative_infinity()

Check if self is $-\infty$.

EXAMPLES:

```
sage: a = RDF(2)/RDF(0)
sage: a.is_negative_infinity()
False
sage: a = RDF(-3)/RDF(0)
sage: a.is_negative_infinity()
True
```

```
>>> from sage.all import *
>>> a = RDF(Integer(2))/RDF(Integer(0))
>>> a.is_negative_infinity()
False
>>> a = RDF(-Integer(3))/RDF(Integer(0))
>>> a.is_negative_infinity()
True
```

is_positive_infinity()

Check if self is $+\infty$.

EXAMPLES:

```
sage: a = RDF(1)/RDF(0)
sage: a.is_positive_infinity()
True
sage: a = RDF(-1)/RDF(0)
sage: a.is_positive_infinity()
False
```

```
>>> from sage.all import *
>>> a = RDF(Integer(1))/RDF(Integer(0))
>>> a.is_positive_infinity()
True
>>> a = RDF(-Integer(1))/RDF(Integer(0))
>>> a.is_positive_infinity()
False
```

is_square()

Return whether or not this number is a square in this field. For the real numbers, this is `True` if and only if `self` is nonnegative.

EXAMPLES:

```
sage: RDF(3.5).is_square()
True
sage: RDF(0).is_square()
True
sage: RDF(-4).is_square()
False
```

```
>>> from sage.all import *
>>> RDF(RealNumber('3.5')).is_square()
True
>>> RDF(Integer(0)).is_square()
True
>>> RDF(-Integer(4)).is_square()
False
```

multiplicative_order()

Return n such that `selfn == 1`.

Only ± 1 have finite multiplicative order.

EXAMPLES:

```
sage: RDF(1).multiplicative_order()
1
sage: RDF(-1).multiplicative_order()
2
sage: RDF(3).multiplicative_order()
+Infinity
```

```
>>> from sage.all import *
>>> RDF(Integer(1)).multiplicative_order()
1
```

(continues on next page)

(continued from previous page)

```
>>> RDF(-Integer(1)).multiplicative_order()
2
>>> RDF(Integer(3)).multiplicative_order()
+Infinity
```

nan()

Return Not-a-Number NaN.

EXAMPLES:

```
sage: RDF.NaN()
NaN
```

```
>>> from sage.all import *
>>> RDF.NaN()
NaN
```

prec()

Return the precision of this number in bits.

Always returns 53.

EXAMPLES:

```
sage: RDF(0).prec()
53
```

```
>>> from sage.all import *
>>> RDF(Integer(0)).prec()
53
```

real()

Return `self` - we are already real.

EXAMPLES:

```
sage: a = RDF(3)
sage: a.real()
3.0
```

```
>>> from sage.all import *
>>> a = RDF(Integer(3))
>>> a.real()
3.0
```

round()

Round `self` to the nearest integer.

This uses the convention of rounding half to even (i.e., if the fractional part of `self` is 0.5, then it is rounded to the nearest even integer).

EXAMPLES:

```
sage: RDF(0.49).round()
0
sage: a=RDF(0.51).round(); a
1
sage: RDF(0.5).round()
0
sage: RDF(1.5).round()
2
```

```
>>> from sage.all import *
>>> RDF(RealNumber('0.49')).round()
0
>>> a=RDF(RealNumber('0.51')).round(); a
1
>>> RDF(RealNumber('0.5')).round()
0
>>> RDF(RealNumber('1.5')).round()
2
```

sign()

Return -1, 0, or 1 if `self` is negative, zero, or positive; respectively.

EXAMPLES:

```
sage: RDF(-1.5).sign()
-1
sage: RDF(0).sign()
0
sage: RDF(2.5).sign()
1
```

```
>>> from sage.all import *
>>> RDF(-RealNumber('1.5')).sign()
-1
>>> RDF(Integer(0)).sign()
0
>>> RDF(RealNumber('2.5')).sign()
1
```

sign_mantissa_exponent()

Return the sign, mantissa, and exponent of `self`.

In Sage (as in MPFR), floating-point numbers of precision p are of the form $sm2^{e-p}$, where $s \in \{-1, 1\}$, $2^{p-1} \leq m < 2^p$, and $-2^{30} + 1 \leq e \leq 2^{30} - 1$; plus the special values `+0`, `-0`, `+infinity`, `-infinity`, and `NaN` (which stands for Not-a-Number).

This function returns s , m , and $e - p$. For the special values:

- `+0` returns $(1, 0, 0)$
- `-0` returns $(-1, 0, 0)$
- the return values for `+infinity`, `-infinity`, and `NaN` are not specified.

EXAMPLES:


```

sage: # needs sage.symbolic
sage: a = RDF(exp(1.0)); a
2.718281828459045
sage: sign, mantissa, exponent = RDF(exp(1.0)).sign_mantissa_exponent()
sage: sign, mantissa, exponent
(1, 6121026514868073, -51)
sage: sign*mantissa*(2**exponent) == a
True

```

```

>>> from sage.all import *
>>> # needs sage.symbolic
>>> a = RDF(exp(RealNumber('1.0'))); a
2.718281828459045
>>> sign, mantissa, exponent = RDF(exp(RealNumber('1.0'))).sign_mantissa_
↳exponent()
>>> sign, mantissa, exponent
(1, 6121026514868073, -51)
>>> sign*mantissa*(Integer(2)**exponent) == a
True

```

The mantissa is always a nonnegative number:

```

sage: RDF(-1).sign_mantissa_exponent()
↳needs sage.rings.real_mpfr
(-1, 4503599627370496, -52)

```

```

>>> from sage.all import *
>>> RDF(-Integer(1)).sign_mantissa_exponent()
↳ # needs sage.rings.real_mpfr
(-1, 4503599627370496, -52)

```

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

The square root function.

INPUT:

- *extend* – boolean (default: True); if True, return a square root in a complex field if necessary if *self* is negative. Otherwise raise a `ValueError`.
- *all* – boolean (default: False); if True, return a list of all square roots

EXAMPLES:

```

sage: r = RDF(4.0)
sage: r.sqrt()
2.0
sage: r.sqrt()^2 == r
True

```

```

>>> from sage.all import *
>>> r = RDF(RealNumber('4.0'))
>>> r.sqrt()
2.0

```

(continues on next page)

(continued from previous page)

```
>>> r.sqrt()**Integer(2) == r
True
```

```
sage: r = RDF(4344)
sage: r.sqrt()
65.90902821313632
sage: r.sqrt()^2 - r
0.0
```

```
>>> from sage.all import *
>>> r = RDF(Integer(4344))
>>> r.sqrt()
65.90902821313632
>>> r.sqrt()**Integer(2) - r
0.0
```

```
sage: r = RDF(-2.0)
sage: r.sqrt()
↳needs sage.rings.complex_double
1.4142135623730951*I
```

```
>>> from sage.all import *
>>> r = RDF(-RealNumber('2.0'))
>>> r.sqrt()
↳needs sage.rings.complex_double
1.4142135623730951*I
```

```
sage: RDF(2).sqrt(all=True)
[1.4142135623730951, -1.4142135623730951]
sage: RDF(0).sqrt(all=True)
[0.0]
sage: RDF(-2).sqrt(all=True)
↳needs sage.rings.complex_double
[1.4142135623730951*I, -1.4142135623730951*I]
```

```
>>> from sage.all import *
>>> RDF(Integer(2)).sqrt(all=True)
[1.4142135623730951, -1.4142135623730951]
>>> RDF(Integer(0)).sqrt(all=True)
[0.0]
>>> RDF(-Integer(2)).sqrt(all=True)
↳ # needs sage.rings.complex_double
[1.4142135623730951*I, -1.4142135623730951*I]
```

str()Return the informal string representation of `self`.

EXAMPLES:

```
sage: a = RDF('4.5'); a.str()
'4.5'
```

(continues on next page)

(continued from previous page)

```
sage: a = RDF('49203480923840.2923904823048'); a.str()
'49203480923840.29'
sage: a = RDF(1)/RDF(0); a.str()
'+infinity'
sage: a = -RDF(1)/RDF(0); a.str()
'-infinity'
sage: a = RDF(0)/RDF(0); a.str()
'NaN'
```

```
>>> from sage.all import *
>>> a = RDF('4.5'); a.str()
'4.5'
>>> a = RDF('49203480923840.2923904823048'); a.str()
'49203480923840.29'
>>> a = RDF(Integer(1))/RDF(Integer(0)); a.str()
'+infinity'
>>> a = -RDF(Integer(1))/RDF(Integer(0)); a.str()
'-infinity'
>>> a = RDF(Integer(0))/RDF(Integer(0)); a.str()
'NaN'
```

We verify consistency with RR (mpfr reals):

```
sage: str(RR(RDF(1)/RDF(0))) == str(RDF(1)/RDF(0))
True
sage: str(RR(-RDF(1)/RDF(0))) == str(-RDF(1)/RDF(0))
True
sage: str(RR(RDF(0)/RDF(0))) == str(RDF(0)/RDF(0))
True
```

```
>>> from sage.all import *
>>> str(RR(RDF(Integer(1))/RDF(Integer(0)))) == str(RDF(Integer(1))/
↳ RDF(Integer(0)))
True
>>> str(RR(-RDF(Integer(1))/RDF(Integer(0)))) == str(-RDF(Integer(1))/
↳ RDF(Integer(0)))
True
>>> str(RR(RDF(Integer(0))/RDF(Integer(0)))) == str(RDF(Integer(0))/
↳ RDF(Integer(0)))
True
```

trunc()

Truncates this number (returns integer part).

EXAMPLES:

```
sage: RDF(2.99).trunc()
2
sage: RDF(-2.00).trunc()
-2
sage: RDF(0.00).trunc()
0
```

```
>>> from sage.all import *
>>> RDF(RealNumber('2.99')).trunc()
2
>>> RDF(-RealNumber('2.00')).trunc()
-2
>>> RDF(RealNumber('0.00')).trunc()
0
```

ulp()

Return the unit of least precision of `self`, which is the weight of the least significant bit of `self`. This is always a strictly positive number. It is also the gap between this number and the closest number with larger absolute value that can be represented.

EXAMPLES:

```
sage: a = RDF(pi) #_
↪needs sage.symbolic
sage: a.ulp() #_
↪needs sage.symbolic
4.440892098500626e-16
sage: b = a + a.ulp() #_
↪needs sage.symbolic
```

```
>>> from sage.all import *
>>> a = RDF(pi) #_
↪needs sage.symbolic
>>> a.ulp() #_
↪needs sage.symbolic
4.440892098500626e-16
>>> b = a + a.ulp() #_
↪needs sage.symbolic
```

Adding or subtracting an ulp always gives a different number:

```
sage: # needs sage.symbolic
sage: a + a.ulp() == a
False
sage: a - a.ulp() == a
False
sage: b + b.ulp() == b
False
sage: b - b.ulp() == b
False
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> a + a.ulp() == a
False
>>> a - a.ulp() == a
False
>>> b + b.ulp() == b
False
```

(continues on next page)

(continued from previous page)

```
>>> b - b.ulp() == b
False
```

Since the default rounding mode is round-to-nearest, adding or subtracting something less than half an ulp always gives the same number, unless the result has a smaller ulp. The latter can only happen if the input number is (up to sign) exactly a power of 2:

```
sage: # needs sage.symbolic
sage: a - a.ulp()/3 == a
True
sage: a + a.ulp()/3 == a
True
sage: b - b.ulp()/3 == b
True
sage: b + b.ulp()/3 == b
True

sage: c = RDF(1)
sage: c - c.ulp()/3 == c
False
sage: c.ulp()
2.220446049250313e-16
sage: (c - c.ulp()).ulp()
1.1102230246251565e-16
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> a - a.ulp()/Integer(3) == a
True
>>> a + a.ulp()/Integer(3) == a
True
>>> b - b.ulp()/Integer(3) == b
True
>>> b + b.ulp()/Integer(3) == b
True

>>> c = RDF(Integer(1))
>>> c - c.ulp()/Integer(3) == c
False
>>> c.ulp()
2.220446049250313e-16
>>> (c - c.ulp()).ulp()
1.1102230246251565e-16
```

The ulp is always positive:

```
sage: RDF(-1).ulp()
2.220446049250313e-16
```

```
>>> from sage.all import *
>>> RDF(-Integer(1)).ulp()
2.220446049250313e-16
```

The ulp of zero is the smallest positive number in RDF:

```
sage: RDF(0).ulp()
5e-324
sage: RDF(0).ulp()/2
0.0
```

```
>>> from sage.all import *
>>> RDF(Integer(0)).ulp()
5e-324
>>> RDF(Integer(0)).ulp()/Integer(2)
0.0
```

Some special values:

```
sage: a = RDF(1)/RDF(0); a
+infinity
sage: a.ulp()
+infinity
sage: (-a).ulp()
+infinity
sage: a = RDF('nan')
sage: a.ulp() is a
True
```

```
>>> from sage.all import *
>>> a = RDF(Integer(1))/RDF(Integer(0)); a
+infinity
>>> a.ulp()
+infinity
>>> (-a).ulp()
+infinity
>>> a = RDF('nan')
>>> a.ulp() is a
True
```

The ulp method works correctly with small numbers:

```
sage: u = RDF(0).ulp()
sage: u.ulp() == u
True
sage: x = u * (2^52-1) # largest denormal number
sage: x.ulp() == u
True
sage: x = u * 2^52 # smallest normal number
sage: x.ulp() == u
True
```

```
>>> from sage.all import *
>>> u = RDF(Integer(0)).ulp()
>>> u.ulp() == u
True
>>> x = u * (Integer(2)**Integer(52)-Integer(1)) # largest denormal number
```

(continues on next page)

(continued from previous page)

```
>>> x.ulp() == u
True
>>> x = u * Integer(2)**Integer(52)  # smallest normal number
>>> x.ulp() == u
True
```

`sage.rings.real_double.RealDoubleField()`

Return the unique instance of the *real double field*.

EXAMPLES:

```
sage: RealDoubleField() is RealDoubleField()
True
```

```
>>> from sage.all import *
>>> RealDoubleField() is RealDoubleField()
True
```

class `sage.rings.real_double.RealDoubleField_class`

Bases: `RealDoubleField`

An approximation to the field of real numbers using double precision floating point numbers. Answers derived from calculations in this approximation may differ from what they would be if those calculations were performed in the true field of real numbers. This is due to the rounding errors inherent to finite precision calculations.

EXAMPLES:

```
sage: RR == RDF                                     #␣
↪needs sage.rings.real_mpfr
False
sage: RDF == RealDoubleField()  # RDF is the shorthand
True
```

```
>>> from sage.all import *
>>> RR == RDF                                     #␣
↪needs sage.rings.real_mpfr
False
>>> RDF == RealDoubleField()  # RDF is the shorthand
True
```

```
sage: RDF(1)
1.0
sage: RDF(2/3)
0.6666666666666666
```

```
>>> from sage.all import *
>>> RDF(Integer(1))
1.0
>>> RDF(Integer(2)/Integer(3))
0.6666666666666666
```

A `TypeError` is raised if the coercion doesn't make sense:

```
sage: RDF(QQ['x'].0)
Traceback (most recent call last):
...
TypeError: cannot convert nonconstant polynomial
sage: RDF(QQ['x'](3))
3.0
```

```
>>> from sage.all import *
>>> RDF(QQ['x'].gen(0))
Traceback (most recent call last):
...
TypeError: cannot convert nonconstant polynomial
>>> RDF(QQ['x'](Integer(3)))
3.0
```

One can convert back and forth between double precision real numbers and higher-precision ones, though of course there may be loss of precision:

```
sage: # needs sage.rings.real_mpfr
sage: a = RealField(200)(2).sqrt(); a
1.4142135623730950488016887242096980785696718753769480731767
sage: b = RDF(a); b
1.4142135623730951
sage: a.parent()(b)
1.4142135623730951454746218587388284504413604736328125000000
sage: a.parent()(b) == b
True
sage: b == RR(a)
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.real_mpfr
>>> a = RealField(Integer(200))(Integer(2)).sqrt(); a
1.4142135623730950488016887242096980785696718753769480731767
>>> b = RDF(a); b
1.4142135623730951
>>> a.parent()(b)
1.4142135623730951454746218587388284504413604736328125000000
>>> a.parent()(b) == b
True
>>> b == RR(a)
True
```

NaN()

Return Not-a-Number NaN.

EXAMPLES:

```
sage: RDF.NaN()
NaN
```

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

(continues on next page)

(continued from previous page)

```
>>> RDF.NaN()
NaN
```

algebraic_closure()

Return the algebraic closure of `self`, i.e., the complex double field.

EXAMPLES:

```
sage: RDF.algebraic_closure()
↪needs sage.rings.complex_double
Complex Double Field
```

```
>>> from sage.all import *
>>> RDF.algebraic_closure()
↪needs sage.rings.complex_double
Complex Double Field
```

characteristic()

Return 0, since the field of real numbers has characteristic 0.

EXAMPLES:

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

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

complex_field()

Return the complex field with the same precision as `self`, i.e., the complex double field.

EXAMPLES:

```
sage: RDF.complex_field()
↪needs sage.rings.complex_double
Complex Double Field
```

```
>>> from sage.all import *
>>> RDF.complex_field()
↪needs sage.rings.complex_double
Complex Double Field
```

construction()

Return the functorial construction of `self`, namely, completion of the rational numbers with respect to the prime at ∞ .

Also preserves other information that makes this field unique (i.e. the Real Double Field).

EXAMPLES:

```
sage: c, S = RDF.construction(); S
Rational Field
```

(continues on next page)

(continued from previous page)

```
sage: RDF == c(S)
True
```

```
>>> from sage.all import *
>>> c, S = RDF.construction(); S
Rational Field
>>> RDF == c(S)
True
```

euler_constant()

Return Euler's gamma constant to double precision.

EXAMPLES:

```
sage: RDF.euler_constant()
0.5772156649015329
```

```
>>> from sage.all import *
>>> RDF.euler_constant()
0.5772156649015329
```

factorial(*n*)

Return the factorial of the integer *n* as a real number.

EXAMPLES:

```
sage: RDF.factorial(100)
9.332621544394415e+157
```

```
>>> from sage.all import *
>>> RDF.factorial(Integer(100))
9.332621544394415e+157
```

gen(*n=0*)

Return the generator of the real double field.

EXAMPLES:

```
sage: RDF.0
1.0
sage: RDF.gens()
(1.0,)
```

```
>>> from sage.all import *
>>> RDF.gen(0)
1.0
>>> RDF.gens()
(1.0,)
```

is_exact()

Return False, because doubles are not exact.

EXAMPLES:

```
sage: RDF.is_exact()
False
```

```
>>> from sage.all import *
>>> RDF.is_exact()
False
```

log2()

Return $\log(2)$ to the precision of this field.

EXAMPLES:

```
sage: RDF.log2()
0.6931471805599453
sage: RDF(2).log()
0.6931471805599453
```

```
>>> from sage.all import *
>>> RDF.log2()
0.6931471805599453
>>> RDF(Integer(2)).log()
0.6931471805599453
```

name()

The name of self.

EXAMPLES:

```
sage: RDF.name()
'RealDoubleField'
```

```
>>> from sage.all import *
>>> RDF.name()
'RealDoubleField'
```

nan()

Return Not-a-Number NaN.

EXAMPLES:

```
sage: RDF.NaN()
NaN
```

```
>>> from sage.all import *
>>> RDF.NaN()
NaN
```

ngens()

Return the number of generators which is always 1.

EXAMPLES:

```
sage: RDF.ngens()
1
```

```
>>> from sage.all import *
>>> RDF.ngens()
1
```

pi()

Return π to double-precision.

EXAMPLES:

```
sage: RDF.pi()
3.141592653589793
sage: RDF.pi().sqrt()/2
0.8862269254527579
```

```
>>> from sage.all import *
>>> RDF.pi()
3.141592653589793
>>> RDF.pi().sqrt()/Integer(2)
0.8862269254527579
```

prec()

Return the precision of this real double field in bits.

Always returns 53.

EXAMPLES:

```
sage: RDF.precision()
53
```

```
>>> from sage.all import *
>>> RDF.precision()
53
```

precision()

Return the precision of this real double field in bits.

Always returns 53.

EXAMPLES:

```
sage: RDF.precision()
53
```

```
>>> from sage.all import *
>>> RDF.precision()
53
```

random_element(min=-1, max=1)

Return a random element of this real double field in the interval $[\min, \max]$.

EXAMPLES:

```
sage: RDF.random_element().parent() is RDF
True
sage: -1 <= RDF.random_element() <= 1
True
sage: 100 <= RDF.random_element(min=100, max=110) <= 110
True
```

```
>>> from sage.all import *
>>> RDF.random_element().parent() is RDF
True
>>> -Integer(1) <= RDF.random_element() <= Integer(1)
True
>>> Integer(100) <= RDF.random_element(min=Integer(100), max=Integer(110)) <= Integer(110)
True
```

to_prec (*prec*)

Return the real field to the specified precision. As doubles have fixed precision, this will only return a real double field if *prec* is exactly 53.

EXAMPLES:

```
sage: RDF.to_prec(52) #_
↳ needs sage.rings.real_mpfr
Real Field with 52 bits of precision
sage: RDF.to_prec(53)
Real Double Field
```

```
>>> from sage.all import *
>>> RDF.to_prec(Integer(52))
↳ # needs sage.rings.real_mpfr
Real Field with 52 bits of precision
>>> RDF.to_prec(Integer(53))
Real Double Field
```

zeta (*n=2*)

Return an *n*-th root of unity in the real field, if one exists, or raise a `ValueError` otherwise.

EXAMPLES:

```
sage: RDF.zeta()
-1.0
sage: RDF.zeta(1)
1.0
sage: RDF.zeta(5)
Traceback (most recent call last):
...
ValueError: No 5th root of unity in self
```

```
>>> from sage.all import *
>>> RDF.zeta()
-1.0
>>> RDF.zeta(Integer(1))
```

(continues on next page)

(continued from previous page)

```

1.0
>>> RDF.zeta(Integer(5))
Traceback (most recent call last):
...
ValueError: No 5th root of unity in self

```

class sage.rings.real_double.**ToRDF**

Bases: `Morphism`

Fast morphism from anything with a `__float__` method to an RDF element.

EXAMPLES:

```

sage: f = RDF.coerce_map_from(ZZ); f
Native morphism:
  From: Integer Ring
  To:   Real Double Field
sage: f(4)
4.0
sage: f = RDF.coerce_map_from(QQ); f
Native morphism:
  From: Rational Field
  To:   Real Double Field
sage: f(1/2)
0.5
sage: f = RDF.coerce_map_from(int); f
Native morphism:
  From: Set of Python objects of class 'int'
  To:   Real Double Field
sage: f(3r)
3.0
sage: f = RDF.coerce_map_from(float); f
Native morphism:
  From: Set of Python objects of class 'float'
  To:   Real Double Field
sage: f(3.5)
3.5

```

```

>>> from sage.all import *
>>> f = RDF.coerce_map_from(ZZ); f
Native morphism:
  From: Integer Ring
  To:   Real Double Field
>>> f(Integer(4))
4.0
>>> f = RDF.coerce_map_from(QQ); f
Native morphism:
  From: Rational Field
  To:   Real Double Field
>>> f(Integer(1)/Integer(2))
0.5
>>> f = RDF.coerce_map_from(int); f
Native morphism:

```

(continues on next page)

(continued from previous page)

```

From: Set of Python objects of class 'int'
To:   Real Double Field
>>> f(3)
3.0
>>> f = RDF.coerce_map_from(float); f
Native morphism:
From: Set of Python objects of class 'float'
To:   Real Double Field
>>> f(RealNumber('3.5'))
3.5

```

`sage.rings.real_double.is_RealDoubleElement(x)`

Check if x is an element of the real double field.

EXAMPLES:

```

sage: from sage.rings.real_double import is_RealDoubleElement
sage: is_RealDoubleElement(RDF(3))
doctest:warning...
DeprecationWarning: The function is_RealDoubleElement is deprecated;
use 'isinstance(..., RealDoubleElement)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
sage: is_RealDoubleElement(RIF(3))                                     #_
↪needs sage.rings.real_interval_field
False

```

```

>>> from sage.all import *
>>> from sage.rings.real_double import is_RealDoubleElement
>>> is_RealDoubleElement(RDF(Integer(3)))
doctest:warning...
DeprecationWarning: The function is_RealDoubleElement is deprecated;
use 'isinstance(..., RealDoubleElement)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
>>> is_RealDoubleElement(RIF(Integer(3)))                             _
↪      # needs sage.rings.real_interval_field
False

```

1.5 Double precision floating point complex numbers

Sage supports arithmetic using double-precision complex numbers. A double-precision complex number is a complex number $x + I*y$ with x, y 64-bit (8 byte) floating point numbers (double precision).

The field `ComplexDoubleField` implements the field of all double-precision complex numbers. You can refer to this field by the shorthand `CDF`. Elements of this field are of type `ComplexDoubleElement`. If x and y are coercible to doubles, you can create a complex double element using `ComplexDoubleElement(x, y)`. You can coerce more general objects z to complex doubles by typing either `ComplexDoubleField(x)` or `CDF(x)`.

EXAMPLES:

```

sage: ComplexDoubleField()
Complex Double Field
sage: CDF
Complex Double Field
sage: type(CDF.0)
<class 'sage.rings.complex_double.ComplexDoubleElement'>
sage: ComplexDoubleElement(sqrt(2), 3)
↳needs sage.symbolic
1.4142135623730951 + 3.0*I
sage: parent(CDF(-2))
Complex Double Field

```

```

>>> from sage.all import *
>>> ComplexDoubleField()
Complex Double Field
>>> CDF
Complex Double Field
>>> type(CDF.gen(0))
<class 'sage.rings.complex_double.ComplexDoubleElement'>
>>> ComplexDoubleElement(sqrt(Integer(2)), Integer(3))
↳ # needs sage.symbolic
1.4142135623730951 + 3.0*I
>>> parent(CDF(-Integer(2)))
Complex Double Field

```

```

sage: CC == CDF
False
sage: CDF is ComplexDoubleField()
True
sage: CDF == ComplexDoubleField()
True

```

```

>>> from sage.all import *
>>> CC == CDF
False
>>> CDF is ComplexDoubleField()
True
>>> CDF == ComplexDoubleField()
True

```

The underlying arithmetic of complex numbers is implemented using functions and macros in [GSL](#) (the GNU Scientific Library), and should be very fast. Also, all standard complex trig functions, log, exponents, etc., are implemented using [GSL](#), and are also robust and fast. Several other special functions, e.g. eta, gamma, incomplete gamma, etc., are implemented using the [PARI C](#) library.

AUTHORS:

- William Stein (2006-09): first version
- Travis Scrimshaw (2012-10-18): Added doctests to get full coverage
- Jeroen Demeyer (2013-02-27): fixed all PARI calls ([Issue #14082](#))
- Vincent Klein (2017-11-15) : add `__mpc__()` to class `ComplexDoubleElement`. `ComplexDoubleElement` constructor support and `gmpy2.mpc` parameter.

class `sage.rings.complex_double.ComplexDoubleElement`

Bases: `FieldElement`

An approximation to a complex number using double precision floating point numbers. Answers derived from calculations with such approximations may differ from what they would be if those calculations were performed with true complex numbers. This is due to the rounding errors inherent to finite precision calculations.

abs()

This function returns the magnitude $|z|$ of the complex number z .

See also

- `norm()`

EXAMPLES:

```
sage: CDF(2,3).abs()
3.605551275463989
```

```
>>> from sage.all import *
>>> CDF(Integer(2), Integer(3)).abs()
3.605551275463989
```

abs2()

This function returns the squared magnitude $|z|^2$ of the complex number z , otherwise known as the complex norm.

See also

- `norm()`

EXAMPLES:

```
sage: CDF(2,3).abs2()
13.0
```

```
>>> from sage.all import *
>>> CDF(Integer(2), Integer(3)).abs2()
13.0
```

agm(right, algorithm='optimal')

Return the Arithmetic-Geometric Mean (AGM) of `self` and `right`.

INPUT:

- `right` – complex; another complex number
- `algorithm` – string (default: 'optimal'); the algorithm to use (see below)

OUTPUT:

(complex) A value of the AGM of `self` and `right`. Note that this is a multi-valued function, and the algorithm used affects the value returned, as follows:

- 'pari': Call the `pari:agm` function from the pari library.
- 'optimal': Use the AGM sequence such that at each stage (a, b) is replaced by $(a_1, b_1) = ((a + b)/2, \pm\sqrt{ab})$ where the sign is chosen so that $|a_1 - b_1| \leq |a_1 + b_1|$, or equivalently $\Re(b_1/a_1) \geq 0$. The resulting limit is maximal among all possible values.
- 'principal': Use the AGM sequence such that at each stage (a, b) is replaced by $(a_1, b_1) = ((a + b)/2, \pm\sqrt{ab})$ where the sign is chosen so that $\Re(b_1/a_1) \geq 0$ (the so-called principal branch of the square root).

See [Wikipedia article Arithmetic-geometric mean](#)

EXAMPLES:

```
sage: i = CDF(I) #_
↪needs sage.symbolic
sage: (1+i).agm(2-i) # rel tol 1e-15 #_
↪needs sage.symbolic
1.6278054848727064 + 0.1368275483973686*I
```

```
>>> from sage.all import *
>>> i = CDF(I) #_
↪needs sage.symbolic
>>> (Integer(1)+i).agm(Integer(2)-i) # rel tol 1e-15 _
↪ # needs sage.symbolic
1.6278054848727064 + 0.1368275483973686*I
```

An example to show that the returned value depends on the algorithm parameter:

```
sage: a = CDF(-0.95, -0.65)
sage: b = CDF(0.683, 0.747)
sage: a.agm(b, algorithm='optimal')
-0.3715916523517613 + 0.31989466020683*I
sage: a.agm(b, algorithm='principal') # rel tol 1e-15
0.33817546298618006 - 0.013532696956540503*I
sage: a.agm(b, algorithm='pari')
-0.37159165235176134 + 0.31989466020683005*I
```

```
>>> from sage.all import *
>>> a = CDF(-RealNumber('0.95'), -RealNumber('0.65'))
>>> b = CDF(RealNumber('0.683'), RealNumber('0.747'))
>>> a.agm(b, algorithm='optimal')
-0.3715916523517613 + 0.31989466020683*I
>>> a.agm(b, algorithm='principal') # rel tol 1e-15
0.33817546298618006 - 0.013532696956540503*I
>>> a.agm(b, algorithm='pari')
-0.37159165235176134 + 0.31989466020683005*I
```

Some degenerate cases:

```
sage: CDF(0).agm(a)
0.0
sage: a.agm(0)
0.0
sage: a.agm(-a)
0.0
```

```

>>> from sage.all import *
>>> CDF(Integer(0)).agm(a)
0.0
>>> a.agm(Integer(0))
0.0
>>> a.agm(-a)
0.0

```

algdep(*n*)

Return a polynomial of degree at most n which is approximately satisfied by this complex number.

Note that the returned polynomial need not be irreducible, and indeed usually will not be if z is a good approximation to an algebraic number of degree less than n .

ALGORITHM: Uses the PARI C-library `pari:algdep` command.

EXAMPLES:

```

sage: z = (1/2)*(1 + RDF(sqrt(3)) * CDF.0); z # abs tol 1e-16 #_
↪needs sage.symbolic
0.5 + 0.8660254037844387*I
sage: p = z.algebraic_dependency(5); p #_
↪needs sage.libs.pari sage.symbolic
x^2 - x + 1
sage: abs(z^2 - z + 1) < 1e-14 #_
↪needs sage.symbolic
True

```

```

>>> from sage.all import *
>>> z = (Integer(1)/Integer(2))*(Integer(1) + RDF(sqrt(Integer(3)))) * CDF.
↪gen(0)); z # abs tol 1e-16 # needs sage.symbolic
0.5 + 0.8660254037844387*I
>>> p = z.algebraic_dependency(Integer(5)); p #_
↪ # needs sage.libs.pari sage.symbolic
x^2 - x + 1
>>> abs(z**Integer(2) - z + Integer(1)) < RealNumber('1e-14') #_
↪ # needs sage.symbolic
True

```

```

sage: CDF(0,2).algebraic_dependency(10) #_
↪needs sage.libs.pari
x^2 + 4
sage: CDF(1,5).algebraic_dependency(2) #_
↪needs sage.libs.pari
x^2 - 2*x + 26

```

```

>>> from sage.all import *
>>> CDF(Integer(0),Integer(2)).algebraic_dependency(Integer(10)) #_
↪ # needs sage.libs.pari
x^2 + 4
>>> CDF(Integer(1),Integer(5)).algebraic_dependency(Integer(2)) #_
↪ # needs sage.libs.pari
x^2 - 2*x + 26

```

algebraic_dependency(*n*)

Return a polynomial of degree at most n which is approximately satisfied by this complex number.

Note that the returned polynomial need not be irreducible, and indeed usually will not be if z is a good approximation to an algebraic number of degree less than n .

ALGORITHM: Uses the PARI C-library `pari:algdep` command.

EXAMPLES:

```
sage: z = (1/2)*(1 + RDF(sqrt(3)) * CDF.0); z      # abs tol 1e-16      #_
↪needs sage.symbolic
0.5 + 0.8660254037844387*I
sage: p = z.algebraic_dependency(5); p          #_
↪needs sage.libs.pari sage.symbolic
x^2 - x + 1
sage: abs(z^2 - z + 1) < 1e-14                #_
↪needs sage.symbolic
True
```

```
>>> from sage.all import *
>>> z = (Integer(1)/Integer(2))*(Integer(1) + RDF(sqrt(Integer(3))) * CDF.
↪gen(0)); z      # abs tol 1e-16      # needs sage.symbolic
0.5 + 0.8660254037844387*I
>>> p = z.algebraic_dependency(Integer(5)); p          _
↪      # needs sage.libs.pari sage.symbolic
x^2 - x + 1
>>> abs(z**Integer(2) - z + Integer(1)) < RealNumber('1e-14')      _
↪      # needs sage.symbolic
True
```

```
sage: CDF(0,2).algebraic_dependency(10)          #_
↪needs sage.libs.pari
x^2 + 4
sage: CDF(1,5).algebraic_dependency(2)          #_
↪needs sage.libs.pari
x^2 - 2*x + 26
```

```
>>> from sage.all import *
>>> CDF(Integer(0),Integer(2)).algebraic_dependency(Integer(10))      _
↪      # needs sage.libs.pari
x^2 + 4
>>> CDF(Integer(1),Integer(5)).algebraic_dependency(Integer(2))      _
↪      # needs sage.libs.pari
x^2 - 2*x + 26
```

arccos()

This function returns the complex arccosine of the complex number z , $\arccos(z)$. The branch cuts are on the real axis, less than -1 and greater than 1 .

EXAMPLES:

```
sage: CDF(1,1).arccos()
0.9045568943023814 - 1.0612750619050357*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).arccos()
0.9045568943023814 - 1.0612750619050357*I
```

arccosh()

This function returns the complex hyperbolic arccosine of the complex number z , $\text{arccosh}(z)$. The branch cut is on the real axis, less than 1.

EXAMPLES:

```
sage: CDF(1,1).arccosh()
1.0612750619050357 + 0.9045568943023814*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).arccosh()
1.0612750619050357 + 0.9045568943023814*I
```

arccot()

This function returns the complex arccotangent of the complex number z , $\text{arccot}(z) = \arctan(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arccot() # rel tol 1e-15
0.5535743588970452 - 0.4023594781085251*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).arccot() # rel tol 1e-15
0.5535743588970452 - 0.4023594781085251*I
```

arccoth()

This function returns the complex hyperbolic arccotangent of the complex number z , $\text{arccoth}(z) = \text{arctanh}(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arccoth() # rel tol 1e-15
0.4023594781085251 - 0.5535743588970452*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).arccoth() # rel tol 1e-15
0.4023594781085251 - 0.5535743588970452*I
```

arccsc()

This function returns the complex arccosecant of the complex number z , $\text{arccsc}(z) = \arcsin(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arccsc() # rel tol 1e-15
0.45227844715119064 - 0.5306375309525178*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).arccsc() # rel tol 1e-15
0.45227844715119064 - 0.5306375309525178*I
```

arccsch()

This function returns the complex hyperbolic arccosecant of the complex number z , $\text{arccsch}(z) = \arcsin(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arccsch() # rel tol 1e-15
0.5306375309525178 - 0.45227844715119064*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).arccsch() # rel tol 1e-15
0.5306375309525178 - 0.45227844715119064*I
```

arcsec()

This function returns the complex arcsecant of the complex number z , $\text{arcsec}(z) = \arccos(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arcsec() # rel tol 1e-15
1.118517879643706 + 0.5306375309525178*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).arcsec() # rel tol 1e-15
1.118517879643706 + 0.5306375309525178*I
```

arcsech()

This function returns the complex hyperbolic arcsecant of the complex number z , $\text{arcsech}(z) = \text{arccosh}(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arcsech() # rel tol 1e-15
0.5306375309525176 - 1.118517879643706*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).arcsech() # rel tol 1e-15
0.5306375309525176 - 1.118517879643706*I
```

arcsin()

This function returns the complex arcsine of the complex number z , $\text{arcsin}(z)$. The branch cuts are on the real axis, less than -1 and greater than 1.

EXAMPLES:

```
sage: CDF(1,1).arcsin()
0.6662394324925152 + 1.0612750619050357*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).arcsin()
0.6662394324925152 + 1.0612750619050357*I
```

arcsinh()

This function returns the complex hyperbolic arcsine of the complex number z , $\text{arcsinh}(z)$. The branch cuts are on the imaginary axis, below $-i$ and above i .

EXAMPLES:

```
sage: CDF(1,1).arcsinh()
1.0612750619050357 + 0.6662394324925152*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1),Integer(1)).arcsinh()
1.0612750619050357 + 0.6662394324925152*I
```

arctan()

This function returns the complex arctangent of the complex number z , $\arctan(z)$. The branch cuts are on the imaginary axis, below $-i$ and above i .

EXAMPLES:

```
sage: CDF(1,1).arctan()
1.0172219678978514 + 0.4023594781085251*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1),Integer(1)).arctan()
1.0172219678978514 + 0.4023594781085251*I
```

arctanh()

This function returns the complex hyperbolic arctangent of the complex number z , $\operatorname{arctanh}(z)$. The branch cuts are on the real axis, less than -1 and greater than 1 .

EXAMPLES:

```
sage: CDF(1,1).arctanh()
0.4023594781085251 + 1.0172219678978514*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1),Integer(1)).arctanh()
0.4023594781085251 + 1.0172219678978514*I
```

arg()

This function returns the argument of `self`, the complex number z , denoted by $\arg(z)$, where $-\pi < \arg(z) \leq \pi$.

EXAMPLES:

```
sage: CDF(1,0).arg()
0.0
sage: CDF(0,1).arg()
1.5707963267948966
sage: CDF(0,-1).arg()
-1.5707963267948966
sage: CDF(-1,0).arg()
3.141592653589793
```

```
>>> from sage.all import *
>>> CDF(Integer(1),Integer(0)).arg()
0.0
>>> CDF(Integer(0),Integer(1)).arg()
```

(continues on next page)

(continued from previous page)

```

1.5707963267948966
>>> CDF(Integer(0), -Integer(1)).arg()
-1.5707963267948966
>>> CDF(-Integer(1), Integer(0)).arg()
3.141592653589793

```

argument()

This function returns the argument of the `self`, the complex number z , in the interval $-\pi < \arg(z) \leq \pi$.

EXAMPLES:

```

sage: CDF(6).argument()
0.0
sage: CDF(i).argument()                                     # _
↪needs sage.symbolic
1.5707963267948966
sage: CDF(-1).argument()
3.141592653589793
sage: CDF(-1 - 0.000001*i).argument()                     # _
↪needs sage.symbolic
-3.1415916535897934

```

```

>>> from sage.all import *
>>> CDF(Integer(6)).argument()
0.0
>>> CDF(i).argument()                                     # _
↪needs sage.symbolic
1.5707963267948966
>>> CDF(-Integer(1)).argument()
3.141592653589793
>>> CDF(-Integer(1) - RealNumber('0.000001')*i).argument()
↪                                     # needs sage.symbolic
-3.1415916535897934

```

conj()

This function returns the complex conjugate of the complex number z :

$$\bar{z} = x - iy.$$

EXAMPLES:

```

sage: z = CDF(2,3); z.conj()
2.0 - 3.0*I

```

```

>>> from sage.all import *
>>> z = CDF(Integer(2), Integer(3)); z.conj()
2.0 - 3.0*I

```

conjugate()

This function returns the complex conjugate of the complex number z :

$$\bar{z} = x - iy.$$

EXAMPLES:


```
sage: z = CDF(2,3); z.conjugate()
2.0 - 3.0*I
```

```
>>> from sage.all import *
>>> z = CDF(Integer(2),Integer(3)); z.conjugate()
2.0 - 3.0*I
```

cos()

This function returns the complex cosine of the complex number z :

$$\cos(z) = \frac{e^{iz} + e^{-iz}}{2}$$

EXAMPLES:

```
sage: CDF(1,1).cos() # abs tol 1e-16
0.8337300251311491 - 0.9888977057628651*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1),Integer(1)).cos() # abs tol 1e-16
0.8337300251311491 - 0.9888977057628651*I
```

cosh()

This function returns the complex hyperbolic cosine of the complex number z :

$$\cosh(z) = \frac{e^z + e^{-z}}{2}.$$

EXAMPLES:

```
sage: CDF(1,1).cosh() # abs tol 1e-16
0.8337300251311491 + 0.9888977057628651*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1),Integer(1)).cosh() # abs tol 1e-16
0.8337300251311491 + 0.9888977057628651*I
```

cot()

This function returns the complex cotangent of the complex number z :

$$\cot(z) = \frac{1}{\tan(z)}.$$

EXAMPLES:

```
sage: CDF(1,1).cot() # rel tol 1e-15
0.21762156185440268 - 0.8680141428959249*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1),Integer(1)).cot() # rel tol 1e-15
0.21762156185440268 - 0.8680141428959249*I
```

coth()

This function returns the complex hyperbolic cotangent of the complex number z :

$$\coth(z) = \frac{1}{\tanh(z)}.$$

EXAMPLES:

```
sage: CDF(1,1).coth() # rel tol 1e-15
0.8680141428959249 - 0.21762156185440268*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).coth() # rel tol 1e-15
0.8680141428959249 - 0.21762156185440268*I
```

csc()

This function returns the complex cosecant of the complex number z :

$$\csc(z) = \frac{1}{\sin(z)}.$$

EXAMPLES:

```
sage: CDF(1,1).csc() # rel tol 1e-15
0.6215180171704284 - 0.30393100162842646*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).csc() # rel tol 1e-15
0.6215180171704284 - 0.30393100162842646*I
```

csch()

This function returns the complex hyperbolic cosecant of the complex number z :

$$\operatorname{csch}(z) = \frac{1}{\sinh(z)}.$$

EXAMPLES:

```
sage: CDF(1,1).csch() # rel tol 1e-15
0.30393100162842646 - 0.6215180171704284*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).csch() # rel tol 1e-15
0.30393100162842646 - 0.6215180171704284*I
```

dilog()

Return the principal branch of the dilogarithm of x , i.e., analytic continuation of the power series

$$\log_2(x) = \sum_{n \geq 1} x^n / n^2.$$

EXAMPLES:

```

sage: CDF(1,2).dilog() #_
↪needs sage.libs.pari
-0.059474798673809476 + 2.0726479717747566*I
sage: CDF(10000000,10000000).dilog() #_
↪needs sage.libs.pari
-134.411774490731 + 38.79396299904504*I

```

```

>>> from sage.all import *
>>> CDF(Integer(1),Integer(2)).dilog() _
↪ # needs sage.libs.pari
-0.059474798673809476 + 2.0726479717747566*I
>>> CDF(Integer(10000000),Integer(10000000)).dilog() _
↪ # needs sage.libs.pari
-134.411774490731 + 38.79396299904504*I

```

eta (*omit_frac=0*)

Return the value of the Dedekind η function on *self*.

INPUT:

- *self* – element of the upper half plane (if not, raises a ValueError)
- *omit_frac* – boolean (default: False); if True, omit the $e^{\pi iz/12}$ factor

OUTPUT: a complex double number

ALGORITHM: Uses the PARI C library.

The η function is

$$\eta(z) = e^{\pi iz/12} \prod_{n=1}^{\infty} (1 - e^{2\pi inz})$$

EXAMPLES:

We compute a few values of *eta*():

```

sage: CDF(0,1).eta() #_
↪needs sage.libs.pari
0.7682254223260566
sage: CDF(1,1).eta() #_
↪needs sage.libs.pari
0.7420487758365647 + 0.1988313702299107*I
sage: CDF(25,1).eta() #_
↪needs sage.libs.pari
0.7420487758365647 + 0.1988313702299107*I

```

```

>>> from sage.all import *
>>> CDF(Integer(0),Integer(1)).eta() _
↪ # needs sage.libs.pari
0.7682254223260566
>>> CDF(Integer(1),Integer(1)).eta() _
↪ # needs sage.libs.pari
0.7420487758365647 + 0.1988313702299107*I
>>> CDF(Integer(25),Integer(1)).eta() _
↪ # needs sage.libs.pari
0.7420487758365647 + 0.1988313702299107*I

```

`eta()` works even if the inputs are large:

```
sage: CDF(0, 10^15).eta()
0.0
sage: CDF(10^15, 0.1).eta() # abs tol 1e-10 #_
↪needs sage.libs.pari
-0.115342592727 - 0.19977923088*I
```

```
>>> from sage.all import *
>>> CDF(Integer(0), Integer(10)**Integer(15)).eta()
0.0
>>> CDF(Integer(10)**Integer(15), RealNumber('0.1')).eta() # abs tol 1e-10 _
↪ # needs sage.libs.pari
-0.115342592727 - 0.19977923088*I
```

We compute a few values of `eta()`, but with the fractional power of e omitted:

```
sage: CDF(0,1).eta(True) #_
↪needs sage.libs.pari
0.9981290699259585
```

```
>>> from sage.all import *
>>> CDF(Integer(0), Integer(1)).eta(True) _
↪ # needs sage.libs.pari
0.9981290699259585
```

We compute `eta()` to low precision directly from the definition:

```
sage: z = CDF(1,1); z.eta() #_
↪needs sage.libs.pari
0.7420487758365647 + 0.1988313702299107*I
sage: i = CDF(0,1); pi = CDF(pi) #_
↪needs sage.symbolic
sage: exp(pi * i * z / 12) * prod(1 - exp(2*pi*i*n*z)) #_
↪needs sage.libs.pari sage.symbolic
....: for n in range(1, 10))
0.7420487758365647 + 0.19883137022991068*I
```

```
>>> from sage.all import *
>>> z = CDF(Integer(1), Integer(1)); z.eta() _
↪ # needs sage.libs.pari
0.7420487758365647 + 0.1988313702299107*I
>>> i = CDF(Integer(0), Integer(1)); pi = CDF(pi) _
↪ # needs sage.symbolic
>>> exp(pi * i * z / Integer(12)) * prod(Integer(1) -_
↪exp(Integer(2)*pi*i*n*z) # needs sage.libs.pari sage.
↪symbolic
....: for n in range(Integer(1), Integer(10)))
0.7420487758365647 + 0.19883137022991068*I
```

The optional argument allows us to omit the fractional part:

```

sage: z.eta(omit_frac=True) #_
↳needs sage.libs.pari
0.9981290699259585
sage: pi = CDF(pi) #_
↳needs sage.symbolic
sage: prod(1 - exp(2*pi*i*n*z) for n in range(1,10)) # abs tol 1e-12 #_
↳needs sage.libs.pari sage.symbolic
0.998129069926 + 4.59084695545e-19*I

```

```

>>> from sage.all import *
>>> z.eta(omit_frac=True) #_
↳needs sage.libs.pari
0.9981290699259585
>>> pi = CDF(pi) #_
↳needs sage.symbolic
>>> prod(Integer(1) - exp(Integer(2)*pi*i*n*z) for n in range(Integer(1),
↳Integer(10))) # abs tol 1e-12 # needs sage.libs.pari sage.symbolic
0.998129069926 + 4.59084695545e-19*I

```

We illustrate what happens when z is not in the upper half plane:

```

sage: z = CDF(1)
sage: z.eta()
Traceback (most recent call last):
...
ValueError: value must be in the upper half plane

```

```

>>> from sage.all import *
>>> z = CDF(Integer(1))
>>> z.eta()
Traceback (most recent call last):
...
ValueError: value must be in the upper half plane

```

You can also use functional notation:

```

sage: z = CDF(1,1)
sage: eta(z) #_
↳needs sage.libs.pari
0.7420487758365647 + 0.1988313702299107*I

```

```

>>> from sage.all import *
>>> z = CDF(Integer(1),Integer(1))
>>> eta(z) #_
↳needs sage.libs.pari
0.7420487758365647 + 0.1988313702299107*I

```

exp()

This function returns the complex exponential of the complex number z , $\exp(z)$.

EXAMPLES:

```
sage: CDF(1,1).exp() # abs tol 4e-16
1.4686939399158851 + 2.2873552871788423*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).exp() # abs tol 4e-16
1.4686939399158851 + 2.2873552871788423*I
```

We numerically verify a famous identity to the precision of a double:

```
sage: z = CDF(0, 2*pi); z #_
↪needs sage.symbolic
6.283185307179586*I
sage: exp(z) # rel tol 1e-4 #_
↪needs sage.symbolic
1.0 - 2.4492935982947064e-16*I
```

```
>>> from sage.all import *
>>> z = CDF(Integer(0), Integer(2)*pi); z #_
↪ # needs sage.symbolic
6.283185307179586*I
>>> exp(z) # rel tol 1e-4 #_
↪needs sage.symbolic
1.0 - 2.4492935982947064e-16*I
```

`gamma()`

Return the gamma function $\Gamma(z)$ evaluated at `self`, the complex number z .

EXAMPLES:

```
sage: # needs sage.libs.pari
sage: CDF(5,0).gamma()
24.0
sage: CDF(1,1).gamma()
0.49801566811835607 - 0.15494982830181067*I
sage: CDF(0).gamma()
Infinity
sage: CDF(-1,0).gamma()
Infinity
```

```
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> CDF(Integer(5), Integer(0)).gamma()
24.0
>>> CDF(Integer(1), Integer(1)).gamma()
0.49801566811835607 - 0.15494982830181067*I
>>> CDF(Integer(0)).gamma()
Infinity
>>> CDF(-Integer(1), Integer(0)).gamma()
Infinity
```

`gamma_inc(t)`

Return the incomplete gamma function evaluated at this complex number.

EXAMPLES:

```

sage: CDF(1,1).gamma_inc(CDF(2,3)) #_
↪needs sage.libs.pari
0.0020969148636468277 - 0.059981913655449706*I
sage: CDF(1,1).gamma_inc(5) #_
↪needs sage.libs.pari
-0.001378130936215849 + 0.006519820023119819*I
sage: CDF(2,0).gamma_inc(CDF(1,1)) #_
↪needs sage.libs.pari
0.7070920963459381 - 0.4203536409598115*I

```

```

>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).gamma_inc(CDF(Integer(2), Integer(3))) #_
↪ # needs sage.libs.pari
0.0020969148636468277 - 0.059981913655449706*I
>>> CDF(Integer(1), Integer(1)).gamma_inc(Integer(5)) #_
↪ # needs sage.libs.pari
-0.001378130936215849 + 0.006519820023119819*I
>>> CDF(Integer(2), Integer(0)).gamma_inc(CDF(Integer(1), Integer(1))) #_
↪ # needs sage.libs.pari
0.7070920963459381 - 0.4203536409598115*I

```

imag()

Return the imaginary part of this complex double.

EXAMPLES:

```

sage: a = CDF(3,-2)
sage: a.imag()
-2.0
sage: a.imag_part()
-2.0

```

```

>>> from sage.all import *
>>> a = CDF(Integer(3), -Integer(2))
>>> a.imag()
-2.0
>>> a.imag_part()
-2.0

```

imag_part()

Return the imaginary part of this complex double.

EXAMPLES:

```

sage: a = CDF(3,-2)
sage: a.imag()
-2.0
sage: a.imag_part()
-2.0

```

```

>>> from sage.all import *
>>> a = CDF(Integer(3), -Integer(2))
>>> a.imag()

```

(continues on next page)

(continued from previous page)

```
-2.0
>>> a.imag_part()
-2.0
```

is_NaN()

Check if self is not-a-number.

EXAMPLES:

```
sage: CDF(1, 2).is_NaN()
False
sage: CDF(NaN).is_NaN()
↳needs sage.symbolic
True
sage: (1/CDF(0, 0)).is_NaN()
True
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(2)).is_NaN()
False
>>> CDF(NaN).is_NaN()
↳needs sage.symbolic
True
>>> (Integer(1)/CDF(Integer(0), Integer(0))).is_NaN()
True
```

is_infinity()

Check if self is ∞ .

EXAMPLES:

```
sage: CDF(1, 2).is_infinity()
False
sage: CDF(0, oo).is_infinity()
True
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(2)).is_infinity()
False
>>> CDF(Integer(0), oo).is_infinity()
True
```

is_integer()

Return True if this number is a integer.

EXAMPLES:

```
sage: CDF(0.5).is_integer()
False
sage: CDF(I).is_integer()
↳needs sage.symbolic
False
```

(continues on next page)

(continued from previous page)

```
sage: CDF(2).is_integer()
True
```

```
>>> from sage.all import *
>>> CDF(RealNumber('0.5')).is_integer()
False
>>> CDF(I).is_integer()
↪needs sage.symbolic
False
>>> CDF(Integer(2)).is_integer()
True
```

is_negative_infinity()Check if self is $-\infty$.

EXAMPLES:

```
sage: CDF(1, 2).is_negative_infinity()
False
sage: CDF(-oo, 0).is_negative_infinity()
True
sage: CDF(0, -oo).is_negative_infinity()
False
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(2)).is_negative_infinity()
False
>>> CDF(-oo, Integer(0)).is_negative_infinity()
True
>>> CDF(Integer(0), -oo).is_negative_infinity()
False
```

is_positive_infinity()Check if self is $+\infty$.

EXAMPLES:

```
sage: CDF(1, 2).is_positive_infinity()
False
sage: CDF(oo, 0).is_positive_infinity()
True
sage: CDF(0, oo).is_positive_infinity()
False
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(2)).is_positive_infinity()
False
>>> CDF(oo, Integer(0)).is_positive_infinity()
True
>>> CDF(Integer(0), oo).is_positive_infinity()
False
```

is_square()

This function always returns `True` as $\mathbf{C}$ is algebraically closed.

EXAMPLES:

```
sage: CDF(-1).is_square()
True
```

```
>>> from sage.all import *
>>> CDF(-Integer(1)).is_square()
True
```

log(base=None)

This function returns the complex natural logarithm to the given base of the complex number z , $\log(z)$. The branch cut is the negative real axis.

INPUT:

- `base` – (default: e) the base of the natural logarithm

EXAMPLES:

```
sage: CDF(1,1).log()
0.34657359027997264 + 0.7853981633974483*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).log()
0.34657359027997264 + 0.7853981633974483*I
```

This is the only example different from the GSL:

```
sage: CDF(0,0).log()
-infinity
```

```
>>> from sage.all import *
>>> CDF(Integer(0), Integer(0)).log()
-infinity
```

log10()

This function returns the complex base-10 logarithm of the complex number z , $\log_{10}(z)$.

The branch cut is the negative real axis.

EXAMPLES:

```
sage: CDF(1,1).log10()
0.15051499783199057 + 0.3410940884604603*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).log10()
0.15051499783199057 + 0.3410940884604603*I
```

log_b(b)

This function returns the complex base- b logarithm of the complex number z , $\log_b(z)$. This quantity is computed as the ratio $\log(z)/\log(b)$.

The branch cut is the negative real axis.

EXAMPLES:

```
sage: CDF(1,1).log_b(10) # rel tol 1e-15
0.15051499783199057 + 0.3410940884604603*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).log_b(Integer(10)) # rel tol 1e-15
0.15051499783199057 + 0.3410940884604603*I
```

logabs()

This function returns the natural logarithm of the magnitude of the complex number z , $\log |z|$.

This allows for an accurate evaluation of $\log |z|$ when $|z|$ is close to 1. The direct evaluation of $\log(\text{abs}(z))$ would lead to a loss of precision in this case.

EXAMPLES:

```
sage: CDF(1.1, 0.1).logabs()
0.09942542937258267
sage: log(abs(CDF(1.1, 0.1)))
0.09942542937258259
```

```
>>> from sage.all import *
>>> CDF(RealNumber('1.1'), RealNumber('0.1')).logabs()
0.09942542937258267
>>> log(abs(CDF(RealNumber('1.1'), RealNumber('0.1'))))
0.09942542937258259
```

```
sage: log(abs(ComplexField(200)(1.1, 0.1)))
0.099425429372582595066319157757531449594489450091985182495705
```

```
>>> from sage.all import *
>>> log(abs(ComplexField(Integer(200))(RealNumber('1.1'), RealNumber('0.1'))))
0.099425429372582595066319157757531449594489450091985182495705
```

norm()

This function returns the squared magnitude $|z|^2$ of the complex number z , otherwise known as the complex norm. If $c = a + bi$ is a complex number, then the norm of c is defined as the product of c and its complex conjugate:

$$\text{norm}(c) = \text{norm}(a + bi) = c \cdot \bar{c} = a^2 + b^2.$$

The norm of a complex number is different from its absolute value. The absolute value of a complex number is defined to be the square root of its norm. A typical use of the complex norm is in the integral domain $\mathbf{Z}[i]$ of Gaussian integers, where the norm of each Gaussian integer $c = a + bi$ is defined as its complex norm.

See also

- [abs\(\)](#)
- [abs2\(\)](#)

- `sage.misc.functional.norm()`
- `sage.rings.complex_mpfr.ComplexNumber.norm()`

EXAMPLES:

```
sage: CDF(2,3).norm()
13.0
```

```
>>> from sage.all import *
>>> CDF(Integer(2), Integer(3)).norm()
13.0
```

nth_root (*n*, *all=False*)

The *n*-th root function.

INPUT:

- *all* – boolean (default: False); if True, return a list of all *n*-th roots

EXAMPLES:

```
sage: a = CDF(125)
sage: a.nth_root(3)
5.000000000000001
sage: a = CDF(10, 2)
sage: [r^5 for r in a.nth_root(5, all=True)] # rel tol 1e-14
[9.999999999999998 + 2.0*I, 9.999999999999993 + 2.000000000000002*I, 9.
↪999999999999996 + 1.9999999999999907*I, 9.999999999999993 + 2.
↪0000000000000004*I, 9.999999999999998 + 1.9999999999999802*I]
sage: abs(sum(a.nth_root(111, all=True))) # rel tol 0.1
1.1057313523818259e-13
```

```
>>> from sage.all import *
>>> a = CDF(Integer(125))
>>> a.nth_root(Integer(3))
5.000000000000001
>>> a = CDF(Integer(10), Integer(2))
>>> [r**Integer(5) for r in a.nth_root(Integer(5), all=True)] # rel tol 1e-14
[9.999999999999998 + 2.0*I, 9.999999999999993 + 2.000000000000002*I, 9.
↪999999999999996 + 1.9999999999999907*I, 9.999999999999993 + 2.
↪0000000000000004*I, 9.999999999999998 + 1.9999999999999802*I]
>>> abs(sum(a.nth_root(Integer(111), all=True))) # rel tol 0.1
1.1057313523818259e-13
```

prec ()

Return the precision of this number (to be more similar to *ComplexNumber*). Always returns 53.

EXAMPLES:

```
sage: CDF(0).prec()
53
```

```
>>> from sage.all import *
>>> CDF(Integer(0)).prec()
53
```

real()

Return the real part of this complex double.

EXAMPLES:

```
sage: a = CDF(3,-2)
sage: a.real()
3.0
sage: a.real_part()
3.0
```

```
>>> from sage.all import *
>>> a = CDF(Integer(3),-Integer(2))
>>> a.real()
3.0
>>> a.real_part()
3.0
```

real_part()

Return the real part of this complex double.

EXAMPLES:

```
sage: a = CDF(3,-2)
sage: a.real()
3.0
sage: a.real_part()
3.0
```

```
>>> from sage.all import *
>>> a = CDF(Integer(3),-Integer(2))
>>> a.real()
3.0
>>> a.real_part()
3.0
```

sec()

This function returns the complex secant of the complex number z :

$$\sec(z) = \frac{1}{\cos(z)}.$$

EXAMPLES:

```
sage: CDF(1,1).sec() # rel tol 1e-15
0.4983370305551868 + 0.591083841721045*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1),Integer(1)).sec() # rel tol 1e-15
0.4983370305551868 + 0.591083841721045*I
```

sech()

This function returns the complex hyperbolic secant of the complex number z :

$$\operatorname{sech}(z) = \frac{1}{\cosh(z)}.$$

EXAMPLES:

```
sage: CDF(1,1).sech() # rel tol 1e-15
0.4983370305551868 - 0.591083841721045*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).sech() # rel tol 1e-15
0.4983370305551868 - 0.591083841721045*I
```

sin()

This function returns the complex sine of the complex number z :

$$\sin(z) = \frac{e^{iz} - e^{-iz}}{2i}.$$

EXAMPLES:

```
sage: CDF(1,1).sin()
1.2984575814159773 + 0.6349639147847361*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).sin()
1.2984575814159773 + 0.6349639147847361*I
```

sinh()

This function returns the complex hyperbolic sine of the complex number z :

$$\sinh(z) = \frac{e^z - e^{-z}}{2}.$$

EXAMPLES:

```
sage: CDF(1,1).sinh()
0.6349639147847361 + 1.2984575814159773*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).sinh()
0.6349639147847361 + 1.2984575814159773*I
```

sqrt(all=False, **kws)

The square root function.

INPUT:

- `all` – boolean (default: `False`); if `True`, return a list of all square roots

If `all` is `False`, the branch cut is the negative real axis. The result always lies in the right half of the complex plane.

EXAMPLES:

We compute several square roots:

```
sage: a = CDF(2,3)
sage: b = a.sqrt(); b # rel tol 1e-15
1.6741492280355401 + 0.8959774761298381*I
sage: b^2 # rel tol 1e-15
2.0 + 3.0*I
sage: a^(1/2) # abs tol 1e-16
1.6741492280355401 + 0.895977476129838*I
```

```
>>> from sage.all import *
>>> a = CDF(Integer(2),Integer(3))
>>> b = a.sqrt(); b # rel tol 1e-15
1.6741492280355401 + 0.8959774761298381*I
>>> b**Integer(2) # rel tol 1e-15
2.0 + 3.0*I
>>> a**(Integer(1)/Integer(2)) # abs tol 1e-16
1.6741492280355401 + 0.895977476129838*I
```

We compute the square root of -1:

```
sage: a = CDF(-1)
sage: a.sqrt()
1.0*I
```

```
>>> from sage.all import *
>>> a = CDF(-Integer(1))
>>> a.sqrt()
1.0*I
```

We compute all square roots:

```
sage: CDF(-2).sqrt(all=True)
[1.4142135623730951*I, -1.4142135623730951*I]
sage: CDF(0).sqrt(all=True)
[0.0]
```

```
>>> from sage.all import *
>>> CDF(-Integer(2)).sqrt(all=True)
[1.4142135623730951*I, -1.4142135623730951*I]
>>> CDF(Integer(0)).sqrt(all=True)
[0.0]
```

tan()

This function returns the complex tangent of the complex number z :

$$\tan(z) = \frac{\sin(z)}{\cos(z)}.$$

EXAMPLES:

```
sage: CDF(1,1).tan()
0.27175258531951174 + 1.0839233273386946*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).tan()
0.27175258531951174 + 1.0839233273386946*I
```

tanh()

This function returns the complex hyperbolic tangent of the complex number z :

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)}.$$

EXAMPLES:

```
sage: CDF(1, 1).tanh()
1.0839233273386946 + 0.27175258531951174*I
```

```
>>> from sage.all import *
>>> CDF(Integer(1), Integer(1)).tanh()
1.0839233273386946 + 0.27175258531951174*I
```

zeta()

Return the Riemann zeta function evaluated at this complex number.

EXAMPLES:

```
sage: z = CDF(1, 1)
sage: z.zeta() #_
↳needs sage.libs.pari
0.5821580597520036 - 0.9268485643308071*I
sage: zeta(z) #_
↳needs sage.libs.pari
0.5821580597520036 - 0.9268485643308071*I
sage: zeta(CDF(1)) #_
↳needs sage.libs.pari
Infinity
```

```
>>> from sage.all import *
>>> z = CDF(Integer(1), Integer(1))
>>> z.zeta() #_
↳needs sage.libs.pari
0.5821580597520036 - 0.9268485643308071*I
>>> zeta(z) #_
↳needs sage.libs.pari
0.5821580597520036 - 0.9268485643308071*I
>>> zeta(CDF(Integer(1))) _
↳ # needs sage.libs.pari
Infinity
```

`sage.rings.complex_double.ComplexDoubleField()`

Return the field of double precision complex numbers.

EXAMPLES:

```
sage: ComplexDoubleField()
Complex Double Field
```

(continues on next page)

(continued from previous page)

```
sage: ComplexDoubleField() is CDF
True
```

```
>>> from sage.all import *
>>> ComplexDoubleField()
Complex Double Field
>>> ComplexDoubleField() is CDF
True
```

```
class sage.rings.complex_double.ComplexDoubleField_class
```

Bases: `ComplexDoubleField`

An approximation to the field of complex numbers using double precision floating point numbers. Answers derived from calculations in this approximation may differ from what they would be if those calculations were performed in the true field of complex numbers. This is due to the rounding errors inherent to finite precision calculations.

ALGORITHM:

Arithmetic is done using GSL (the GNU Scientific Library).

```
algebraic_closure()
```

Return the algebraic closure of `self`, i.e., the complex double field.

EXAMPLES:

```
sage: CDF.algebraic_closure()
Complex Double Field
```

```
>>> from sage.all import *
>>> CDF.algebraic_closure()
Complex Double Field
```

```
characteristic()
```

Return the characteristic of the complex double field, which is 0.

EXAMPLES:

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

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

```
construction()
```

Return the functorial construction of `self`, namely, algebraic closure of the real double field.

EXAMPLES:

```
sage: c, S = CDF.construction(); S
Real Double Field
sage: CDF == c(S)
True
```

```
>>> from sage.all import *
>>> c, S = CDF.construction(); S
Real Double Field
>>> CDF == c(S)
True
```

gen ($n=0$)

Return the generator of the complex double field.

EXAMPLES:

```
sage: CDF.0
1.0*I
sage: CDF.gen(0)
1.0*I
```

```
>>> from sage.all import *
>>> CDF.gen(0)
1.0*I
>>> CDF.gen(Integer(0))
1.0*I
```

is_exact ()

Return whether or not this field is exact, which is always `False`.

EXAMPLES:

```
sage: CDF.is_exact()
False
```

```
>>> from sage.all import *
>>> CDF.is_exact()
False
```

ngens ()

The number of generators of this complex field as an $\mathbf{R}$ -algebra.

There is one generator, namely `sqrt(-1)`.

EXAMPLES:

```
sage: CDF.ngens()
1
```

```
>>> from sage.all import *
>>> CDF.ngens()
1
```

pi ()

Return π as a double precision complex number.

EXAMPLES:

```
sage: CDF.pi()
3.141592653589793
```

```
>>> from sage.all import *
>>> CDF.pi()
3.141592653589793
```

prec()

Return the precision of this complex double field (to be more similar to *ComplexField*). Always returns 53.

EXAMPLES:

```
sage: CDF.prec()
53
```

```
>>> from sage.all import *
>>> CDF.prec()
53
```

precision()

Return the precision of this complex double field (to be more similar to *ComplexField*). Always returns 53.

EXAMPLES:

```
sage: CDF.prec()
53
```

```
>>> from sage.all import *
>>> CDF.prec()
53
```

random_element(xmin=-1, xmax=1, ymin=-1, ymax=1)

Return a random element of this complex double field with real and imaginary part bounded by *xmin*, *xmax*, *ymin*, *ymax*.

EXAMPLES:

```
sage: CDF.random_element().parent() is CDF
True
sage: re, im = CDF.random_element()
sage: -1 <= re <= 1, -1 <= im <= 1
(True, True)
sage: re, im = CDF.random_element(-10,10,-10,10)
sage: -10 <= re <= 10, -10 <= im <= 10
(True, True)
sage: re, im = CDF.random_element(-10^20,10^20,-2,2)
sage: -10^20 <= re <= 10^20, -2 <= im <= 2
(True, True)
```

```
>>> from sage.all import *
>>> CDF.random_element().parent() is CDF
True
>>> re, im = CDF.random_element()
>>> -Integer(1) <= re <= Integer(1), -Integer(1) <= im <= Integer(1)
```

(continues on next page)

(continued from previous page)

```

(True, True)
>>> re, im = CDF.random_element(-Integer(10), Integer(10), -Integer(10),
↳ Integer(10))
>>> -Integer(10) <= re <= Integer(10), -Integer(10) <= im <= Integer(10)
(True, True)
>>> re, im = CDF.random_element(-Integer(10)**Integer(20),
↳ Integer(10)**Integer(20), -Integer(2), Integer(2))
>>> -Integer(10)**Integer(20) <= re <= Integer(10)**Integer(20), -Integer(2)
↳ <= im <= Integer(2)
(True, True)

```

real_double_field()

The real double field, which you may view as a subfield of this complex double field.

EXAMPLES:

```

sage: CDF.real_double_field()
Real Double Field

```

```

>>> from sage.all import *
>>> CDF.real_double_field()
Real Double Field

```

to_prec (prec)

Return the complex field to the specified precision. As doubles have fixed precision, this will only return a complex double field if *prec* is exactly 53.

EXAMPLES:

```

sage: CDF.to_prec(53)
Complex Double Field
sage: CDF.to_prec(250)
Complex Field with 250 bits of precision

```

```

>>> from sage.all import *
>>> CDF.to_prec(Integer(53))
Complex Double Field
>>> CDF.to_prec(Integer(250))
Complex Field with 250 bits of precision

```

zeta (n=2)

Return a primitive *n*-th root of unity in this CDF, for $n \geq 1$.

INPUT:

- *n* – positive integer (default: 2)

OUTPUT: a complex *n*-th root of unity

EXAMPLES:

```

sage: CDF.zeta(7) # rel tol 1e-15
0.6234898018587336 + 0.7818314824680298*I
sage: CDF.zeta(1)

```

(continues on next page)

(continued from previous page)

```
1.0
sage: CDF.zeta()
-1.0
sage: CDF.zeta() == CDF.zeta(2)
True
```

```
>>> from sage.all import *
>>> CDF.zeta(Integer(7)) # rel tol 1e-15
0.6234898018587336 + 0.7818314824680298*I
>>> CDF.zeta(Integer(1))
1.0
>>> CDF.zeta()
-1.0
>>> CDF.zeta() == CDF.zeta(Integer(2))
True
```

```
sage: CDF.zeta(0.5)
Traceback (most recent call last):
...
ValueError: n must be a positive integer
sage: CDF.zeta(0)
Traceback (most recent call last):
...
ValueError: n must be a positive integer
sage: CDF.zeta(-1)
Traceback (most recent call last):
...
ValueError: n must be a positive integer
```

```
>>> from sage.all import *
>>> CDF.zeta(RealNumber('0.5'))
Traceback (most recent call last):
...
ValueError: n must be a positive integer
>>> CDF.zeta(Integer(0))
Traceback (most recent call last):
...
ValueError: n must be a positive integer
>>> CDF.zeta(-Integer(1))
Traceback (most recent call last):
...
ValueError: n must be a positive integer
```

class sage.rings.complex_double.**ComplexToCDF**

Bases: `Morphism`

Fast morphism for anything such that the elements have attributes `.real` and `.imag` (e.g. numpy complex types).

EXAMPLES:

```
sage: # needs numpy
sage: import numpy
```

(continues on next page)

(continued from previous page)

```
sage: f = CDF.coerce_map_from(numpy.complex128)
sage: f(numpy.complex128(I))
1.0*I
sage: f(numpy.complex128(I)).parent()
Complex Double Field
```

```
>>> from sage.all import *
>>> # needs numpy
>>> import numpy
>>> f = CDF.coerce_map_from(numpy.complex128)
>>> f(numpy.complex128(I))
1.0*I
>>> f(numpy.complex128(I)).parent()
Complex Double Field
```

```
class sage.rings.complex_double.FloatToCDF
```

Bases: `Morphism`

Fast morphism from anything with a `__float__` method to a CDF element.

EXAMPLES:

```
sage: f = CDF.coerce_map_from(ZZ); f
Native morphism:
  From: Integer Ring
  To:   Complex Double Field
sage: f(4)
4.0
sage: f = CDF.coerce_map_from(QQ); f
Native morphism:
  From: Rational Field
  To:   Complex Double Field
sage: f(1/2)
0.5
sage: f = CDF.coerce_map_from(int); f
Native morphism:
  From: Set of Python objects of class 'int'
  To:   Complex Double Field
sage: f(3r)
3.0
sage: f = CDF.coerce_map_from(float); f
Native morphism:
  From: Set of Python objects of class 'float'
  To:   Complex Double Field
sage: f(3.5)
3.5
```

```
>>> from sage.all import *
>>> f = CDF.coerce_map_from(ZZ); f
Native morphism:
  From: Integer Ring
  To:   Complex Double Field
```

(continues on next page)

(continued from previous page)

```

>>> f(Integer(4))
4.0
>>> f = CDF.coerce_map_from(QQ); f
Native morphism:
  From: Rational Field
  To:   Complex Double Field
>>> f(Integer(1)/Integer(2))
0.5
>>> f = CDF.coerce_map_from(int); f
Native morphism:
  From: Set of Python objects of class 'int'
  To:   Complex Double Field
>>> f(3)
3.0
>>> f = CDF.coerce_map_from(float); f
Native morphism:
  From: Set of Python objects of class 'float'
  To:   Complex Double Field
>>> f(RealNumber('3.5'))
3.5

```

`sage.rings.complex_double.is_ComplexDoubleElement(x)`

Return True if `x` is a *ComplexDoubleElement*.

EXAMPLES:

```

sage: from sage.rings.complex_double import is_ComplexDoubleElement
sage: is_ComplexDoubleElement(0)
doctest:warning...
DeprecationWarning: The function is_ComplexDoubleElement is deprecated;
use 'isinstance(..., ComplexDoubleElement)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
False
sage: is_ComplexDoubleElement(CDF(0))
True

```

```

>>> from sage.all import *
>>> from sage.rings.complex_double import is_ComplexDoubleElement
>>> is_ComplexDoubleElement(Integer(0))
doctest:warning...
DeprecationWarning: The function is_ComplexDoubleElement is deprecated;
use 'isinstance(..., ComplexDoubleElement)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
False
>>> is_ComplexDoubleElement(CDF(Integer(0)))
True

```


INTERVAL ARITHMETIC

Sage implements real and complex interval arithmetic using **MPFI** (*RealIntervalField*, *ComplexIntervalField*) and **FLINT** (*RealBallField*, *ComplexBallField*).

2.1 Arbitrary precision real intervals using MPFI

AUTHORS:

- Carl Witty (2007-01-21): based on `real_mpfr.pyx`; changed it to use `mpfi` rather than `mpfr`.
- William Stein (2007-01-24): modifications and clean up and docs, etc.
- Niles Johnson (2010-08): [Issue #3893](#): `random_element()` should pass on `*args` and `**kwds`.
- Travis Scrimshaw (2012-10-20): Fixing scientific notation output to fix [Issue #13634](#).
- Travis Scrimshaw (2012-11-02): Added doctests for full coverage

This is a straightforward binding to the **MPFI library**; it may be useful to refer to its documentation for more details.

An interval is represented as a pair of floating-point numbers a and b (where $a \leq b$) and is printed as a standard floating-point number with a question mark (for instance, `3.1416?`). The question mark indicates that the preceding digit may have an error of ± 1 . These floating-point numbers are implemented using MPFR (the same as the *RealNumber* elements of *RealField_class*).

There is also an alternate method of printing, where the interval prints as `[a .. b]` (for instance, `[3.1415 .. 3.1416]`).

The interval represents the set $\{x : a \leq x \leq b\}$ (so if $a = b$, then the interval represents that particular floating-point number). The endpoints can include positive and negative infinity, with the obvious meaning. It is also possible to have a NaN (Not-a-Number) interval, which is represented by having either endpoint be NaN.

PRINTING:

There are two styles for printing intervals: ‘brackets’ style and ‘question’ style (the default).

In question style, we print the “known correct” part of the number, followed by a question mark. The question mark indicates that the preceding digit is possibly wrong by ± 1 .

```
sage: RIF(sqrt(2))
↪needs sage.symbolic
1.414213562373095?
```

```
>>> from sage.all import *
>>> RIF(sqrt(Integer(2)))
↪      # needs sage.symbolic
1.414213562373095?
```

However, if the interval is precise (its lower bound is equal to its upper bound) and equal to a not-too-large integer, then we just print that integer.

```
sage: RIF(0)
0
sage: RIF(654321)
654321
```

```
>>> from sage.all import *
>>> RIF(Integer(0))
0
>>> RIF(Integer(654321))
654321
```

```
sage: RIF(123, 125)
124.?
sage: RIF(123, 126)
1.3?e2
```

```
>>> from sage.all import *
>>> RIF(Integer(123), Integer(125))
124.?
>>> RIF(Integer(123), Integer(126))
1.3?e2
```

As we see in the last example, question style can discard almost a whole digit's worth of precision. We can reduce this by allowing “error digits”: an error following the question mark, that gives the maximum error of the digit(s) before the question mark. If the error is absent (which it always is in the default printing), then it is taken to be 1.

```
sage: RIF(123, 126).str(error_digits=1)
'125.??'
sage: RIF(123, 127).str(error_digits=1)
'125.??'
sage: v = RIF(-e, pi); v                                     #_
↳needs sage.symbolic
0.?e1
sage: v.str(error_digits=1)                                   #_
↳needs sage.symbolic
'1.??4'
sage: v.str(error_digits=5)                                   #_
↳needs sage.symbolic
'0.2117?29300'
```

```
>>> from sage.all import *
>>> RIF(Integer(123), Integer(126)).str(error_digits=Integer(1))
'125.??'
>>> RIF(Integer(123), Integer(127)).str(error_digits=Integer(1))
'125.??'
>>> v = RIF(-e, pi); v                                       #_
↳needs sage.symbolic
0.?e1
>>> v.str(error_digits=Integer(1))                             _
↳ # needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
'1.?4'
>>> v.str(error_digits=Integer(5))
↪      # needs sage.symbolic
'0.2117?29300'
```

Error digits also sometimes let us indicate that the interval is actually equal to a single floating-point number:

```
sage: RIF(54321/256)
212.19140625000000?
sage: RIF(54321/256).str(error_digits=1)
'212.19140625000000?0'
```

```
>>> from sage.all import *
>>> RIF(Integer(54321)/Integer(256))
212.19140625000000?
>>> RIF(Integer(54321)/Integer(256)).str(error_digits=Integer(1))
'212.19140625000000?0'
```

In brackets style, intervals are printed with the left value rounded down and the right rounded up, which is conservative, but in some ways unsatisfying.

Consider a 3-bit interval containing exactly the floating-point number 1.25. In round-to-nearest or round-down, this prints as 1.2; in round-up, this prints as 1.3. The straightforward options, then, are to print this interval as $[1.2 \dots 1.2]$ (which does not even contain the true value, 1.25), or to print it as $[1.2 \dots 1.3]$ (which gives the impression that the upper and lower bounds are not equal, even though they really are). Neither of these is very satisfying, but we have chosen the latter.

```
sage: R = RealIntervalField(3)
sage: a = R(1.25)
sage: a.str(style='brackets')
'[1.2 .. 1.3]'
sage: a == 5/4
True
sage: a == 2
False
```

```
>>> from sage.all import *
>>> R = RealIntervalField(Integer(3))
>>> a = R(RealNumber('1.25'))
>>> a.str(style='brackets')
'[1.2 .. 1.3]'
>>> a == Integer(5)/Integer(4)
True
>>> a == Integer(2)
False
```

Some default printing options can be set by modifying module globals:

```
sage: from sage.rings import real_mpfi
sage: x = RIF(sqrt(2), sqrt(2)+1e-10); x
1.4142135624?
sage: real_mpfi.printing_error_digits = 2
sage: x
```

(continues on next page)

(continued from previous page)

```
1.414213562424?51
sage: real_mpfi.printing_style = 'brackets'
sage: x
[1.4142135623730949 .. 1.4142135624730952]
sage: real_mpfi.printing_style = 'question'; real_mpfi.printing_error_digits = 0 #
↪revert to default
```

```
>>> from sage.all import *
>>> from sage.rings import real_mpfi
>>> x = RIF(sqrt(Integer(2)), sqrt(Integer(2))+RealNumber('1e-10')); x
1.4142135624?
>>> real_mpfi.printing_error_digits = Integer(2)
>>> x
1.414213562424?51
>>> real_mpfi.printing_style = 'brackets'
>>> x
[1.4142135623730949 .. 1.4142135624730952]
>>> real_mpfi.printing_style = 'question'; real_mpfi.printing_error_digits =
↪Integer(0) # revert to default
```

The default value of using scientific notation can be configured at field construction instead:

```
sage: RealIntervalField(53, sci_not=False)(0.5)
0.5000000000000000000?
sage: RealIntervalField(53, sci_not=True)(0.5)
5.000000000000000000?e-1
```

```
>>> from sage.all import *
>>> RealIntervalField(Integer(53), sci_not=False)(RealNumber('0.5'))
0.5000000000000000000?
>>> RealIntervalField(Integer(53), sci_not=True)(RealNumber('0.5'))
5.000000000000000000?e-1
```

COMPARISONS:

Comparison operations (`==`, `!=`, `<`, `<=`, `>`, `>=`) return `True` if every value in the first interval has the given relation to every value in the second interval.

This convention for comparison operators has good and bad points. The good:

- Expected transitivity properties hold (if $a > b$ and $b == c$, then $a > c$; etc.)
- $a == 0$ is true if the interval contains only the floating-point number 0; similarly for $a == 1$
- $a > 0$ means something useful (that every value in the interval is greater than 0)

The bad:

- Trichotomy fails to hold: there are values (a, b) such that none of $a < b$, $a == b$, or $a > b$ are true
- There are values a and b such that $a <= b$ but neither $a < b$ nor $a == b$ hold.
- There are values a and b such that neither $a != b$ nor $a == b$ hold.

Note

Intervals a and b overlap iff `not (a != b)`.

Warning

The `cmp(a, b)` function should not be used to compare real intervals. Note that `cmp` will disappear in Python3.

EXAMPLES:

```
sage: 0 < RIF(1, 2)
True
sage: 0 == RIF(0)
True
sage: not(0 == RIF(0, 1))
True
sage: not(0 != RIF(0, 1))
True
sage: 0 <= RIF(0, 1)
True
sage: not(0 < RIF(0, 1))
True
```

```
>>> from sage.all import *
>>> Integer(0) < RIF(Integer(1), Integer(2))
True
>>> Integer(0) == RIF(Integer(0))
True
>>> not(Integer(0) == RIF(Integer(0), Integer(1)))
True
>>> not(Integer(0) != RIF(Integer(0), Integer(1)))
True
>>> Integer(0) <= RIF(Integer(0), Integer(1))
True
>>> not(Integer(0) < RIF(Integer(0), Integer(1)))
True
```

Comparison with infinity is defined through coercion to the infinity ring where semi-infinite intervals are sent to their central value (plus or minus infinity); this implements the above convention for inequalities:

```
sage: InfinityRing.has_coerce_map_from(RIF)
True
sage: -oo < RIF(-1, 1) < oo
True
sage: -oo < RIF(0, oo) <= oo
True
sage: -oo <= RIF(-oo, -1) < oo
True
```

```
>>> from sage.all import *
>>> InfinityRing.has_coerce_map_from(RIF)
True
```

(continues on next page)

(continued from previous page)

```
>>> -oo < RIF(-Integer(1), Integer(1)) < oo
True
>>> -oo < RIF(Integer(0), oo) <= oo
True
>>> -oo <= RIF(-oo, -Integer(1)) < oo
True
```

Comparison by equality shows what the semi-infinite intervals actually coerce to:

```
sage: RIF(1, oo) == oo
True
sage: RIF(-oo, -1) == -oo
True
```

```
>>> from sage.all import *
>>> RIF(Integer(1), oo) == oo
True
>>> RIF(-oo, -Integer(1)) == -oo
True
```

For lack of a better value in the infinity ring, the doubly infinite interval coerces to plus infinity:

```
sage: RIF(-oo, oo) == oo
True
```

```
>>> from sage.all import *
>>> RIF(-oo, oo) == oo
True
```

If you want to compare two intervals lexicographically, you can use the method `lexico_cmp`. However, the behavior of this method is not specified if given a non-interval and an interval:

```
sage: RIF(0).lexico_cmp(RIF(0, 1))
-1
sage: RIF(0, 1).lexico_cmp(RIF(0))
1
sage: RIF(0, 1).lexico_cmp(RIF(1))
-1
sage: RIF(0, 1).lexico_cmp(RIF(0, 1))
0
```

```
>>> from sage.all import *
>>> RIF(Integer(0)).lexico_cmp(RIF(Integer(0), Integer(1)))
-1
>>> RIF(Integer(0), Integer(1)).lexico_cmp(RIF(Integer(0)))
1
>>> RIF(Integer(0), Integer(1)).lexico_cmp(RIF(Integer(1)))
-1
>>> RIF(Integer(0), Integer(1)).lexico_cmp(RIF(Integer(0), Integer(1)))
0
```

Warning

Mixing symbolic expressions with intervals (in particular, converting constant symbolic expressions to intervals), can lead to incorrect results:

```
sage: ref = RealIntervalField(100)(ComplexBallField(100).one().airy_ai().real())
sage: ref
0.135292416312881415524147423515?
sage: val = RIF(airy_ai(1)); val # known bug
0.13529241631288142?
sage: val.overlaps(ref) # known bug
False
```

```
>>> from sage.all import *
>>> ref = RealIntervalField(Integer(100))(ComplexBallField(Integer(100)).one().airy_
→ai().real())
>>> ref
0.135292416312881415524147423515?
>>> val = RIF(airy_ai(Integer(1))); val # known bug
0.13529241631288142?
>>> val.overlaps(ref) # known bug
False
```

`sage.rings.real_mpfi.RealInterval(s, upper=None, base=10, pad=0, min_prec=53)`

Return the real number defined by the string `s` as an element of `RealIntervalField(prec=n)`, where `n` potentially has slightly more (controlled by `pad`) bits than given by `s`.

INPUT:

- `s` – string that defines a real number (or something whose string representation defines a number)
- `upper` – (default: `None`) upper endpoint of interval if given, in which case `s` is the lower endpoint
- `base` – integer between 2 and 36
- `pad` – integer (default: 0)
- `min_prec` – number will have at least this many bits of precision, no matter what

EXAMPLES:

```
sage: RealInterval('2.3')
2.300000000000000?
sage: RealInterval(10)
10
sage: RealInterval('1.0000000000000000000000000000000000000000000000')
1
sage: RealInterval('1.2345678901234567890123456789012345')
1.2345678901234567890123456789012345?
sage: RealInterval(29308290382930840239842390482, 3^20).str(style='brackets')
'[3.4867844010000000000000000000000000e9 .. 2.93082903829308402398423904820e28]'
```

```
>>> from sage.all import *
>>> RealInterval('2.3')
2.300000000000000?
>>> RealInterval(Integer(10))
```

(continues on next page)

(continued from previous page)

```

10
>>> RealInterval('1.00000000000000000000000000000000')
1
>>> RealInterval('1.2345678901234567890123456789012345')
1.23456789012345678901234567890123450?
>>> RealInterval(Integer(29308290382930840239842390482), Integer(3)**Integer(20)).
↳str(style='brackets')
'[3.4867844010000000000000000000000e9 .. 2.93082903829308402398423904820e28]'

```

sage.rings.real_mpfi.**RealIntervalField**(prec=53, sci_not=False)

Construct a *RealIntervalField_class*, with caching.

INPUT:

- `prec` – integer (default: 53); precision. The number of bits used to represent the mantissa of a floating-point number. The precision can be any integer between `mpfr_prec_min()` and `mpfr_prec_max()`. In the current implementation, `mpfr_prec_min()` is equal to 2.
- `sci_not` – boolean (default: False); whether or not to display using scientific notation

EXAMPLES:

```

sage: RealIntervalField()
Real Interval Field with 53 bits of precision
sage: RealIntervalField(200, sci_not=True)
Real Interval Field with 200 bits of precision
sage: RealIntervalField(53) is RIF
True
sage: RealIntervalField(200) is RIF
False
sage: RealIntervalField(200) is RealIntervalField(200)
True

```

```

>>> from sage.all import *
>>> RealIntervalField()
Real Interval Field with 53 bits of precision
>>> RealIntervalField(Integer(200), sci_not=True)
Real Interval Field with 200 bits of precision
>>> RealIntervalField(Integer(53)) is RIF
True
>>> RealIntervalField(Integer(200)) is RIF
False
>>> RealIntervalField(Integer(200)) is RealIntervalField(Integer(200))
True

```

See the documentation for *RealIntervalField_class* for many more examples.

class sage.rings.real_mpfi.**RealIntervalFieldElement**

Bases: *RingElement*

A real number interval.

absolute_diameter()

The diameter of this interval (for $[a..b]$, this is $b - a$), rounded upward, as a *RealNumber*.

EXAMPLES:


```
sage: RIF(1, pi).absolute_diameter()
↪needs sage.symbolic
2.14159265358979
```

```
>>> from sage.all import *
>>> RIF(Integer(1), pi).absolute_diameter()
↪ # needs sage.symbolic
2.14159265358979
```

alea()

Return a floating-point number picked at random from the interval.

EXAMPLES:

```
sage: RIF(1, 2).alea() # random
1.34696133696137
```

```
>>> from sage.all import *
>>> RIF(Integer(1), Integer(2)).alea() # random
1.34696133696137
```

algdep(*n*)

Return a polynomial of degree at most *n* which is approximately satisfied by *self*.

Note

The returned polynomial need not be irreducible, and indeed usually won't be if *self* is a good approximation to an algebraic number of degree less than *n*.

Pari needs to know the number of “known good bits” in the number; we automatically get that from the interval width.

ALGORITHM:

This uses the PARI C-library `pari:algdep` command.

EXAMPLES:

```
sage: r = sqrt(RIF(2)); r
1.414213562373095?
sage: r.algebraic_dependency(5)
x^2 - 2
```

```
>>> from sage.all import *
>>> r = sqrt(RIF(Integer(2))); r
1.414213562373095?
>>> r.algebraic_dependency(Integer(5))
x^2 - 2
```

If we compute a wrong, but precise, interval, we get a wrong answer:

```
sage: r = sqrt(RealIntervalField(200)(2)) + (1/2)^40; r
1.414213562374004543503461652447613117632171875376948073176680?
```

(continues on next page)

(continued from previous page)

```
sage: r.algebraic_dependency(5)
7266488*x^5 + 22441629*x^4 - 90470501*x^3 + 23297703*x^2 + 45778664*x +
↳13681026
```

```
>>> from sage.all import *
>>> r = sqrt(RealIntervalField(Integer(200))(Integer(2))) + (Integer(1)/
↳Integer(2))**Integer(40); r
1.414213562374004543503461652447613117632171875376948073176680?
>>> r.algebraic_dependency(Integer(5))
7266488*x^5 + 22441629*x^4 - 90470501*x^3 + 23297703*x^2 + 45778664*x +
↳13681026
```

But if we compute an interval that includes the number we mean, we're much more likely to get the right answer, even if the interval is very imprecise:

```
sage: r = r.union(sqrt(2.0))
sage: r.algebraic_dependency(5)
x^2 - 2
```

```
>>> from sage.all import *
>>> r = r.union(sqrt(RealNumber('2.0')))
>>> r.algebraic_dependency(Integer(5))
x^2 - 2
```

Even on this extremely imprecise interval we get an answer which is technically correct:

```
sage: RIF(-1, 1).algebraic_dependency(5)
x
```

```
>>> from sage.all import *
>>> RIF(-Integer(1), Integer(1)).algebraic_dependency(Integer(5))
x
```

`algebraic_dependency(n)`

Return a polynomial of degree at most n which is approximately satisfied by `self`.

Note

The returned polynomial need not be irreducible, and indeed usually won't be if `self` is a good approximation to an algebraic number of degree less than n .

Pari needs to know the number of “known good bits” in the number; we automatically get that from the interval width.

ALGORITHM:

This uses the PARI C-library `pari:algdep` command.

EXAMPLES:

```
sage: r = sqrt(RIF(2)); r
1.414213562373095?
```

(continues on next page)

(continued from previous page)

```
sage: r.algebraic_dependency(5)
x^2 - 2
```

```
>>> from sage.all import *
>>> r = sqrt(RIF(Integer(2))); r
1.414213562373095?
>>> r.algebraic_dependency(Integer(5))
x^2 - 2
```

If we compute a wrong, but precise, interval, we get a wrong answer:

```
sage: r = sqrt(RealIntervalField(200)(2)) + (1/2)^40; r
1.414213562374004543503461652447613117632171875376948073176680?
sage: r.algebraic_dependency(5)
7266488*x^5 + 22441629*x^4 - 90470501*x^3 + 23297703*x^2 + 45778664*x + 13681026
```

```
>>> from sage.all import *
>>> r = sqrt(RealIntervalField(Integer(200))(Integer(2))) + (Integer(1)/
↳ Integer(2))**Integer(40); r
1.414213562374004543503461652447613117632171875376948073176680?
>>> r.algebraic_dependency(Integer(5))
7266488*x^5 + 22441629*x^4 - 90470501*x^3 + 23297703*x^2 + 45778664*x + 13681026
```

But if we compute an interval that includes the number we mean, we're much more likely to get the right answer, even if the interval is very imprecise:

```
sage: r = r.union(sqrt(2.0))
sage: r.algebraic_dependency(5)
x^2 - 2
```

```
>>> from sage.all import *
>>> r = r.union(sqrt(RealNumber('2.0')))
>>> r.algebraic_dependency(Integer(5))
x^2 - 2
```

Even on this extremely imprecise interval we get an answer which is technically correct:

```
sage: RIF(-1, 1).algebraic_dependency(5)
x
```

```
>>> from sage.all import *
>>> RIF(-Integer(1), Integer(1)).algebraic_dependency(Integer(5))
x
```

arccos()

Return the inverse cosine of `self`.

EXAMPLES:

```

sage: q = RIF.pi()/3; q
1.047197551196598?
sage: i = q.cos(); i
0.500000000000000?
sage: q2 = i.arccos(); q2
1.047197551196598?
sage: q == q2
False
sage: q != q2
False
sage: q2.lower() == q.lower()
False
sage: q - q2
0.?e-15
sage: q in q2
True

```

```

>>> from sage.all import *
>>> q = RIF.pi()/Integer(3); q
1.047197551196598?
>>> i = q.cos(); i
0.500000000000000?
>>> q2 = i.arccos(); q2
1.047197551196598?
>>> q == q2
False
>>> q != q2
False
>>> q2.lower() == q.lower()
False
>>> q - q2
0.?e-15
>>> q in q2
True

```

arccosh()

Return the hyperbolic inverse cosine of `self`.

EXAMPLES:

```

sage: q = RIF.pi()/2
sage: i = q.arccosh(); i
1.023227478547551?

```

```

>>> from sage.all import *
>>> q = RIF.pi()/Integer(2)
>>> i = q.arccosh(); i
1.023227478547551?

```

arccoth()

Return the inverse hyperbolic cotangent of `self`.

EXAMPLES:

```
sage: RealIntervalField(100)(2).arccoth()
0.549306144334054845697622618462?
sage: (2.0).arccoth()
0.549306144334055
```

```
>>> from sage.all import *
>>> RealIntervalField(Integer(100))(Integer(2)).arccoth()
0.549306144334054845697622618462?
>>> (RealNumber('2.0')).arccoth()
0.549306144334055
```

arccsch()

Return the inverse hyperbolic cosecant of `self`.

EXAMPLES:

```
sage: RealIntervalField(100)(2).arccsch()
0.481211825059603447497758913425?
sage: (2.0).arccsch()
0.481211825059603
```

```
>>> from sage.all import *
>>> RealIntervalField(Integer(100))(Integer(2)).arccsch()
0.481211825059603447497758913425?
>>> (RealNumber('2.0')).arccsch()
0.481211825059603
```

arcsech()

Return the inverse hyperbolic secant of `self`.

EXAMPLES:

```
sage: RealIntervalField(100)(0.5).arcsech()
1.316957896924816708625046347308?
sage: (0.5).arcsech()
1.31695789692482
```

```
>>> from sage.all import *
>>> RealIntervalField(Integer(100))(RealNumber('0.5')).arcsech()
1.316957896924816708625046347308?
>>> (RealNumber('0.5')).arcsech()
1.31695789692482
```

arcsin()

Return the inverse sine of `self`.

EXAMPLES:

```
sage: q = RIF.pi()/5; q
0.6283185307179587?
sage: i = q.sin(); i
0.587785252292474?
sage: q2 = i.arcsin(); q2
```

(continues on next page)

(continued from previous page)

```

0.628318530717959?
sage: q == q2
False
sage: q != q2
False
sage: q2.lower() == q.lower()
False
sage: q - q2
0.?e-15
sage: q in q2
True

```

```

>>> from sage.all import *
>>> q = RIF.pi()/Integer(5); q
0.6283185307179587?
>>> i = q.sin(); i
0.587785252292474?
>>> q2 = i.arcsin(); q2
0.628318530717959?
>>> q == q2
False
>>> q != q2
False
>>> q2.lower() == q.lower()
False
>>> q - q2
0.?e-15
>>> q in q2
True

```

arcsinh()

Return the hyperbolic inverse sine of `self`.

EXAMPLES:

```

sage: q = RIF.pi()/7
sage: i = q.sinh(); i
0.464017630492991?
sage: i.arcsinh() - q
0.?e-15

```

```

>>> from sage.all import *
>>> q = RIF.pi()/Integer(7)
>>> i = q.sinh(); i
0.464017630492991?
>>> i.arcsinh() - q
0.?e-15

```

arctan()

Return the inverse tangent of `self`.

EXAMPLES:

```

sage: q = RIF.pi()/5; q
0.6283185307179587?
sage: i = q.tan(); i
0.726542528005361?
sage: q2 = i.arctan(); q2
0.628318530717959?
sage: q == q2
False
sage: q != q2
False
sage: q2.lower() == q.lower()
False
sage: q - q2
0.?e-15
sage: q in q2
True

```

```

>>> from sage.all import *
>>> q = RIF.pi()/Integer(5); q
0.6283185307179587?
>>> i = q.tan(); i
0.726542528005361?
>>> q2 = i.arctan(); q2
0.628318530717959?
>>> q == q2
False
>>> q != q2
False
>>> q2.lower() == q.lower()
False
>>> q - q2
0.?e-15
>>> q in q2
True

```

arctanh()

Return the hyperbolic inverse tangent of `self`.

EXAMPLES:

```

sage: q = RIF.pi()/7
sage: i = q.tanh() ; i
0.420911241048535?
sage: i.arctanh() - q
0.?e-15

```

```

>>> from sage.all import *
>>> q = RIF.pi()/Integer(7)
>>> i = q.tanh() ; i
0.420911241048535?
>>> i.arctanh() - q
0.?e-15

```

argument()

The argument of this interval, if it is well-defined, in the complex sense. Otherwise raises a `ValueError`.

OUTPUT:

- an element of the parent of this interval (0 or pi)

EXAMPLES:

```
sage: RIF(1).argument()
0
sage: RIF(-1).argument()
3.141592653589794?
sage: RIF(0,1).argument()
0
sage: RIF(-1,0).argument()
3.141592653589794?
sage: RIF(0).argument()
Traceback (most recent call last):
...
ValueError: Can't take the argument of an exact zero
sage: RIF(-1,1).argument()
Traceback (most recent call last):
...
ValueError: Can't take the argument of interval strictly containing zero
```

```
>>> from sage.all import *
>>> RIF(Integer(1)).argument()
0
>>> RIF(-Integer(1)).argument()
3.141592653589794?
>>> RIF(Integer(0), Integer(1)).argument()
0
>>> RIF(-Integer(1), Integer(0)).argument()
3.141592653589794?
>>> RIF(Integer(0)).argument()
Traceback (most recent call last):
...
ValueError: Can't take the argument of an exact zero
>>> RIF(-Integer(1), Integer(1)).argument()
Traceback (most recent call last):
...
ValueError: Can't take the argument of interval strictly containing zero
```

bisection()

Return the bisection of `self` into two intervals of half the size whose union is `self` and intersection is `center()`.

EXAMPLES:

```
sage: a, b = RIF(1,2).bisection()
sage: a.lower(), a.upper()
(1.000000000000000, 1.500000000000000)
sage: b.lower(), b.upper()
(1.500000000000000, 2.000000000000000)
```

(continues on next page)

(continued from previous page)

```

sage: # needs sage.symbolic
sage: I = RIF(e, pi)
sage: a, b = I.bisection()
sage: a.intersection(b) == RIF(I.center())
True
sage: a.union(b).endpoints() == I.endpoints()
True

```

```

>>> from sage.all import *
>>> a, b = RIF(Integer(1), Integer(2)).bisection()
>>> a.lower(), a.upper()
(1.000000000000000, 1.500000000000000)
>>> b.lower(), b.upper()
(1.500000000000000, 2.000000000000000)

>>> # needs sage.symbolic
>>> I = RIF(e, pi)
>>> a, b = I.bisection()
>>> a.intersection(b) == RIF(I.center())
True
>>> a.union(b).endpoints() == I.endpoints()
True

```

ceil()

Return the ceiling of this interval as an interval.

The ceiling of a real number x is the smallest integer larger than or equal to x .

See also

- `unique_ceil()` – return the ceil as an integer if it is unique and raises a `ValueError` otherwise
- `floor()` – truncation towards minus infinity
- `trunc()` – truncation towards zero
- `round()` – rounding

EXAMPLES:

```

sage: (2.99).ceil()
3
sage: (2.00).ceil()
2
sage: (2.01).ceil()
3
sage: R = RealIntervalField(30)
sage: a = R(-9.5, -11.3); a.str(style='brackets')
'[-11.300000012 .. -9.500000000]'
sage: a.floor().str(style='brackets')
'[-12.000000000 .. -10.000000000]'

```

(continues on next page)

(continued from previous page)

```
sage: a.ceil()
-10.?
sage: ceil(a).str(style='brackets')
'[-11.0000000000 .. -9.0000000000]'
```

```
>>> from sage.all import *
>>> (RealNumber('2.99')).ceil()
3
>>> (RealNumber('2.00')).ceil()
2
>>> (RealNumber('2.01')).ceil()
3
>>> R = RealIntervalField(Integer(30))
>>> a = R(-RealNumber('9.5'), -RealNumber('11.3')); a.str(style='brackets')
'[-11.300000012 .. -9.5000000000]'
>>> a.floor().str(style='brackets')
'[-12.0000000000 .. -10.0000000000]'
>>> a.ceil()
-10.?
>>> ceil(a).str(style='brackets')
'[-11.0000000000 .. -9.0000000000]'
```

ceiling()

Return the ceiling of this interval as an interval.

The ceiling of a real number x is the smallest integer larger than or equal to x .

See also

- `unique_ceil()` – return the ceil as an integer if it is unique and raises a `ValueError` otherwise
- `floor()` – truncation towards minus infinity
- `trunc()` – truncation towards zero
- `round()` – rounding

EXAMPLES:

```
sage: (2.99).ceil()
3
sage: (2.00).ceil()
2
sage: (2.01).ceil()
3
sage: R = RealIntervalField(30)
sage: a = R(-9.5, -11.3); a.str(style='brackets')
'[-11.300000012 .. -9.5000000000]'
sage: a.floor().str(style='brackets')
'[-12.0000000000 .. -10.0000000000]'
sage: a.ceil()
-10.?
```

(continues on next page)

(continued from previous page)

```
sage: ceil(a).str(style='brackets')
'[-11.0000000000 .. -9.0000000000]'
```

```
>>> from sage.all import *
>>> (RealNumber('2.99')).ceil()
3
>>> (RealNumber('2.00')).ceil()
2
>>> (RealNumber('2.01')).ceil()
3
>>> R = RealIntervalField(Integer(30))
>>> a = R(-RealNumber('9.5'), -RealNumber('11.3')); a.str(style='brackets')
'[-11.300000012 .. -9.5000000000]'
>>> a.floor().str(style='brackets')
'[-12.0000000000 .. -10.0000000000]'
>>> a.ceil()
-10.?
>>> ceil(a).str(style='brackets')
'[-11.0000000000 .. -9.0000000000]'
```

center()

Compute the center of the interval $[a..b]$ which is $(a + b)/2$.

EXAMPLES:

```
sage: RIF(1, 2).center()
1.500000000000000
```

```
>>> from sage.all import *
>>> RIF(Integer(1), Integer(2)).center()
1.500000000000000
```

contains_zero()

Return True if self is an interval containing zero.

EXAMPLES:

```
sage: RIF(0).contains_zero()
True
sage: RIF(1, 2).contains_zero()
False
sage: RIF(-1, 1).contains_zero()
True
sage: RIF(-1, 0).contains_zero()
True
```

```
>>> from sage.all import *
>>> RIF(Integer(0)).contains_zero()
True
>>> RIF(Integer(1), Integer(2)).contains_zero()
False
>>> RIF(-Integer(1), Integer(1)).contains_zero()
```

(continues on next page)

(continued from previous page)

```
True
>>> RIF(-Integer(1), Integer(0)).contains_zero()
True
```

cos()Return the cosine of `self`.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: t = RIF(pi)/2
sage: t.cos()
0.?e-15
sage: t.cos().str(style='brackets')
'[-1.6081226496766367e-16 .. 6.1232339957367661e-17]'
sage: t.cos().cos()
0.9999999999999999?
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> t = RIF(pi)/Integer(2)
>>> t.cos()
0.?e-15
>>> t.cos().str(style='brackets')
'[-1.6081226496766367e-16 .. 6.1232339957367661e-17]'
>>> t.cos().cos()
0.9999999999999999?
```

cosh()Return the hyperbolic cosine of `self`.

EXAMPLES:

```
sage: q = RIF.pi()/12
sage: q.cosh()
1.034465640095511?
```

```
>>> from sage.all import *
>>> q = RIF.pi()/Integer(12)
>>> q.cosh()
1.034465640095511?
```

cot()Return the cotangent of `self`.

EXAMPLES:

```
sage: RealIntervalField(100)(2).cot()
-0.457657554360285763750277410432?
```

```
>>> from sage.all import *
>>> RealIntervalField(Integer(100))(Integer(2)).cot()
-0.457657554360285763750277410432?
```

coth()

Return the hyperbolic cotangent of `self`.

EXAMPLES:

```
sage: RealIntervalField(100)(2).coth()
1.03731472072754809587780976477?
```

```
>>> from sage.all import *
>>> RealIntervalField(Integer(100))(Integer(2)).coth()
1.03731472072754809587780976477?
```

csc()

Return the cosecant of `self`.

EXAMPLES:

```
sage: RealIntervalField(100)(2).csc()
1.099750170294616466756697397026?
```

```
>>> from sage.all import *
>>> RealIntervalField(Integer(100))(Integer(2)).csc()
1.099750170294616466756697397026?
```

csch()

Return the hyperbolic cosecant of `self`.

EXAMPLES:

```
sage: RealIntervalField(100)(2).csch()
0.275720564771783207758351482163?
```

```
>>> from sage.all import *
>>> RealIntervalField(Integer(100))(Integer(2)).csch()
0.275720564771783207758351482163?
```

diameter()

If 0 is in `self`, then return `absolute_diameter()`, otherwise return `relative_diameter()`.

EXAMPLES:

```
sage: RIF(1, 2).diameter()
0.6666666666666667
sage: RIF(1, 2).absolute_diameter()
1.0000000000000000
sage: RIF(1, 2).relative_diameter()
0.6666666666666667

sage: # needs sage.symbolic
sage: RIF(pi).diameter()
1.41357985842823e-16
sage: RIF(pi).absolute_diameter()
4.44089209850063e-16
sage: RIF(pi).relative_diameter()
```

(continues on next page)

(continued from previous page)

```

1.41357985842823e-16
sage: (RIF(pi) - RIF(3, 22/7)).diameter()
0.142857142857144
sage: (RIF(pi) - RIF(3, 22/7)).absolute_diameter()
0.142857142857144
sage: (RIF(pi) - RIF(3, 22/7)).relative_diameter()
2.03604377705518

```

```

>>> from sage.all import *
>>> RIF(Integer(1), Integer(2)).diameter()
0.666666666666667
>>> RIF(Integer(1), Integer(2)).absolute_diameter()
1.000000000000000
>>> RIF(Integer(1), Integer(2)).relative_diameter()
0.666666666666667

>>> # needs sage.symbolic
>>> RIF(pi).diameter()
1.41357985842823e-16
>>> RIF(pi).absolute_diameter()
4.44089209850063e-16
>>> RIF(pi).relative_diameter()
1.41357985842823e-16
>>> (RIF(pi) - RIF(Integer(3), Integer(22)/Integer(7))).diameter()
0.142857142857144
>>> (RIF(pi) - RIF(Integer(3), Integer(22)/Integer(7))).absolute_diameter()
0.142857142857144
>>> (RIF(pi) - RIF(Integer(3), Integer(22)/Integer(7))).relative_diameter()
2.03604377705518

```

edges()

Return the lower and upper endpoints of this interval as intervals.

OUTPUT: a 2-tuple of real intervals (lower endpoint, upper endpoint) each containing just one point.

See also

`endpoints()` which returns the endpoints as real numbers instead of intervals.

EXAMPLES:

```

sage: RIF(1,2).edges()
(1, 2)
sage: RIF(pi).edges()
↳needs sage.symbolic
(3.1415926535897932?, 3.1415926535897936?)

```

```

>>> from sage.all import *
>>> RIF(Integer(1), Integer(2)).edges()
(1, 2)
>>> RIF(pi).edges()

```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
(3.1415926535897932?, 3.1415926535897936?)
```

endpoints (*rnd=None*)

Return the lower and upper endpoints of this interval.

OUTPUT: a 2-tuple of real numbers (lower endpoint, upper endpoint)

See also`edges()` which returns the endpoints as exact intervals instead of real numbers.**EXAMPLES:**

```
sage: RIF(1,2).endpoints()
(1.000000000000000, 2.000000000000000)
sage: RIF(pi).endpoints()
↪needs sage.symbolic
(3.14159265358979, 3.14159265358980)
sage: a = CIF(RIF(1,2), RIF(3,4))
sage: a.real().endpoints()
(1.000000000000000, 2.000000000000000)
```

#

```
>>> from sage.all import *
>>> RIF(Integer(1),Integer(2)).endpoints()
(1.000000000000000, 2.000000000000000)
>>> RIF(pi).endpoints()
↪needs sage.symbolic
(3.14159265358979, 3.14159265358980)
>>> a = CIF(RIF(Integer(1),Integer(2)), RIF(Integer(3),Integer(4)))
>>> a.real().endpoints()
(1.000000000000000, 2.000000000000000)
```

#

As with `lower()` and `upper()`, a rounding mode is accepted:

```
sage: RIF(1,2).endpoints('RNDD')[0].parent()
Real Field with 53 bits of precision and rounding RNDD
```

```
>>> from sage.all import *
>>> RIF(Integer(1),Integer(2)).endpoints('RNDD')[Integer(0)].parent()
Real Field with 53 bits of precision and rounding RNDD
```

exp()Return e^{self} .**EXAMPLES:**

```
sage: r = RIF(0.0)
sage: r.exp()
1
```

```
>>> from sage.all import *
>>> r = RIF(RealNumber('0.0'))
>>> r.exp()
1
```

```
sage: r = RIF(32.3)
sage: a = r.exp(); a
1.065888472748645?e14
sage: a.log()
32.300000000000000?
```

```
>>> from sage.all import *
>>> r = RIF(RealNumber('32.3'))
>>> a = r.exp(); a
1.065888472748645?e14
>>> a.log()
32.300000000000000?
```

```
sage: r = RIF(-32.3)
sage: r.exp()
9.38184458849869?e-15
```

```
>>> from sage.all import *
>>> r = RIF(-RealNumber('32.3'))
>>> r.exp()
9.38184458849869?e-15
```

exp2()

Return 2^{self} .

EXAMPLES:

```
sage: r = RIF(0.0)
sage: r.exp2()
1
```

```
>>> from sage.all import *
>>> r = RIF(RealNumber('0.0'))
>>> r.exp2()
1
```

```
sage: r = RIF(32.0)
sage: r.exp2()
4294967296
```

```
>>> from sage.all import *
>>> r = RIF(RealNumber('32.0'))
>>> r.exp2()
4294967296
```



```
sage: r = RIF(-32.3)
sage: r.exp2()
1.891172482530207?e-10
```

```
>>> from sage.all import *
>>> r = RIF(-RealNumber('32.3'))
>>> r.exp2()
1.891172482530207?e-10
```

factorial()

Return the factorial evaluated on `self`.

EXAMPLES:

```
sage: RIF(5).factorial()
120
sage: RIF(2.3, 5.7).factorial()
1.?e3
sage: RIF(2.3).factorial()
2.683437381955768?
```

```
>>> from sage.all import *
>>> RIF(Integer(5)).factorial()
120
>>> RIF(RealNumber('2.3'), RealNumber('5.7')).factorial()
1.?e3
>>> RIF(RealNumber('2.3')).factorial()
2.683437381955768?
```

Recover the factorial as integer:

```
sage: f = RealIntervalField(200)(50).factorial()
sage: f
3.0414093201713378043612608166064768844377641568960512000000000?e64
sage: f.unique_integer()
30414093201713378043612608166064768844377641568960512000000000000
sage: 50.factorial()
30414093201713378043612608166064768844377641568960512000000000000
```

```
>>> from sage.all import *
>>> f = RealIntervalField(Integer(200))(Integer(50)).factorial()
>>> f
3.0414093201713378043612608166064768844377641568960512000000000?e64
>>> f.unique_integer()
30414093201713378043612608166064768844377641568960512000000000000
>>> Integer(50).factorial()
30414093201713378043612608166064768844377641568960512000000000000
```

floor()

Return the floor of this interval as an interval.

The floor of a real number x is the largest integer smaller than or equal to x .

- `unique_floor()` – method which returns the floor as an integer if it is unique or raises a `ValueError` otherwise
- `ceil()` – truncation towards plus infinity
- `round()` – rounding
- `trunc()` – truncation towards zero

EXAMPLES:

[illegible]

```
>>> from sage.all import *
>>> R = RealIntervalField()
>>> (RealNumber('2.99')).floor()
2
>>> (RealNumber('2.00')).floor()
2
>>> floor(RR(-Integer(5)/Integer(2)))
-3
>>> R = RealIntervalField(Integer(100))
>>> a = R(RealNumber('9.5'), RealNumber('11.3')); a.str(style='brackets')
'[9.500000000000000000000000000000000000000000000000000 .. 11.300000000000000000000000000000000000000000000000000]'
>>> floor(a).str(style='brackets')
'[9.000000000000000000000000000000000000000000000000000 .. 11.000000000000000000000000000000000000000000000000000]'
>>> a.floor()
10.?
>>> ceil(a)
11.?
>>> a.ceil().str(style='brackets')
'[10.000000000000000000000000000000000000000000000000000 .. 12.000000000000000000000000000000000000000000000000000]'
```

```
fp_rank_diameter()
```

Compute the diameter of this interval in terms of the “floating-point rank”.

The floating-point rank is the number of floating-point numbers (of the current precision) contained in the given interval, minus one. An `fp_rank_diameter` of 0 means that the interval is exact; an `fp_rank_diameter` of 1 means that the interval is as tight as possible, unless the number you're trying to represent is actually exactly representable as a floating-point number.

EXAMPLES:

```
sage: RIF(12345).fp_rank_diameter()
0
sage: RIF(5/8).fp_rank_diameter()
0
sage: RIF(5/7).fp_rank_diameter()
1

sage: # needs sage.symbolic
sage: RIF(pi).fp_rank_diameter()
1
sage: RIF(-sqrt(2)).fp_rank_diameter()
1
sage: a = RIF(pi)^12345; a
2.06622879260?e6137
sage: a.fp_rank_diameter()
30524
sage: (RIF(sqrt(2)) - RIF(sqrt(2))).fp_rank_diameter()
9671406088542672151117826 # 32-bit
41538374868278620559869609387229186 # 64-bit
```

```
>>> from sage.all import *
>>> RIF(Integer(12345)).fp_rank_diameter()
0
>>> RIF(Integer(5)/Integer(8)).fp_rank_diameter()
0
>>> RIF(Integer(5)/Integer(7)).fp_rank_diameter()
1

>>> # needs sage.symbolic
>>> RIF(pi).fp_rank_diameter()
1
>>> RIF(-sqrt(Integer(2))).fp_rank_diameter()
1
>>> a = RIF(pi)**Integer(12345); a
2.06622879260?e6137
>>> a.fp_rank_diameter()
30524
>>> (RIF(sqrt(Integer(2))) - RIF(sqrt(Integer(2)))).fp_rank_diameter()
9671406088542672151117826 # 32-bit
41538374868278620559869609387229186 # 64-bit
```

Just because we have the best possible interval, doesn't mean the interval is actually small:

```
sage: a = RIF(pi)^12345678901234567890; a #_
↪needs sage.symbolic
[2.0985787164673874e323228496 .. +infinity] # 32-bit
[5.8756537891115869e1388255822130839282 .. +infinity] # 64-bit
```

(continues on next page)

(continued from previous page)

```
sage: a.fp_rank_diameter() #_
↳needs sage.symbolic
1
```

```
>>> from sage.all import *
>>> a = RIF(pi)**Integer(12345678901234567890); a #_
↳ # needs sage.symbolic
[2.0985787164673874e323228496 .. +infinity] # 32-bit
[5.8756537891115869e1388255822130839282 .. +infinity] # 64-bit
>>> a.fp_rank_diameter() #_
↳needs sage.symbolic
1
```

frac()

Return the fractional part of this interval as an interval.

The fractional part y of a real number x is the unique element in the interval $(-1, 1)$ that has the same sign as x and such that $x - y$ is an integer. The integer $x - y$ can be obtained through the method `trunc()`.

The output of this function is the smallest interval that contains all possible values of $\text{frac}(x)$ for x in this interval. Note that if it contains an integer then the answer might not be very meaningful. More precisely, if the endpoints are a and b then:

- if $\text{floor}(b) > \max(a, 0)$ then the interval obtained contains $[0, 1]$,
- if $\text{ceil}(a) < \min(b, 0)$ then the interval obtained contains $[-1, 0]$.

See also

`trunc()` – return the integer part complement to this fractional part

EXAMPLES:

```
sage: RIF(2.37123, 2.372).frac()
0.372?
sage: RIF(-23.12, -23.13).frac()
-0.13?

sage: RIF(.5, 1).frac().endpoints()
(0.0000000000000000, 1.0000000000000000)
sage: RIF(1, 1.5).frac().endpoints()
(0.0000000000000000, 0.5000000000000000)

sage: r = RIF(-22.47, -22.468)
sage: r in (r.frac() + r.trunc())
True

sage: r = RIF(18.222, 18.223)
sage: r in (r.frac() + r.trunc())
True

sage: RIF(1.99, 2.025).frac().endpoints()
(0.0000000000000000, 1.0000000000000000)
```

(continues on next page)

(continued from previous page)

```
sage: RIF(1.99, 2.00).frac().endpoints()
(0.0000000000000000, 1.0000000000000000)
sage: RIF(2.00, 2.025).frac().endpoints()
(0.0000000000000000, 0.0250000000000000)

sage: RIF(-2.1, -0.9).frac().endpoints()
(-1.0000000000000000, -0.0000000000000000)
sage: RIF(-0.5, 0.5).frac().endpoints()
(-0.5000000000000000, 0.5000000000000000)
```

```
>>> from sage.all import *
>>> RIF(RealNumber('2.37123'), RealNumber('2.372')).frac()
0.372?
>>> RIF(-RealNumber('23.12'), -RealNumber('23.13')).frac()
-0.13?

>>> RIF(RealNumber('.5'), Integer(1)).frac().endpoints()
(0.0000000000000000, 1.0000000000000000)
>>> RIF(Integer(1), RealNumber('1.5')).frac().endpoints()
(0.0000000000000000, 0.5000000000000000)

>>> r = RIF(-RealNumber('22.47'), -RealNumber('22.468'))
>>> r in (r.frac() + r.trunc())
True

>>> r = RIF(RealNumber('18.222'), RealNumber('18.223'))
>>> r in (r.frac() + r.trunc())
True

>>> RIF(RealNumber('1.99'), RealNumber('2.025')).frac().endpoints()
(0.0000000000000000, 1.0000000000000000)
>>> RIF(RealNumber('1.99'), RealNumber('2.00')).frac().endpoints()
(0.0000000000000000, 1.0000000000000000)
>>> RIF(RealNumber('2.00'), RealNumber('2.025')).frac().endpoints()
(0.0000000000000000, 0.0250000000000000)

>>> RIF(-RealNumber('2.1'), -RealNumber('0.9')).frac().endpoints()
(-1.0000000000000000, -0.0000000000000000)
>>> RIF(-RealNumber('0.5'), RealNumber('0.5')).frac().endpoints()
(-0.5000000000000000, 0.5000000000000000)
```

gamma()

Return the gamma function evaluated on `self`.

EXAMPLES:

```
sage: RIF(1).gamma()
1
sage: RIF(5).gamma()
24
sage: a = RIF(3,4).gamma(); a
1.?e1
```

(continues on next page)

(continued from previous page)

```

sage: a.lower(), a.upper()
(2.000000000000000, 6.000000000000000)
sage: RIF(-1/2).gamma()
-3.54490770181104?
sage: gamma(-1/2).n(100) in RIF(-1/2).gamma() #_
↪needs sage.symbolic
True
sage: RIF1000 = RealIntervalField(1000)
sage: 0 in (RIF1000(RealField(2000))(-19/3).gamma()) - RIF1000(-19/3).gamma()
True
sage: gamma(RIF(100))
9.33262154439442?e155
sage: gamma(RIF(-10000/3))
1.31280781451?e-10297

```

```

>>> from sage.all import *
>>> RIF(Integer(1)).gamma()
1
>>> RIF(Integer(5)).gamma()
24
>>> a = RIF(Integer(3), Integer(4)).gamma(); a
1.?e1
>>> a.lower(), a.upper()
(2.000000000000000, 6.000000000000000)
>>> RIF(-Integer(1)/Integer(2)).gamma()
-3.54490770181104?
>>> gamma(-Integer(1)/Integer(2)).n(Integer(100)) in RIF(-Integer(1)/
↪Integer(2)).gamma() # needs sage.symbolic
True
>>> RIF1000 = RealIntervalField(Integer(1000))
>>> Integer(0) in (RIF1000(RealField(Integer(2000))(-Integer(19)/Integer(3)).
↪gamma()) - RIF1000(-Integer(19)/Integer(3)).gamma()
True
>>> gamma(RIF(Integer(100)))
9.33262154439442?e155
>>> gamma(RIF(-Integer(10000)/Integer(3)))
1.31280781451?e-10297

```

Verify the result contains the local minima:

```

sage: 0.88560319441088 in RIF(1, 2).gamma()
True
sage: 0.88560319441088 in RIF(0.25, 4).gamma()
True
sage: 0.88560319441088 in RIF(1.4616, 1.46164).gamma()
True

sage: (-0.99).gamma()
-100.436954665809
sage: (-0.01).gamma()
-100.587197964411
sage: RIF(-0.99, -0.01).gamma().upper()

```

(continues on next page)

(continued from previous page)

```
-1.60118039970055
```

```
>>> from sage.all import *
>>> RealNumber('0.88560319441088') in RIF(Integer(1), Integer(2)).gamma()
True
>>> RealNumber('0.88560319441088') in RIF(RealNumber('0.25'), Integer(4)).
↳gamma()
True
>>> RealNumber('0.88560319441088') in RIF(RealNumber('1.4616'), RealNumber('1.
↳46164')).gamma()
True

>>> (-RealNumber('0.99')).gamma()
-100.436954665809
>>> (-RealNumber('0.01')).gamma()
-100.587197964411
>>> RIF(-RealNumber('0.99'), -RealNumber('0.01')).gamma().upper()
-1.60118039970055
```

Correctly detects poles:

```
sage: gamma(RIF(-3/2, -1/2))
[-infinity .. +infinity]
```

```
>>> from sage.all import *
>>> gamma(RIF(-Integer(3)/Integer(2), -Integer(1)/Integer(2)))
[-infinity .. +infinity]
```

imag()

Return the imaginary part of this real interval.

(Since this interval is real, this simply returns the zero interval.)

See also

`real()`

EXAMPLES:

```
sage: RIF(2, 3).imag()
0
```

```
>>> from sage.all import *
>>> RIF(Integer(2), Integer(3)).imag()
0
```

intersection(*other*)

Return the intersection of two intervals. If the intervals do not overlap, raises a `ValueError`.

EXAMPLES:

```

sage: RIF(1, 2).intersection(RIF(1.5, 3)).str(style='brackets')
'[1.5000000000000000 .. 2.0000000000000000]'
sage: RIF(1, 2).intersection(RIF(4/3, 5/3)).str(style='brackets')
'[1.3333333333333332 .. 1.6666666666666668]'
sage: RIF(1, 2).intersection(RIF(3, 4))
Traceback (most recent call last):
...
ValueError: intersection of non-overlapping intervals

```

```

>>> from sage.all import *
>>> RIF(Integer(1), Integer(2)).intersection(RIF(RealNumber('1.5'),
↳ Integer(3))).str(style='brackets')
'[1.5000000000000000 .. 2.0000000000000000]'
>>> RIF(Integer(1), Integer(2)).intersection(RIF(Integer(4)/Integer(3),
↳ Integer(5)/Integer(3))).str(style='brackets')
'[1.3333333333333332 .. 1.6666666666666668]'
>>> RIF(Integer(1), Integer(2)).intersection(RIF(Integer(3), Integer(4)))
Traceback (most recent call last):
...
ValueError: intersection of non-overlapping intervals

```

is_NaN()

Check to see if self is Not-a-Number NaN.

EXAMPLES:

```

sage: a = RIF(0) / RIF(0.0, 0.00); a
[.. NaN ..]
sage: a.is_NaN()
True

```

```

>>> from sage.all import *
>>> a = RIF(Integer(0)) / RIF(RealNumber('0.0'), RealNumber('0.00')); a
[.. NaN ..]
>>> a.is_NaN()
True

```

is_exact()

Return whether this real interval is exact (i.e. contains exactly one real value).

EXAMPLES:

```

sage: RIF(3).is_exact()
True
sage: RIF(2*pi).is_exact()
↳ needs sage.symbolic #
False

```

```

>>> from sage.all import *
>>> RIF(Integer(3)).is_exact()
True
>>> RIF(Integer(2)*pi).is_exact()

```

(continues on next page)

(continued from previous page)

```

↪      # needs sage.symbolic
False

```

is_int()

Check to see whether this interval includes exactly one integer.

OUTPUT:

If this contains exactly one integer, it returns the tuple `(True, n)`, where `n` is that integer; otherwise, this returns `(False, None)`.

EXAMPLES:

```

sage: a = RIF(0.8, 1.5)
sage: a.is_int()
(True, 1)
sage: a = RIF(1.1, 1.5)
sage: a.is_int()
(False, None)
sage: a = RIF(1, 2)
sage: a.is_int()
(False, None)
sage: a = RIF(-1.1, -0.9)
sage: a.is_int()
(True, -1)
sage: a = RIF(0.1, 1.9)
sage: a.is_int()
(True, 1)
sage: RIF(+infinity, +infinity).is_int()
(False, None)

```

```

>>> from sage.all import *
>>> a = RIF(RealNumber('0.8'), RealNumber('1.5'))
>>> a.is_int()
(True, 1)
>>> a = RIF(RealNumber('1.1'), RealNumber('1.5'))
>>> a.is_int()
(False, None)
>>> a = RIF(Integer(1), Integer(2))
>>> a.is_int()
(False, None)
>>> a = RIF(-RealNumber('1.1'), -RealNumber('0.9'))
>>> a.is_int()
(True, -1)
>>> a = RIF(RealNumber('0.1'), RealNumber('1.9'))
>>> a.is_int()
(True, 1)
>>> RIF(+infinity, +infinity).is_int()
(False, None)

```

lexico_cmp(left, right)

Compare two intervals lexicographically.

This means that the left bounds are compared first and then the right bounds are compared if the left bounds

coincide.

Return 0 if they are the same interval, -1 if the second is larger, or 1 if the first is larger.

EXAMPLES:

```
sage: RIF(0).lexico_cmp(RIF(1))
-1
sage: RIF(0, 1).lexico_cmp(RIF(1))
-1
sage: RIF(0, 1).lexico_cmp(RIF(1, 2))
-1
sage: RIF(0, 0.99999).lexico_cmp(RIF(1, 2))
-1
sage: RIF(1, 2).lexico_cmp(RIF(0, 1))
1
sage: RIF(1, 2).lexico_cmp(RIF(0))
1
sage: RIF(0, 1).lexico_cmp(RIF(0, 2))
-1
sage: RIF(0, 1).lexico_cmp(RIF(0, 1))
0
sage: RIF(0, 1).lexico_cmp(RIF(0, 1/2))
1
```

```
>>> from sage.all import *
>>> RIF(Integer(0)).lexico_cmp(RIF(Integer(1)))
-1
>>> RIF(Integer(0), Integer(1)).lexico_cmp(RIF(Integer(1)))
-1
>>> RIF(Integer(0), Integer(1)).lexico_cmp(RIF(Integer(1), Integer(2)))
-1
>>> RIF(Integer(0), RealNumber('0.99999')).lexico_cmp(RIF(Integer(1),
↪Integer(2)))
-1
>>> RIF(Integer(1), Integer(2)).lexico_cmp(RIF(Integer(0), Integer(1)))
1
>>> RIF(Integer(1), Integer(2)).lexico_cmp(RIF(Integer(0)))
1
>>> RIF(Integer(0), Integer(1)).lexico_cmp(RIF(Integer(0), Integer(2)))
-1
>>> RIF(Integer(0), Integer(1)).lexico_cmp(RIF(Integer(0), Integer(1)))
0
>>> RIF(Integer(0), Integer(1)).lexico_cmp(RIF(Integer(0), Integer(1)/
↪Integer(2)))
1
```

log (*base*='e')

Return the logarithm of *self* to the given *base*.

EXAMPLES:

```
sage: R = RealIntervalField()
sage: r = R(2); r.log()
0.6931471805599453?
```

(continues on next page)

(continued from previous page)

```
sage: r = R(-2); r.log()
0.6931471805599453? + 3.141592653589794?*I
```

```
>>> from sage.all import *
>>> R = RealIntervalField()
>>> r = R(Integer(2)); r.log()
0.6931471805599453?
>>> r = R(-Integer(2)); r.log()
0.6931471805599453? + 3.141592653589794?*I
```

log10()

Return log to the base 10 of self.

EXAMPLES:

```
sage: r = RIF(16.0); r.log10()
1.204119982655925?
sage: r.log() / RIF(10).log()
1.204119982655925?
```

```
>>> from sage.all import *
>>> r = RIF(RealNumber('16.0')); r.log10()
1.204119982655925?
>>> r.log() / RIF(Integer(10)).log()
1.204119982655925?
```

```
sage: r = RIF(39.9); r.log10()
1.600972895686749?
```

```
>>> from sage.all import *
>>> r = RIF(RealNumber('39.9')); r.log10()
1.600972895686749?
```

```
sage: r = RIF(0.0)
sage: r.log10()
[-infinity .. -infinity]
```

```
>>> from sage.all import *
>>> r = RIF(RealNumber('0.0'))
>>> r.log10()
[-infinity .. -infinity]
```

```
sage: r = RIF(-1.0)
sage: r.log10()
1.364376353841841?*I
```

```
>>> from sage.all import *
>>> r = RIF(-RealNumber('1.0'))
>>> r.log10()
1.364376353841841?*I
```

log2()

Return log to the base 2 of self.

EXAMPLES:

```
sage: r = RIF(16.0)
sage: r.log2()
4
```

```
>>> from sage.all import *
>>> r = RIF(RealNumber('16.0'))
>>> r.log2()
4
```

```
sage: r = RIF(31.9); r.log2()
4.995484518877507?
```

```
>>> from sage.all import *
>>> r = RIF(RealNumber('31.9')); r.log2()
4.995484518877507?
```

```
sage: r = RIF(0.0, 2.0)
sage: r.log2()
[-infinity .. 1.0000000000000000]
```

```
>>> from sage.all import *
>>> r = RIF(RealNumber('0.0'), RealNumber('2.0'))
>>> r.log2()
[-infinity .. 1.0000000000000000]
```

lower (*rnd=None*)

Return the lower bound of this interval.

INPUT:

- *rnd* – the rounding mode (default: towards minus infinity, see [sage.rings.real_mpfr.RealField](#) for possible values)

The rounding mode does not affect the value returned as a floating-point number, but it does control which variety of `RealField` the returned number is in, which affects printing and subsequent operations.

EXAMPLES:

```
sage: R = RealIntervalField(13)
sage: R.pi().lower().str()
'3.1411'
```

```
>>> from sage.all import *
>>> R = RealIntervalField(Integer(13))
>>> R.pi().lower().str()
'3.1411'
```

```
sage: x = R(1.2, 1.3); x.str(style='brackets')
'[1.1999 .. 1.3001]'
```

(continues on next page)

(continued from previous page)

```

sage: x.lower()
1.19
sage: x.lower('RNDU')
1.20
sage: x.lower('RNDN')
1.20
sage: x.lower('RNDZ')
1.19
sage: x.lower('RNDA')
1.20
sage: x.lower().parent()
Real Field with 13 bits of precision and rounding RNDD
sage: x.lower('RNDU').parent()
Real Field with 13 bits of precision and rounding RNDU
sage: x.lower('RNDA').parent()
Real Field with 13 bits of precision and rounding RNDA
sage: x.lower() == x.lower('RNDU')
True

```

```

>>> from sage.all import *
>>> x = R(RealNumber('1.2'), RealNumber('1.3')); x.str(style='brackets')
'[1.1999 .. 1.3001]'
>>> x.lower()
1.19
>>> x.lower('RNDU')
1.20
>>> x.lower('RNDN')
1.20
>>> x.lower('RNDZ')
1.19
>>> x.lower('RNDA')
1.20
>>> x.lower().parent()
Real Field with 13 bits of precision and rounding RNDD
>>> x.lower('RNDU').parent()
Real Field with 13 bits of precision and rounding RNDU
>>> x.lower('RNDA').parent()
Real Field with 13 bits of precision and rounding RNDA
>>> x.lower() == x.lower('RNDU')
True

```

magnitude()

The largest absolute value of the elements of the interval.

OUTPUT: a real number with rounding mode RNDU

EXAMPLES:

```

sage: RIF(-2, 1).magnitude()
2.000000000000000
sage: RIF(-1, 2).magnitude()
2.000000000000000
sage: parent(RIF(1).magnitude())

```

(continues on next page)

(continued from previous page)

```
Real Field with 53 bits of precision and rounding RNDU
```

```
>>> from sage.all import *
>>> RIF(-Integer(2), Integer(1)).magnitude()
2.000000000000000
>>> RIF(-Integer(1), Integer(2)).magnitude()
2.000000000000000
>>> parent(RIF(Integer(1)).magnitude())
Real Field with 53 bits of precision and rounding RNDU
```

max (*_others)Return an interval containing the maximum of `self` and the arguments.

EXAMPLES:

```
sage: RIF(-1, 1).max(0).endpoints()
(0.000000000000000, 1.000000000000000)
sage: RIF(-1, 1).max(RIF(2, 3)).endpoints()
(2.000000000000000, 3.000000000000000)
sage: RIF(-1, 1).max(RIF(-100, 100)).endpoints()
(-1.000000000000000, 100.0000000000000)
sage: RIF(-1, 1).max(RIF(-100, 100), RIF(5, 10)).endpoints()
(5.000000000000000, 100.0000000000000)
```

```
>>> from sage.all import *
>>> RIF(-Integer(1), Integer(1)).max(Integer(0)).endpoints()
(0.000000000000000, 1.000000000000000)
>>> RIF(-Integer(1), Integer(1)).max(RIF(Integer(2), Integer(3))).endpoints()
(2.000000000000000, 3.000000000000000)
>>> RIF(-Integer(1), Integer(1)).max(RIF(-Integer(100), Integer(100))).
↳ endpoints()
(-1.000000000000000, 100.0000000000000)
>>> RIF(-Integer(1), Integer(1)).max(RIF(-Integer(100), Integer(100)),
↳ RIF(Integer(5), Integer(10))).endpoints()
(5.000000000000000, 100.0000000000000)
```

Note that if the maximum is one of the given elements, that element will be returned.

```
sage: a = RIF(-1, 1)
sage: b = RIF(2, 3)
sage: c = RIF(3, 4)
sage: c.max(a, b) is c
True
sage: b.max(a, c) is c
True
sage: a.max(b, c) is c
True
```

```
>>> from sage.all import *
>>> a = RIF(-Integer(1), Integer(1))
>>> b = RIF(Integer(2), Integer(3))
>>> c = RIF(Integer(3), Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> c.max(a, b) is c
True
>>> b.max(a, c) is c
True
>>> a.max(b, c) is c
True
```

It might also be convenient to call the method as a function:

```
sage: from sage.rings.real_mpfi import RealIntervalFieldElement
sage: RealIntervalFieldElement.max(a, b, c) is c
True
sage: elements = [a, b, c]
sage: RealIntervalFieldElement.max(*elements) is c
True
```

```
>>> from sage.all import *
>>> from sage.rings.real_mpfi import RealIntervalFieldElement
>>> RealIntervalFieldElement.max(a, b, c) is c
True
>>> elements = [a, b, c]
>>> RealIntervalFieldElement.max(*elements) is c
True
```

The generic max does not always do the right thing:

```
sage: max(0, RIF(-1, 1))
0
sage: max(RIF(-1, 1), RIF(-100, 100)).endpoints()
(-1.000000000000000, 1.000000000000000)
```

```
>>> from sage.all import *
>>> max(Integer(0), RIF(-Integer(1), Integer(1)))
0
>>> max(RIF(-Integer(1), Integer(1)), RIF(-Integer(100), Integer(100))).
↪ endpoints()
(-1.000000000000000, 1.000000000000000)
```

Note that calls involving NaNs try to return a number when possible. This is consistent with IEEE-754-2008 but may be surprising.

```
sage: RIF('nan').max(1, 2)
2
sage: RIF(-1/3).max(RIF('nan'))
-0.3333333333333334?
sage: RIF('nan').max(RIF('nan'))
[.. NaN ..]
```

```
>>> from sage.all import *
>>> RIF('nan').max(Integer(1), Integer(2))
2
>>> RIF(-Integer(1)/Integer(3)).max(RIF('nan'))
```

(continues on next page)

(continued from previous page)

```
-0.3333333333333334?
>>> RIF('nan').max(RIF('nan'))
[... NaN ...]
```

See also`min()`**mignitude()**

The smallest absolute value of the elements of the interval.

OUTPUT: a real number with rounding mode RNDD

EXAMPLES:

```
sage: RIF(-2, 1).mignitude()
0.0000000000000000
sage: RIF(-2, -1).mignitude()
1.0000000000000000
sage: RIF(3, 4).mignitude()
3.0000000000000000
sage: parent(RIF(1).mignitude())
Real Field with 53 bits of precision and rounding RNDD
```

```
>>> from sage.all import *
>>> RIF(-Integer(2), Integer(1)).mignitude()
0.0000000000000000
>>> RIF(-Integer(2), -Integer(1)).mignitude()
1.0000000000000000
>>> RIF(Integer(3), Integer(4)).mignitude()
3.0000000000000000
>>> parent(RIF(Integer(1)).mignitude())
Real Field with 53 bits of precision and rounding RNDD
```

min(*_others)

Return an interval containing the minimum of `self` and the arguments.

EXAMPLES:

```
sage: a = RIF(-1, 1).min(0).endpoints()
sage: a[0] == -1.0 and a[1].abs() == 0.0 # in MPFI, the sign of 0.0 is not
↳ specified
True
sage: RIF(-1, 1).min(pi).endpoints() #
↳ needs sage.symbolic
(-1.0000000000000000, 1.0000000000000000)
sage: RIF(-1, 1).min(RIF(-100, 100)).endpoints()
(-100.00000000000000, 1.0000000000000000)
sage: RIF(-1, 1).min(RIF(-100, 0)).endpoints()
(-100.00000000000000, 0.0000000000000000)
sage: RIF(-1, 1).min(RIF(-100, 2), RIF(-200, -3)).endpoints()
(-200.00000000000000, -3.0000000000000000)
```



```

>>> from sage.all import *
>>> a = RIF(-Integer(1), Integer(1)).min(Integer(0)).endpoints()
>>> a[Integer(0)] == -RealNumber('1.0') and a[Integer(1)].abs() == RealNumber(
↳ '0.0') # in MPFI, the sign of 0.0 is not specified
True
>>> RIF(-Integer(1), Integer(1)).min(pi).endpoints()
↳ # needs sage.symbolic
(-1.000000000000000, 1.000000000000000)
>>> RIF(-Integer(1), Integer(1)).min(RIF(-Integer(100), Integer(100))).
↳ endpoints()
(-100.0000000000000, 1.000000000000000)
>>> RIF(-Integer(1), Integer(1)).min(RIF(-Integer(100), Integer(0))).
↳ endpoints()
(-100.0000000000000, 0.000000000000000)
>>> RIF(-Integer(1), Integer(1)).min(RIF(-Integer(100), Integer(2)), RIF(-
↳ Integer(200), -Integer(3))).endpoints()
(-200.0000000000000, -3.000000000000000)

```

Note that if the minimum is one of the given elements, that element will be returned.

```

sage: a = RIF(-1, 1)
sage: b = RIF(2, 3)
sage: c = RIF(3, 4)
sage: c.min(a, b) is a
True
sage: b.min(a, c) is a
True
sage: a.min(b, c) is a
True

```

```

>>> from sage.all import *
>>> a = RIF(-Integer(1), Integer(1))
>>> b = RIF(Integer(2), Integer(3))
>>> c = RIF(Integer(3), Integer(4))
>>> c.min(a, b) is a
True
>>> b.min(a, c) is a
True
>>> a.min(b, c) is a
True

```

It might also be convenient to call the method as a function:

```

sage: from sage.rings.real_mpfi import RealIntervalFieldElement
sage: RealIntervalFieldElement.min(a, b, c) is a
True
sage: elements = [a, b, c]
sage: RealIntervalFieldElement.min(*elements) is a
True

```

```

>>> from sage.all import *
>>> from sage.rings.real_mpfi import RealIntervalFieldElement
>>> RealIntervalFieldElement.min(a, b, c) is a

```

(continues on next page)

(continued from previous page)

```
True
>>> elements = [a, b, c]
>>> RealIntervalFieldElement.min(*elements) is a
True
```

The generic min does not always do the right thing:

```
sage: min(0, RIF(-1, 1))
0
sage: min(RIF(-1, 1), RIF(-100, 100)).endpoints()
(-1.000000000000000, 1.000000000000000)
```

```
>>> from sage.all import *
>>> min(Integer(0), RIF(-Integer(1), Integer(1)))
0
>>> min(RIF(-Integer(1), Integer(1)), RIF(-Integer(100), Integer(100))).
↳ endpoints()
(-1.000000000000000, 1.000000000000000)
```

Note that calls involving NaNs try to return a number when possible. This is consistent with IEEE-754-2008 but may be surprising.

```
sage: RIF('nan').min(2, 1)
1
sage: RIF(-1/3).min(RIF('nan'))
-0.3333333333333334?
sage: RIF('nan').min(RIF('nan'))
[.. NaN ..]
```

```
>>> from sage.all import *
>>> RIF('nan').min(Integer(2), Integer(1))
1
>>> RIF(-Integer(1)/Integer(3)).min(RIF('nan'))
-0.3333333333333334?
>>> RIF('nan').min(RIF('nan'))
[.. NaN ..]
```

See also

`max()`

`multiplicative_order()`

Return n such that $\text{self}^n == 1$.

Only ± 1 have finite multiplicative order.

EXAMPLES:

```
sage: RIF(1).multiplicative_order()
1
sage: RIF(-1).multiplicative_order()
```

(continues on next page)

(continued from previous page)

```
2
sage: RIF(3).multiplicative_order()
+Infinity
```

```
>>> from sage.all import *
>>> RIF(Integer(1)).multiplicative_order()
1
>>> RIF(-Integer(1)).multiplicative_order()
2
>>> RIF(Integer(3)).multiplicative_order()
+Infinity
```

overlaps (*other*)

Return True if self and other are intervals with at least one value in common. For intervals a and b, we have `a.overlaps(b)` iff not `(a!=b)`.

EXAMPLES:

```
sage: RIF(0, 1).overlaps(RIF(1, 2))
True
sage: RIF(1, 2).overlaps(RIF(0, 1))
True
sage: RIF(0, 1).overlaps(RIF(2, 3))
False
sage: RIF(2, 3).overlaps(RIF(0, 1))
False
sage: RIF(0, 3).overlaps(RIF(1, 2))
True
sage: RIF(0, 2).overlaps(RIF(1, 3))
True
```

```
>>> from sage.all import *
>>> RIF(Integer(0), Integer(1)).overlaps(RIF(Integer(1), Integer(2)))
True
>>> RIF(Integer(1), Integer(2)).overlaps(RIF(Integer(0), Integer(1)))
True
>>> RIF(Integer(0), Integer(1)).overlaps(RIF(Integer(2), Integer(3)))
False
>>> RIF(Integer(2), Integer(3)).overlaps(RIF(Integer(0), Integer(1)))
False
>>> RIF(Integer(0), Integer(3)).overlaps(RIF(Integer(1), Integer(2)))
True
>>> RIF(Integer(0), Integer(2)).overlaps(RIF(Integer(1), Integer(3)))
True
```

prec ()

Return the precision of self.

EXAMPLES:

```
sage: RIF(2.1).precision()
53
```

(continues on next page)

(continued from previous page)

```
sage: RealIntervalField(200)(2.1).precision()
200
```

```
>>> from sage.all import *
>>> RIF(RealNumber('2.1')).precision()
53
>>> RealIntervalField(Integer(200))(RealNumber('2.1')).precision()
200
```

precision()

Return the precision of `self`.

EXAMPLES:

```
sage: RIF(2.1).precision()
53
sage: RealIntervalField(200)(2.1).precision()
200
```

```
>>> from sage.all import *
>>> RIF(RealNumber('2.1')).precision()
53
>>> RealIntervalField(Integer(200))(RealNumber('2.1')).precision()
200
```

psi()

Return the digamma function evaluated on `self`.

OUTPUT: a *RealIntervalFieldElement*

EXAMPLES:

```
sage: psi_1 = RIF(1).psi()
sage: psi_1
-0.577215664901533?
sage: psi_1.overlaps(-RIF.euler_constant())
True
```

```
>>> from sage.all import *
>>> psi_1 = RIF(Integer(1)).psi()
>>> psi_1
-0.577215664901533?
>>> psi_1.overlaps(-RIF.euler_constant())
True
```

real()

Return the real part of this real interval.

(Since this interval is real, this simply returns itself.)

See also

imag()

EXAMPLES:

```
sage: RIF(1.2465).real() == RIF(1.2465)
True
```

```
>>> from sage.all import *
>>> RIF(RealNumber('1.2465')).real() == RIF(RealNumber('1.2465'))
True
```

relative_diameter()

The relative diameter of this interval (for $[a..b]$, this is $(b - a)/((a + b)/2)$), rounded upward, as a *Real-Number*.

EXAMPLES:

```
sage: RIF(1, pi).relative_diameter()
↪needs sage.symbolic
1.03418797197910
```

```
>>> from sage.all import *
>>> RIF(Integer(1), pi).relative_diameter()
↪ # needs sage.symbolic
1.03418797197910
```

round()

Return the nearest integer of this interval as an interval.

See also

- *unique_round()* – return the round as an integer if it is unique and raises a `ValueError` otherwise
- *floor()* – truncation towards $-\infty$
- *ceil()* – truncation towards $+\infty$
- *trunc()* – truncation towards 0

EXAMPLES:

```
sage: RIF(7.2, 7.3).round()
7
sage: RIF(-3.2, -3.1).round()
-3
```

```
>>> from sage.all import *
>>> RIF(RealNumber('7.2'), RealNumber('7.3')).round()
7
>>> RIF(-RealNumber('3.2'), -RealNumber('3.1')).round()
-3
```

Be careful that the answer is not an integer but an interval:

```
sage: RIF(2.2, 2.3).round().parent()
Real Interval Field with 53 bits of precision
```

```
>>> from sage.all import *
>>> RIF(RealNumber('2.2'), RealNumber('2.3')).round().parent()
Real Interval Field with 53 bits of precision
```

And in some cases, the lower and upper bounds of this interval do not agree:

```
sage: r = RIF(2.5, 3.5).round()
sage: r
4.?
sage: r.lower()
3.000000000000000
sage: r.upper()
4.000000000000000
```

```
>>> from sage.all import *
>>> r = RIF(RealNumber('2.5'), RealNumber('3.5')).round()
>>> r
4.?
>>> r.lower()
3.000000000000000
>>> r.upper()
4.000000000000000
```

sec()

Return the secant of this number.

EXAMPLES:

```
sage: RealIntervalField(100)(2).sec()
-2.40299796172238098975460040142?
```

```
>>> from sage.all import *
>>> RealIntervalField(Integer(100))(Integer(2)).sec()
-2.40299796172238098975460040142?
```

sech()

Return the hyperbolic secant of `self`.

EXAMPLES:

```
sage: RealIntervalField(100)(2).sech()
0.265802228834079692120862739820?
```

```
>>> from sage.all import *
>>> RealIntervalField(Integer(100))(Integer(2)).sech()
0.265802228834079692120862739820?
```

simplest_rational(low_open=False, high_open=False)

Return the simplest rational in this interval. Given rationals a/b and c/d (both in lowest terms), the former is simpler if $b < d$ or if $b = d$ and $|a| < |c|$.

If optional parameters `low_open` or `high_open` are `True`, then treat this as an open interval on that end.

EXAMPLES:

```
sage: RealIntervalField(10)(pi).simplest_rational() #_
↪needs sage.symbolic
22/7
sage: RealIntervalField(20)(pi).simplest_rational() #_
↪needs sage.symbolic
355/113
sage: RIF(0.123, 0.567).simplest_rational()
1/2
sage: RIF(RR(1/3).nextabove(), RR(3/7)).simplest_rational()
2/5
sage: RIF(1234/567).simplest_rational()
1234/567
sage: RIF(-8765/432).simplest_rational()
-8765/432
sage: RIF(-1.234, 0.003).simplest_rational()
0
sage: RIF(RR(1/3)).simplest_rational()
6004799503160661/18014398509481984
sage: RIF(RR(1/3)).simplest_rational(high_open=True)
Traceback (most recent call last):
...
ValueError: simplest_rational() on open, empty interval
sage: RIF(1/3, 1/2).simplest_rational()
1/2
sage: RIF(1/3, 1/2).simplest_rational(high_open=True)
1/3
sage: phi = ((RealIntervalField(500)(5).sqrt() + 1)/2)
sage: phi.simplest_rational() == fibonacci(362)/fibonacci(361)
True
```

```
>>> from sage.all import *
>>> RealIntervalField(Integer(10))(pi).simplest_rational() #_
↪ # needs sage.symbolic
22/7
>>> RealIntervalField(Integer(20))(pi).simplest_rational() #_
↪ # needs sage.symbolic
355/113
>>> RIF(RealNumber('0.123'), RealNumber('0.567')).simplest_rational()
1/2
>>> RIF(RR(Integer(1)/Integer(3)).nextabove(), RR(Integer(3)/Integer(7))).
↪simplest_rational()
2/5
>>> RIF(Integer(1234)/Integer(567)).simplest_rational()
1234/567
>>> RIF(-Integer(8765)/Integer(432)).simplest_rational()
-8765/432
>>> RIF(-RealNumber('1.234'), RealNumber('0.003')).simplest_rational()
0
>>> RIF(RR(Integer(1)/Integer(3))).simplest_rational()
6004799503160661/18014398509481984
```

(continues on next page)

(continued from previous page)

```

>>> RIF(RR(Integer(1)/Integer(3))).simplest_rational(high_open=True)
Traceback (most recent call last):
...
ValueError: simplest_rational() on open, empty interval
>>> RIF(Integer(1)/Integer(3), Integer(1)/Integer(2)).simplest_rational()
1/2
>>> RIF(Integer(1)/Integer(3), Integer(1)/Integer(2)).simplest_rational(high_
↪open=True)
1/3
>>> phi = ((RealIntervalField(Integer(500))(Integer(5)).sqrt() + Integer(1))/
↪Integer(2))
>>> phi.simplest_rational() == fibonacci(Integer(362))/fibonacci(Integer(361))
True

```

sin()Return the sine of `self`.

EXAMPLES:

```

sage: R = RealIntervalField(100)
sage: R(2).sin()
0.909297426825681695396019865912?

```

```

>>> from sage.all import *
>>> R = RealIntervalField(Integer(100))
>>> R(Integer(2)).sin()
0.909297426825681695396019865912?

```

sinh()Return the hyperbolic sine of `self`.

EXAMPLES:

```

sage: q = RIF.pi()/12
sage: q.sinh()
0.2648002276022707?

```

```

>>> from sage.all import *
>>> q = RIF.pi()/Integer(12)
>>> q.sinh()
0.2648002276022707?

```

sqrt()Return a square root of `self`. Raises an error if `self` is nonpositive.If you use `square_root()` then an interval will always be returned (though it will be NaN if `self` is non-positive).

EXAMPLES:

```

sage: r = RIF(4.0)
sage: r.sqrt()
2

```

(continues on next page)

(continued from previous page)

```
sage: r.sqrt()^2 == r
True
```

```
>>> from sage.all import *
>>> r = RIF(RealNumber('4.0'))
>>> r.sqrt()
2
>>> r.sqrt()**Integer(2) == r
True
```

```
sage: r = RIF(4344)
sage: r.sqrt()
65.90902821313633?
sage: r.sqrt()^2 == r
False
sage: r in r.sqrt()^2
True
sage: r.sqrt()^2 - r
0.?e-11
sage: (r.sqrt()^2 - r).str(style='brackets')
'[-9.0949470177292824e-13 .. 1.8189894035458565e-12]'
```

```
>>> from sage.all import *
>>> r = RIF(Integer(4344))
>>> r.sqrt()
65.90902821313633?
>>> r.sqrt()**Integer(2) == r
False
>>> r in r.sqrt()**Integer(2)
True
>>> r.sqrt()**Integer(2) - r
0.?e-11
>>> (r.sqrt()**Integer(2) - r).str(style='brackets')
'[-9.0949470177292824e-13 .. 1.8189894035458565e-12]'
```

```
sage: r = RIF(-2.0)
sage: r.sqrt()
Traceback (most recent call last):
...
ValueError: self (=-2) is not >= 0
```

```
>>> from sage.all import *
>>> r = RIF(-RealNumber('2.0'))
>>> r.sqrt()
Traceback (most recent call last):
...
ValueError: self (=-2) is not >= 0
```

```
sage: r = RIF(-2, 2)
sage: r.sqrt()
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: self (=0.?e1) is not >= 0
```

```
>>> from sage.all import *
>>> r = RIF(-Integer(2), Integer(2))
>>> r.sqrt()
Traceback (most recent call last):
...
ValueError: self (=0.?e1) is not >= 0
```

square()Return the square of `self`.**Note**

Squaring an interval is different than multiplying it by itself, because the square can never be negative.

EXAMPLES:

```
sage: RIF(1, 2).square().str(style='brackets')
'[1.0000000000000000 .. 4.0000000000000000]'
sage: RIF(-1, 1).square().str(style='brackets')
'[0.0000000000000000 .. 1.0000000000000000]'
sage: (RIF(-1, 1) * RIF(-1, 1)).str(style='brackets')
'[-1.0000000000000000 .. 1.0000000000000000]'
```

```
>>> from sage.all import *
>>> RIF(Integer(1), Integer(2)).square().str(style='brackets')
'[1.0000000000000000 .. 4.0000000000000000]'
>>> RIF(-Integer(1), Integer(1)).square().str(style='brackets')
'[0.0000000000000000 .. 1.0000000000000000]'
>>> (RIF(-Integer(1), Integer(1)) * RIF(-Integer(1), Integer(1))).str(style=
↪ 'brackets')
'[-1.0000000000000000 .. 1.0000000000000000]'
```

square_root()Return a square root of `self`. An interval will always be returned (though it will be NaN if `self` is nonpositive).

EXAMPLES:

```
sage: r = RIF(-2.0)
sage: r.square_root()
[.. NaN ..]
sage: r.sqrt()
Traceback (most recent call last):
...
ValueError: self (=-2) is not >= 0
```

```

>>> from sage.all import *
>>> r = RIF(-RealNumber('2.0'))
>>> r.square_root()
[.. NaN ..]
>>> r.sqrt()
Traceback (most recent call last):
...
ValueError: self (=-2) is not >= 0

```

str (*base=10, style=None, no_sci=None, e=None, error_digits=None*)

Return a string representation of self.

INPUT:

- *base* – base for output
- *style* – the printing style; either 'brackets' or 'question' (or None, to use the current default)
- *no_sci* – if True do not print using scientific notation; if False print with scientific notation; if None (the default), print how the parent prints.
- *e* – symbol used in scientific notation
- *error_digits* – the number of digits of error to print, in 'question' style

We support two different styles of printing; 'question' style and 'brackets' style. In question style (the default), we print the “known correct” part of the number, followed by a question mark:

```

sage: RIF(pi).str()                                     #_
↪needs sage.symbolic
'3.141592653589794?'
sage: RIF(pi, 22/7).str()                               #_
↪needs sage.symbolic
'3.142?'
sage: RIF(pi, 22/7).str(style='question')               #_
↪needs sage.symbolic
'3.142?'

```

```

>>> from sage.all import *
>>> RIF(pi).str()                                     #_
↪needs sage.symbolic
'3.141592653589794?'
>>> RIF(pi, Integer(22)/Integer(7)).str()               _
↪                               # needs sage.symbolic
'3.142?'
>>> RIF(pi, Integer(22)/Integer(7)).str(style='question') _
↪                               # needs sage.symbolic
'3.142?'

```

However, if the interval is precisely equal to some integer that's not too large, we just return that integer:

```

sage: RIF(-42).str()
'-42'
sage: RIF(0).str()
'0'

```

(continues on next page)

(continued from previous page)

```
sage: RIF(12^5).str(base=3)
'110122100000'
```

```
>>> from sage.all import *
>>> RIF(-Integer(42)).str()
'-42'
>>> RIF(Integer(0)).str()
'0'
>>> RIF(Integer(12)**Integer(5)).str(base=Integer(3))
'110122100000'
```

Very large integers, however, revert to the normal question-style printing:

```
sage: RIF(3^7).str()
'2187'
sage: RIF(3^7 * 2^256).str()
'2.5323729916201052?e80'
```

```
>>> from sage.all import *
>>> RIF(Integer(3)**Integer(7)).str()
'2187'
>>> RIF(Integer(3)**Integer(7) * Integer(2)**Integer(256)).str()
'2.5323729916201052?e80'
```

In brackets style, we print the lower and upper bounds of the interval within brackets:

```
sage: RIF(237/16).str(style='brackets')
'[14.812500000000000 .. 14.812500000000000]'
```

```
>>> from sage.all import *
>>> RIF(Integer(237)/Integer(16)).str(style='brackets')
'[14.812500000000000 .. 14.812500000000000]'
```

Note that the lower bound is rounded down, and the upper bound is rounded up. So even if the lower and upper bounds are equal, they may print differently. (This is done so that the printed representation of the interval contains all the numbers in the internal binary interval.)

For instance, we find the best 10-bit floating point representation of $1/3$:

```
sage: RR10 = RealField(10)
sage: RR(RR10(1/3))
0.333496093750000
```

```
>>> from sage.all import *
>>> RR10 = RealField(Integer(10))
>>> RR(RR10(Integer(1)/Integer(3)))
0.333496093750000
```

And we see that the point interval containing only this floating-point number prints as a wider decimal interval, that does contain the number:

```
sage: RIF10 = RealIntervalField(10)
sage: RIF10(RR10(1/3)).str(style='brackets')
'[0.33349 .. 0.33350]'
```

```
>>> from sage.all import *
>>> RIF10 = RealIntervalField(Integer(10))
>>> RIF10(RR10(Integer(1)/Integer(3))).str(style='brackets')
'[0.33349 .. 0.33350]'
```

We always use brackets style for NaN and infinities:

```
sage: RIF(pi, infinity) #_
↪needs sage.symbolic
[3.1415926535897931 .. +infinity]
sage: RIF(NaN) #_
↪needs sage.symbolic
[.. NaN ..]
```

```
>>> from sage.all import *
>>> RIF(pi, infinity) #_
↪needs sage.symbolic
[3.1415926535897931 .. +infinity]
>>> RIF(NaN) #_
↪needs sage.symbolic
[.. NaN ..]
```

Let's take a closer, formal look at the question style. In its full generality, a number printed in the question style looks like:

MANTISSA ?ERROR eEXPONENT

(without the spaces). The “eEXPONENT” part is optional; if it is missing, then the exponent is 0. (If the base is greater than 10, then the exponent separator is “@” instead of “e”.)

The “ERROR” is optional; if it is missing, then the error is 1.

The mantissa is printed in base b , and always contains a decimal point (also known as a radix point, in bases other than 10). (The error and exponent are always printed in base 10.)

We define the “precision” of a floating-point printed representation to be the positional value of the last digit of the mantissa. For instance, in $2.7?e5$, the precision is 10^4 ; in $8.?$, the precision is 10^0 ; and in $9.35?$ the precision is 10^{-2} . This precision will always be 10^k for some k (or, for an arbitrary base b , b^k).

Then the interval is contained in the interval:

$$\text{mantissa} \cdot b^{\text{exponent}} - \text{error} \cdot b^k \dots \text{mantissa} \cdot b^{\text{exponent}} + \text{error} \cdot b^k$$

To control the printing, we can specify a maximum number of error digits. The default is 0, which means that we do not print an error at all (so that the error is always the default, 1).

Now, consider the precisions needed to represent the endpoints (this is the precision that would be produced by `v.lower().str(no_sci=False)`). Our result is no more precise than the less precise endpoint, and is sufficiently imprecise that the error can be represented with the given number of decimal digits. Our result is the most precise possible result, given these restrictions. When there are two possible results of equal precision and with the same error width, then we pick the one which is farther from zero. (For instance, `RIF(0, 123)` with two error digits could print as $61.?62$ or $62.?62$. We prefer the latter because it makes it clear that the interval is known not to be negative.)

EXAMPLES:

```

sage: a = RIF(59/27); a
2.185185185185186?
sage: a.str()
'2.185185185185186?'
sage: a.str(style='brackets')
'[2.1851851851851851 .. 2.1851851851851856]'
sage: a.str(16)
'2.2f684bda12f69?'
sage: a.str(no_sci=False)
'2.185185185185186?e0'
sage: pi_appr = RIF(pi, 22/7)
sage: pi_appr.str(style='brackets')
'[3.1415926535897931 .. 3.1428571428571433]'
sage: pi_appr.str()
'3.142?'
sage: pi_appr.str(error_digits=1)
'3.1422?7'
sage: pi_appr.str(error_digits=2)
'3.14223?64'
sage: pi_appr.str(base=36)
'3.6?'
sage: RIF(NaN)
↳needs sage.symbolic
[.. NaN ..]
sage: RIF(pi, infinity)
↳needs sage.symbolic
[3.1415926535897931 .. +infinity]
sage: RIF(-infinity, pi)
↳needs sage.symbolic
[-infinity .. 3.1415926535897936]
sage: RealIntervalField(210)(3).sqrt()
1.732050807568877293527446341505872366942805253810380628055806980?
sage: RealIntervalField(210)(RIF(3).sqrt())
1.732050807568878?
sage: RIF(3).sqrt()
1.732050807568878?
sage: RIF(0, 3^-150)
↳needs sage.symbolic
1.?e-71

```

```

>>> from sage.all import *
>>> a = RIF(Integer(59)/Integer(27)); a
2.185185185185186?
>>> a.str()
'2.185185185185186?'
>>> a.str(style='brackets')
'[2.1851851851851851 .. 2.1851851851851856]'
>>> a.str(Integer(16))
'2.2f684bda12f69?'
>>> a.str(no_sci=False)
'2.185185185185186?e0'

```

(continues on next page)

(continued from previous page)

```

>>> pi_appr = RIF(pi, Integer(22)/Integer(7))
>>> pi_appr.str(style='brackets')
'[3.1415926535897931 .. 3.1428571428571433]'
>>> pi_appr.str()
'3.142?'
>>> pi_appr.str(error_digits=Integer(1))
'3.1422?7'
>>> pi_appr.str(error_digits=Integer(2))
'3.14223?64'
>>> pi_appr.str(base=Integer(36))
'3.6?'
>>> RIF(NaN) #_
↳needs sage.symbolic
[.. NaN ..]
>>> RIF(pi, infinity) #_
↳needs sage.symbolic
[3.1415926535897931 .. +infinity]
>>> RIF(-infinity, pi) #_
↳needs sage.symbolic
[-infinity .. 3.1415926535897936]
>>> RealIntervalField(Integer(210))(Integer(3)).sqrt()
1.732050807568877293527446341505872366942805253810380628055806980?
>>> RealIntervalField(Integer(210))(RIF(Integer(3)).sqrt())
1.732050807568878?
>>> RIF(Integer(3)).sqrt()
1.732050807568878?
>>> RIF(Integer(0), Integer(3)**-Integer(150)) # needs sage.symbolic
↳
1.?e-71

```

tan()Return the tangent of `self`.

EXAMPLES:

```

sage: q = RIF.pi()/3
sage: q.tan()
1.732050807568877?
sage: q = RIF.pi()/6
sage: q.tan()
0.577350269189626?

```

```

>>> from sage.all import *
>>> q = RIF.pi()/Integer(3)
>>> q.tan()
1.732050807568877?
>>> q = RIF.pi()/Integer(6)
>>> q.tan()
0.577350269189626?

```

tanh()Return the hyperbolic tangent of `self`.

EXAMPLES:

```
sage: q = RIF.pi()/11
sage: q.tanh()
0.2780794292958503?
```

```
>>> from sage.all import *
>>> q = RIF.pi()/Integer(11)
>>> q.tanh()
0.2780794292958503?
```

trunc()

Return the truncation of this interval as an interval.

The truncation of x is the floor of x if x is nonnegative or the ceil of x if x is negative.

See also

- `unique_trunc()` – return the trunc as an integer if it is unique and raises a `ValueError` otherwise
- `floor()` – truncation towards $-\infty$
- `ceil()` – truncation towards $+\infty$
- `round()` – rounding

EXAMPLES:

```
sage: RIF(2.3, 2.7).trunc()
2
sage: parent(_)
Real Interval Field with 53 bits of precision

sage: RIF(-0.9, 0.9).trunc()
0
sage: RIF(-7.5, -7.3).trunc()
-7
```

```
>>> from sage.all import *
>>> RIF(RealNumber('2.3'), RealNumber('2.7')).trunc()
2
>>> parent(_)
Real Interval Field with 53 bits of precision

>>> RIF(-RealNumber('0.9'), RealNumber('0.9')).trunc()
0
>>> RIF(-RealNumber('7.5'), -RealNumber('7.3')).trunc()
-7
```

In the above example, the obtained interval contains only one element. But on the following it is not the case anymore:


```
sage: r = RIF(2.99, 3.01).trunc()
sage: r.upper()
3.000000000000000
sage: r.lower()
2.000000000000000
```

```
>>> from sage.all import *
>>> r = RIF(RealNumber('2.99'), RealNumber('3.01')).trunc()
>>> r.upper()
3.000000000000000
>>> r.lower()
2.000000000000000
```

union (*other*)

Return the union of two intervals, or of an interval and a real number (more precisely, the convex hull).

EXAMPLES:

```
sage: RIF(1, 2).union(RIF(pi, 22/7)).str(style='brackets')
'[1.000000000000000 .. 3.1428571428571433]'
sage: RIF(1, 2).union(pi).str(style='brackets')
'[1.000000000000000 .. 3.1415926535897936]'
sage: RIF(1).union(RIF(0, 2)).str(style='brackets')
'[0.000000000000000 .. 2.000000000000000]'
sage: RIF(1).union(RIF(-1)).str(style='brackets')
'[-1.000000000000000 .. 1.000000000000000]'
```

```
>>> from sage.all import *
>>> RIF(Integer(1), Integer(2)).union(RIF(pi, Integer(22)/Integer(7))).
↳str(style='brackets')
'[1.000000000000000 .. 3.1428571428571433]'
>>> RIF(Integer(1), Integer(2)).union(pi).str(style='brackets')
'[1.000000000000000 .. 3.1415926535897936]'
>>> RIF(Integer(1)).union(RIF(Integer(0), Integer(2))).str(style='brackets')
'[0.000000000000000 .. 2.000000000000000]'
>>> RIF(Integer(1)).union(RIF(-Integer(1))).str(style='brackets')
'[-1.000000000000000 .. 1.000000000000000]'
```

unique_ceil()

Return the unique ceiling of this interval, if it is well defined, otherwise raise a `ValueError`.

OUTPUT: integer

See also

`ceil()` – return the ceil as an interval (and never raise error)

EXAMPLES:

```
sage: RIF(pi).unique_ceil()
↳needs sage.symbolic
4
```

#

(continues on next page)

(continued from previous page)

```
sage: RIF(100*pi).unique_ceil()
↳needs sage.symbolic
315
sage: RIF(100, 200).unique_ceil()
Traceback (most recent call last):
...
ValueError: interval does not have a unique ceil
```

```
>>> from sage.all import *
>>> RIF(pi).unique_ceil()
↳needs sage.symbolic
4
>>> RIF(Integer(100)*pi).unique_ceil()
↳ # needs sage.symbolic
315
>>> RIF(Integer(100), Integer(200)).unique_ceil()
Traceback (most recent call last):
...
ValueError: interval does not have a unique ceil
```

unique_floor()

Return the unique floor of this interval, if it is well defined, otherwise raise a `ValueError`.

OUTPUT: integer

See also

`floor()` – return the floor as an interval (and never raise error)

EXAMPLES:

```
sage: RIF(pi).unique_floor()
↳needs sage.symbolic
3
sage: RIF(100*pi).unique_floor()
↳needs sage.symbolic
314
sage: RIF(100, 200).unique_floor()
Traceback (most recent call last):
...
ValueError: interval does not have a unique floor
```

```
>>> from sage.all import *
>>> RIF(pi).unique_floor()
↳needs sage.symbolic
3
>>> RIF(Integer(100)*pi).unique_floor()
↳ # needs sage.symbolic
314
>>> RIF(Integer(100), Integer(200)).unique_floor()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: interval does not have a unique floor
```

unique_integer()

Return the unique integer in this interval, if there is exactly one, otherwise raise a `ValueError`.

EXAMPLES:

```
sage: RIF(pi).unique_integer()                                     #_
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: interval contains no integer
sage: RIF(pi, pi+1).unique_integer()                             #_
↪needs sage.symbolic
4
sage: RIF(pi, pi+2).unique_integer()                             #_
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: interval contains more than one integer
sage: RIF(100).unique_integer()
100
```

```
>>> from sage.all import *
>>> RIF(pi).unique_integer()                                     #_
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: interval contains no integer
>>> RIF(pi, pi+Integer(1)).unique_integer()                     _
↪      # needs sage.symbolic
4
>>> RIF(pi, pi+Integer(2)).unique_integer()                     _
↪      # needs sage.symbolic
Traceback (most recent call last):
...
ValueError: interval contains more than one integer
>>> RIF(Integer(100)).unique_integer()
100
```

unique_round()

Return the unique round (nearest integer) of this interval, if it is well defined, otherwise raise a `ValueError`.

OUTPUT: integer

See also

`round()` – return the round as an interval (and never raise error)

EXAMPLES:

```

sage: RIF(pi).unique_round()
↪ # needs sage.symbolic
3
sage: RIF(1000*pi).unique_round()
↪ # needs sage.symbolic
3142
sage: RIF(100, 200).unique_round()
Traceback (most recent call last):
...
ValueError: interval does not have a unique round (nearest integer)
sage: RIF(1.2, 1.7).unique_round()
Traceback (most recent call last):
...
ValueError: interval does not have a unique round (nearest integer)
sage: RIF(0.7, 1.2).unique_round()
1
sage: RIF(-pi).unique_round()
↪ # needs sage.symbolic
-3
sage: (RIF(4.5).unique_round(), RIF(-4.5).unique_round())
(5, -5)

```

```

>>> from sage.all import *
>>> RIF(pi).unique_round()
↪ # needs sage.symbolic
3
>>> RIF(Integer(1000)*pi).unique_round()
↪ # needs sage.symbolic
3142
>>> RIF(Integer(100), Integer(200)).unique_round()
Traceback (most recent call last):
...
ValueError: interval does not have a unique round (nearest integer)
>>> RIF(RealNumber('1.2'), RealNumber('1.7')).unique_round()
Traceback (most recent call last):
...
ValueError: interval does not have a unique round (nearest integer)
>>> RIF(RealNumber('0.7'), RealNumber('1.2')).unique_round()
1
>>> RIF(-pi).unique_round()
↪ # needs sage.symbolic
-3
>>> (RIF(RealNumber('4.5')).unique_round(), RIF(-RealNumber('4.5')).
↪ unique_round())
(5, -5)

```

unique_sign()

Return the sign of this element if it is well defined.

This method returns +1 if all elements in this interval are positive, -1 if all of them are negative and 0 if it contains only zero. Otherwise it raises a `ValueError`.

EXAMPLES:

```

sage: RIF(1.2, 5.7).unique_sign()
1
sage: RIF(-3, -2).unique_sign()
-1
sage: RIF(0).unique_sign()
0
sage: RIF(0, 1).unique_sign()
Traceback (most recent call last):
...
ValueError: interval does not have a unique sign
sage: RIF(-1, 0).unique_sign()
Traceback (most recent call last):
...
ValueError: interval does not have a unique sign
sage: RIF(-0.1, 0.1).unique_sign()
Traceback (most recent call last):
...
ValueError: interval does not have a unique sign

```

```

>>> from sage.all import *
>>> RIF(RealNumber('1.2'), RealNumber('5.7')).unique_sign()
1
>>> RIF(-Integer(3), -Integer(2)).unique_sign()
-1
>>> RIF(Integer(0)).unique_sign()
0
>>> RIF(Integer(0), Integer(1)).unique_sign()
Traceback (most recent call last):
...
ValueError: interval does not have a unique sign
>>> RIF(-Integer(1), Integer(0)).unique_sign()
Traceback (most recent call last):
...
ValueError: interval does not have a unique sign
>>> RIF(-RealNumber('0.1'), RealNumber('0.1')).unique_sign()
Traceback (most recent call last):
...
ValueError: interval does not have a unique sign

```

unique_trunc()

Return the nearest integer toward zero if it is unique, otherwise raise a `ValueError`.

See also

`trunc()` – return the truncation as an interval (and never raise error)

EXAMPLES:

```

sage: RIF(1.3, 1.4).unique_trunc()
1
sage: RIF(-3.3, -3.2).unique_trunc()

```

(continues on next page)

(continued from previous page)

```

-3
sage: RIF(2.9,3.2).unique_trunc()
Traceback (most recent call last):
...
ValueError: interval does not have a unique trunc (nearest integer toward
↳zero)
sage: RIF(-3.1,-2.9).unique_trunc()
Traceback (most recent call last):
...
ValueError: interval does not have a unique trunc (nearest integer toward
↳zero)

```

```

>>> from sage.all import *
>>> RIF(RealNumber('1.3'), RealNumber('1.4')).unique_trunc()
1
>>> RIF(-RealNumber('3.3'), -RealNumber('3.2')).unique_trunc()
-3
>>> RIF(RealNumber('2.9'), RealNumber('3.2')).unique_trunc()
Traceback (most recent call last):
...
ValueError: interval does not have a unique trunc (nearest integer toward
↳zero)
>>> RIF(-RealNumber('3.1'), -RealNumber('2.9')).unique_trunc()
Traceback (most recent call last):
...
ValueError: interval does not have a unique trunc (nearest integer toward
↳zero)

```

upper (*rnd=None*)

Return the upper bound of self.

INPUT:

- *rnd* – the rounding mode (default: towards plus infinity, see [sage.rings.real_mpfr.RealField](#) for possible values)

The rounding mode does not affect the value returned as a floating-point number, but it does control which variety of `RealField` the returned number is in, which affects printing and subsequent operations.

EXAMPLES:

```

sage: R = RealIntervalField(13)
sage: R.pi().upper().str()
'3.1417'

```

```

>>> from sage.all import *
>>> R = RealIntervalField(Integer(13))
>>> R.pi().upper().str()
'3.1417'

```

```

sage: R = RealIntervalField(13)
sage: x = R(1.2,1.3); x.str(style='brackets')
'[1.1999 .. 1.3001]'

```

(continues on next page)

(continued from previous page)

```

sage: x.upper()
1.31
sage: x.upper('RNDU')
1.31
sage: x.upper('RNDN')
1.30
sage: x.upper('RNDD')
1.30
sage: x.upper('RNDZ')
1.30
sage: x.upper('RNDA')
1.31
sage: x.upper().parent()
Real Field with 13 bits of precision and rounding RNDU
sage: x.upper('RNDD').parent()
Real Field with 13 bits of precision and rounding RNDD
sage: x.upper() == x.upper('RNDD')
True

```

```

>>> from sage.all import *
>>> R = RealIntervalField(Integer(13))
>>> x = R(RealNumber('1.2'), RealNumber('1.3')); x.str(style='brackets')
'[1.1999 .. 1.3001]'
>>> x.upper()
1.31
>>> x.upper('RNDU')
1.31
>>> x.upper('RNDN')
1.30
>>> x.upper('RNDD')
1.30
>>> x.upper('RNDZ')
1.30
>>> x.upper('RNDA')
1.31
>>> x.upper().parent()
Real Field with 13 bits of precision and rounding RNDU
>>> x.upper('RNDD').parent()
Real Field with 13 bits of precision and rounding RNDD
>>> x.upper() == x.upper('RNDD')
True

```

zeta ($a=None$)

Return the image of this interval by the Hurwitz zeta function.

For $a = 1$ (or $a = None$), this computes the Riemann zeta function.

EXAMPLES:

```

sage: zeta(RIF(3))
1.202056903159594?
sage: _.parent()
Real Interval Field with 53 bits of precision

```

(continues on next page)

(continued from previous page)

```
sage: RIF(3).zeta(1/2)
8.41439832211716?
```

```
>>> from sage.all import *
>>> zeta(RIF(Integer(3)))
1.202056903159594?
>>> _.parent()
Real Interval Field with 53 bits of precision
>>> RIF(Integer(3)).zeta(Integer(1)/Integer(2))
8.41439832211716?
```

class sage.rings.real_mpf.**RealIntervalField_class**

Bases: `RealIntervalField`

Class of the real interval field.

INPUT:

- `prec` – integer (default: 53); precision `prec` is the number of bits used to represent the mantissa of a floating-point number. The precision can be any integer between `mpfr_prec_min()` and `mpfr_prec_max()`. In the current implementation, `mpfr_prec_min()` is equal to 2.
- `sci_not` – boolean (default: False); whether or not to display using scientific notation

EXAMPLES:

```
sage: RealIntervalField(10)
Real Interval Field with 10 bits of precision
sage: RealIntervalField()
Real Interval Field with 53 bits of precision
sage: RealIntervalField(100000)
Real Interval Field with 100000 bits of precision
```

```
>>> from sage.all import *
>>> RealIntervalField(Integer(10))
Real Interval Field with 10 bits of precision
>>> RealIntervalField()
Real Interval Field with 53 bits of precision
>>> RealIntervalField(Integer(100000))
Real Interval Field with 100000 bits of precision
```

Note

The default precision is 53, since according to the GMP manual: ‘mpfr should be able to exactly reproduce all computations with double-precision machine floating-point numbers (double type in C), except the default exponent range is much wider and subnormal numbers are not implemented.’

EXAMPLES:

Creation of elements.

First with default precision. First we coerce elements of various types, then we coerce intervals:


```

sage: RIF = RealIntervalField(); RIF
Real Interval Field with 53 bits of precision
sage: RIF(3)
3
sage: RIF(RIF(3))
3
sage: RIF(pi)
↳needs sage.symbolic
3.141592653589794?
sage: RIF(RealField(53)('1.5'))
1.5000000000000000?
sage: RIF(-2/19)
-0.1052631578947369?
sage: RIF(-3939)
-3939
sage: RIF(-3939r)
-3939
sage: RIF('1.5')
1.5000000000000000?
sage: R200 = RealField(200)
sage: RIF(R200.pi())
3.141592653589794?
sage: RIF(10^100)
1.0000000000000000?e100

```

```

>>> from sage.all import *
>>> RIF = RealIntervalField(); RIF
Real Interval Field with 53 bits of precision
>>> RIF(Integer(3))
3
>>> RIF(RIF(Integer(3)))
3
>>> RIF(pi)
↳needs sage.symbolic
3.141592653589794?
>>> RIF(RealField(Integer(53))('1.5'))
1.5000000000000000?
>>> RIF(-Integer(2)/Integer(19))
-0.1052631578947369?
>>> RIF(-Integer(3939))
-3939
>>> RIF(-3939)
-3939
>>> RIF('1.5')
1.5000000000000000?
>>> R200 = RealField(Integer(200))
>>> RIF(R200.pi())
3.141592653589794?
>>> RIF(Integer(10)**Integer(100))
1.0000000000000000?e100

```

The base must be explicitly specified as a named parameter:

```
sage: RIF('101101', base=2)
45
sage: RIF('+infinity')
[+infinity .. +infinity]
sage: RIF('[1..3]').str(style='brackets')
'[1.0000000000000000 .. 3.0000000000000000]'
```

```
>>> from sage.all import *
>>> RIF('101101', base=Integer(2))
45
>>> RIF('+infinity')
[+infinity .. +infinity]
>>> RIF('[1..3]').str(style='brackets')
'[1.0000000000000000 .. 3.0000000000000000]'
```

All string-like types are accepted:

```
sage: RIF(b"100", u"100")
100
```

```
>>> from sage.all import *
>>> RIF(b"100", u"100")
100
```

Next we coerce some 2-tuples, which define intervals:

```
sage: RIF((-1.5, -1.3))
-1.4?
sage: RIF((RDF('-1.5'), RDF('-1.3')))
-1.4?
sage: RIF((1/3, 2/3)).str(style='brackets')
'[0.3333333333333331 .. 0.66666666666666675]'
```

```
>>> from sage.all import *
>>> RIF((-RealNumber('1.5'), -RealNumber('1.3')))
-1.4?
>>> RIF((RDF('-1.5'), RDF('-1.3')))
-1.4?
>>> RIF((Integer(1)/Integer(3), Integer(2)/Integer(3))).str(style='brackets')
'[0.3333333333333331 .. 0.66666666666666675]'
```

The extra parentheses aren't needed:

```
sage: RIF(1/3, 2/3).str(style='brackets')
'[0.3333333333333331 .. 0.66666666666666675]'
sage: RIF((1, 2)).str(style='brackets')
'[1.0000000000000000 .. 2.0000000000000000]'
sage: RIF((1r, 2r)).str(style='brackets')
'[1.0000000000000000 .. 2.0000000000000000]'
sage: RIF((pi, e)).str(style='brackets')
'[2.7182818284590450 .. 3.1415926535897936]'
```

```

>>> from sage.all import *
>>> RIF(Integer(1)/Integer(3), Integer(2)/Integer(3)).str(style='brackets')
'[0.3333333333333333 .. 0.6666666666666667]'
>>> RIF((Integer(1), Integer(2))).str(style='brackets')
'[1.0000000000000000 .. 2.0000000000000000]'
>>> RIF((1, 2)).str(style='brackets')
'[1.0000000000000000 .. 2.0000000000000000]'
>>> RIF((pi, e)).str(style='brackets')
'[2.7182818284590450 .. 3.1415926535897936]'

```

Values which can be represented as an exact floating-point number (of the precision of this `RealIntervalField`) result in a precise interval, where the lower bound is equal to the upper bound (even if they print differently). Other values typically result in an interval where the lower and upper bounds are adjacent floating-point numbers.

```

sage: def check(x):
....:     return (x, x.lower() == x.upper())
sage: check(RIF(pi))                                     #_
↪needs sage.symbolic
(3.141592653589794?, False)
sage: check(RIF(RR(pi)))                                 #_
↪needs sage.symbolic
(3.1415926535897932?, True)
sage: check(RIF(1.5))
(1.5000000000000000?, True)
sage: check(RIF('1.5'))
(1.5000000000000000?, True)
sage: check(RIF(0.1))
(0.10000000000000001?, True)
sage: check(RIF(1/10))
(0.10000000000000000?, False)
sage: check(RIF('0.1'))
(0.10000000000000000?, False)

```

```

>>> from sage.all import *
>>> def check(x):
...     return (x, x.lower() == x.upper())
>>> check(RIF(pi))                                     #_
↪needs sage.symbolic
(3.141592653589794?, False)
>>> check(RIF(RR(pi)))                                 #_
↪needs sage.symbolic
(3.1415926535897932?, True)
>>> check(RIF(RealNumber('1.5')))
(1.5000000000000000?, True)
>>> check(RIF('1.5'))
(1.5000000000000000?, True)
>>> check(RIF(RealNumber('0.1')))
(0.10000000000000001?, True)
>>> check(RIF(Integer(1)/Integer(10)))
(0.10000000000000000?, False)
>>> check(RIF('0.1'))
(0.10000000000000000?, False)

```

Similarly, when specifying both ends of an interval, the lower end is rounded down and the upper end is rounded up:

```
sage: outward = RIF(1/10, 7/10); outward.str(style='brackets')
'[0.09999999999999991 .. 0.70000000000000007]'
sage: nearest = RIF(RR(1/10), RR(7/10)); nearest.str(style='brackets')
'[0.10000000000000000 .. 0.69999999999999996]'
sage: nearest.lower() - outward.lower()
1.38777878078144e-17
sage: outward.upper() - nearest.upper()
1.11022302462516e-16
```

```
>>> from sage.all import *
>>> outward = RIF(Integer(1)/Integer(10), Integer(7)/Integer(10)); outward.
↳str(style='brackets')
'[0.09999999999999991 .. 0.70000000000000007]'
>>> nearest = RIF(RR(Integer(1)/Integer(10)), RR(Integer(7)/Integer(10)));
↳nearest.str(style='brackets')
'[0.10000000000000000 .. 0.69999999999999996]'
>>> nearest.lower() - outward.lower()
1.38777878078144e-17
>>> outward.upper() - nearest.upper()
1.11022302462516e-16
```

Some examples with a real interval field of higher precision:

```
sage: R = RealIntervalField(100)
sage: R(3)
3
sage: R(R(3))
3
sage: R(pi)
↳needs sage.symbolic #_
3.14159265358979323846264338328?
sage: R(-2/19)
-0.1052631578947368421052631578948?
sage: R(e,pi).str(style='brackets')
↳needs sage.symbolic #_
'[2.7182818284590452353602874713512 .. 3.1415926535897932384626433832825]'
```

```
>>> from sage.all import *
>>> R = RealIntervalField(Integer(100))
>>> R(Integer(3))
3
>>> R(R(Integer(3)))
3
>>> R(pi)
↳needs sage.symbolic #_
3.14159265358979323846264338328?
>>> R(-Integer(2)/Integer(19))
-0.1052631578947368421052631578948?
>>> R(e,pi).str(style='brackets')
↳needs sage.symbolic #_
'[2.7182818284590452353602874713512 .. 3.1415926535897932384626433832825]'
```

(continues on next page)

(continued from previous page)

```
'[2.7182818284590452353602874713512 .. 3.1415926535897932384626433832825]'
```

See also

- `sage.rings.real_mpf`
- `sage.rings.complex_interval_field`
- `sage.rings.real_arb.RealBallField` (alternative implementation of real intervals, with more features)

Element

alias of `RealIntervalFieldElement`

algebraic_closure()

Return the algebraic closure of this interval field, i.e., the complex interval field with the same precision.

EXAMPLES:

```
sage: RIF.algebraic_closure()
Complex Interval Field with 53 bits of precision
sage: RIF.algebraic_closure() is CIF
True
sage: RealIntervalField(100).algebraic_closure()
Complex Interval Field with 100 bits of precision
```

```
>>> from sage.all import *
>>> RIF.algebraic_closure()
Complex Interval Field with 53 bits of precision
>>> RIF.algebraic_closure() is CIF
True
>>> RealIntervalField(Integer(100)).algebraic_closure()
Complex Interval Field with 100 bits of precision
```

characteristic()

Return 0, since the field of real numbers has characteristic 0.

EXAMPLES:

```
sage: RealIntervalField(10).characteristic()
0
```

```
>>> from sage.all import *
>>> RealIntervalField(Integer(10)).characteristic()
0
```

complex_field()

Return complex field of the same precision.

EXAMPLES:

```
sage: RIF.complex_field()
Complex Interval Field with 53 bits of precision
```

```
>>> from sage.all import *
>>> RIF.complex_field()
Complex Interval Field with 53 bits of precision
```

construction()

Return the functorial construction of `self`, namely, completion of the rational numbers with respect to the prime at ∞ , and the note that this is an interval field.

Also preserves other information that makes this field unique (e.g. precision, print mode).

EXAMPLES:

```
sage: R = RealIntervalField(123)
sage: c, S = R.construction(); S
Rational Field
sage: R == c(S)
True
```

```
>>> from sage.all import *
>>> R = RealIntervalField(Integer(123))
>>> c, S = R.construction(); S
Rational Field
>>> R == c(S)
True
```

euler_constant()

Return Euler's gamma constant to the precision of this field.

EXAMPLES:

```
sage: RealIntervalField(100).euler_constant()
0.577215664901532860606512090083?
```

```
>>> from sage.all import *
>>> RealIntervalField(Integer(100)).euler_constant()
0.577215664901532860606512090083?
```

gen(i=0)

Return the i -th generator of `self`.

EXAMPLES:

```
sage: RIF.gen(0)
1
sage: RIF.gen(1)
Traceback (most recent call last):
...
IndexError: self has only one generator
```

```
>>> from sage.all import *
>>> RIF.gen(Integer(0))
1
>>> RIF.gen(Integer(1))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
IndexError: self has only one generator
```

gens()

Return a tuple of generators.

EXAMPLES:

```
sage: RIF.gens()
(1,)
```

```
>>> from sage.all import *
>>> RIF.gens()
(1,)
```

is_exact()

Return whether or not this field is exact, which is always False.

EXAMPLES:

```
sage: RIF.is_exact()
False
```

```
>>> from sage.all import *
>>> RIF.is_exact()
False
```

log2()Return $\log(2)$ to the precision of this field.

EXAMPLES:

```
sage: R=RealIntervalField(100)
sage: R.log2()
0.693147180559945309417232121458?
sage: R(2).log()
0.693147180559945309417232121458?
```

```
>>> from sage.all import *
>>> R=RealIntervalField(Integer(100))
>>> R.log2()
0.693147180559945309417232121458?
>>> R(Integer(2)).log()
0.693147180559945309417232121458?
```

lower_field()

Return the RealField_class with rounding mode 'RNDD' (rounding towards minus infinity).

EXAMPLES:

```
sage: RIF.lower_field()
Real Field with 53 bits of precision and rounding RNDD
sage: RealIntervalField(200).lower_field()
Real Field with 200 bits of precision and rounding RNDD
```

```
>>> from sage.all import *
>>> RIF.lower_field()
Real Field with 53 bits of precision and rounding RNDD
>>> RealIntervalField(Integer(200)).lower_field()
Real Field with 200 bits of precision and rounding RNDD
```

middle_field()

Return the RealField_class with rounding mode 'RNDN' (rounding towards nearest).

EXAMPLES:

```
sage: RIF.middle_field()
Real Field with 53 bits of precision
sage: RealIntervalField(200).middle_field()
Real Field with 200 bits of precision
```

```
>>> from sage.all import *
>>> RIF.middle_field()
Real Field with 53 bits of precision
>>> RealIntervalField(Integer(200)).middle_field()
Real Field with 200 bits of precision
```

name()

Return the name of self.

EXAMPLES:

```
sage: RIF.name()
'IntervalRealIntervalField53'
sage: RealIntervalField(200).name()
'IntervalRealIntervalField200'
```

```
>>> from sage.all import *
>>> RIF.name()
'IntervalRealIntervalField53'
>>> RealIntervalField(Integer(200)).name()
'IntervalRealIntervalField200'
```

ngens()

Return the number of generators of self, which is 1.

EXAMPLES:

```
sage: RIF.ngens()
1
```

```
>>> from sage.all import *
>>> RIF.ngens()
1
```

pi()

Return π to the precision of this field.

EXAMPLES:


```

sage: R = RealIntervalField(100)
sage: R.pi()
3.14159265358979323846264338328?
sage: R.pi().sqrt()/2
0.88622692545275801364908374167?
sage: R = RealIntervalField(150)
sage: R.pi().sqrt()/2
0.886226925452758013649083741670572591398774728?

```

```

>>> from sage.all import *
>>> R = RealIntervalField(Integer(100))
>>> R.pi()
3.14159265358979323846264338328?
>>> R.pi().sqrt()/Integer(2)
0.88622692545275801364908374167?
>>> R = RealIntervalField(Integer(150))
>>> R.pi().sqrt()/Integer(2)
0.886226925452758013649083741670572591398774728?

```

prec()

Return the precision of this field (in bits).

EXAMPLES:

```

sage: RIF.precision()
53
sage: RealIntervalField(200).precision()
200

```

```

>>> from sage.all import *
>>> RIF.precision()
53
>>> RealIntervalField(Integer(200)).precision()
200

```

precision()

Return the precision of this field (in bits).

EXAMPLES:

```

sage: RIF.precision()
53
sage: RealIntervalField(200).precision()
200

```

```

>>> from sage.all import *
>>> RIF.precision()
53
>>> RealIntervalField(Integer(200)).precision()
200

```

random_element(*args, **kws)

Return a random element of `self`. Any arguments or keywords are passed onto the random element function in real field.

By default, this is uniformly distributed in $[-1, 1]$.

EXAMPLES:

```
sage: RIF.random_element().parent() is RIF
True
sage: -100 <= RIF.random_element(-100, 100) <= 100
True
```

```
>>> from sage.all import *
>>> RIF.random_element().parent() is RIF
True
>>> -Integer(100) <= RIF.random_element(-Integer(100), Integer(100)) <=
↳ Integer(100)
True
```

Passes extra positional or keyword arguments through:

```
sage: 0 <= RIF.random_element(min=0, max=100) <= 100
True
sage: -100 <= RIF.random_element(min=-100, max=0) <= 0
True
```

```
>>> from sage.all import *
>>> Integer(0) <= RIF.random_element(min=Integer(0), max=Integer(100)) <=
↳ Integer(100)
True
>>> -Integer(100) <= RIF.random_element(min=-Integer(100), max=Integer(0)) <=
↳ Integer(0)
True
```

scientific_notation (*status=None*)

Set or return the scientific notation printing flag.

If this flag is `True` then real numbers with this space as parent print using scientific notation.

INPUT:

- `status` – boolean optional flag

EXAMPLES:

```
sage: RIF(0.025)
0.025000000000000002?
sage: RIF.scientific_notation(True)
sage: RIF(0.025)
2.5000000000000002?e-2
sage: RIF.scientific_notation(False)
sage: RIF(0.025)
0.025000000000000002?
```

```
>>> from sage.all import *
>>> RIF(RealNumber('0.025'))
0.025000000000000002?
>>> RIF.scientific_notation(True)
```

(continues on next page)

(continued from previous page)

```
>>> RIF(RealNumber('0.025'))
2.50000000000000002?e-2
>>> RIF.scientific_notation(False)
>>> RIF(RealNumber('0.025'))
0.025000000000000002?
```

to_prec(prec)

Return a real interval field to the given precision.

EXAMPLES:

```
sage: RIF.to_prec(200)
Real Interval Field with 200 bits of precision
sage: RIF.to_prec(20)
Real Interval Field with 20 bits of precision
sage: RIF.to_prec(53) is RIF
True
```

```
>>> from sage.all import *
>>> RIF.to_prec(Integer(200))
Real Interval Field with 200 bits of precision
>>> RIF.to_prec(Integer(20))
Real Interval Field with 20 bits of precision
>>> RIF.to_prec(Integer(53)) is RIF
True
```

upper_field()

Return the RealField_class with rounding mode 'RNDU' (rounding towards plus infinity).

EXAMPLES:

```
sage: RIF.upper_field()
Real Field with 53 bits of precision and rounding RNDU
sage: RealIntervalField(200).upper_field()
Real Field with 200 bits of precision and rounding RNDU
```

```
>>> from sage.all import *
>>> RIF.upper_field()
Real Field with 53 bits of precision and rounding RNDU
>>> RealIntervalField(Integer(200)).upper_field()
Real Field with 200 bits of precision and rounding RNDU
```

zeta(n=2)

Return an n -th root of unity in the real field, if one exists, or raise a `ValueError` otherwise.

EXAMPLES:

```
sage: R = RealIntervalField()
sage: R.zeta()
-1
sage: R.zeta(1)
1
sage: R.zeta(5)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: No 5th root of unity in self
```

```
>>> from sage.all import *
>>> R = RealIntervalField()
>>> R.zeta()
-1
>>> R.zeta(Integer(1))
1
>>> R.zeta(Integer(5))
Traceback (most recent call last):
...
ValueError: No 5th root of unity in self
```

`sage.rings.real_mpfi.is_RealIntervalField(x)`

Check if x is a *RealIntervalField_class*.

EXAMPLES:

```
sage: sage.rings.real_mpfi.is_RealIntervalField(RIF)
doctest:warning...
DeprecationWarning: The function is_RealIntervalField is deprecated;
use 'isinstance(..., RealIntervalField_class)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
sage: sage.rings.real_mpfi.is_RealIntervalField(RealIntervalField(200))
True
```

```
>>> from sage.all import *
>>> sage.rings.real_mpfi.is_RealIntervalField(RIF)
doctest:warning...
DeprecationWarning: The function is_RealIntervalField is deprecated;
use 'isinstance(..., RealIntervalField_class)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
>>> sage.rings.real_mpfi.is_RealIntervalField(RealIntervalField(Integer(200)))
True
```

`sage.rings.real_mpfi.is_RealIntervalFieldElement(x)`

Check if x is a *RealIntervalFieldElement*.

EXAMPLES:

```
sage: sage.rings.real_mpfi.is_RealIntervalFieldElement(RIF(2.2))
doctest:warning...
DeprecationWarning: The function is_RealIntervalFieldElement is deprecated;
use 'isinstance(..., RealIntervalFieldElement)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
sage: sage.rings.real_mpfi.is_RealIntervalFieldElement(RealIntervalField(200)(2.
↪2))
True
```

```
>>> from sage.all import *
>>> sage.rings.real_mphi.is_RealIntervalFieldElement(RIF(RealNumber('2.2')))
doctest:warning...
DeprecationWarning: The function is_RealIntervalFieldElement is deprecated;
use 'isinstance(..., RealIntervalFieldElement)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
>>> sage.rings.real_mphi.is_
↪RealIntervalFieldElement(RealIntervalField(Integer(200))(RealNumber('2.2')))
True
```

2.2 Real intervals with a fixed absolute precision

class sage.rings.real_interval_absolute.**Factory**

Bases: `UniqueFactory`

create_key(*prec*)

The only piece of data is the precision.

create_object(*version*, *prec*)

Ensures uniqueness.

class sage.rings.real_interval_absolute.**MpfrOp**

Bases: `object`

This class is used to endow absolute real interval field elements with all the methods of (relative) real interval field elements.

EXAMPLES:

```
sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(100)
sage: R(1).sin()
0.841470984807896506652502321631?
```

```
>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(100))
>>> R(Integer(1)).sin()
0.841470984807896506652502321631?
```

class sage.rings.real_interval_absolute.**RealIntervalAbsoluteElement**

Bases: `FieldElement`

Create a *RealIntervalAbsoluteElement*.

EXAMPLES:

```
sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(50)
sage: R(1)
1
sage: R(1/3)
0.33333333333333334?
```

(continues on next page)

(continued from previous page)

```

sage: R(1.3)
1.3000000000000000?
sage: R(pi)
3.141592653589794?
sage: R((11, 12))
12.?
sage: R((11, 11.00001))
11.00001?

sage: R100 = RealIntervalAbsoluteField(100)
sage: R(R100((5, 6)))
6.?
sage: R100(R((5, 6)))
6.?
sage: RIF(CIF(NaN))
[... NaN ...]

```

```

>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(50))
>>> R(Integer(1))
1
>>> R(Integer(1)/Integer(3))
0.33333333333333334?
>>> R(RealNumber('1.3'))
1.3000000000000000?
>>> R(pi)
3.141592653589794?
>>> R((Integer(11), Integer(12)))
12.?
>>> R((Integer(11), RealNumber('11.00001')))
11.00001?

>>> R100 = RealIntervalAbsoluteField(Integer(100))
>>> R(R100((Integer(5), Integer(6))))
6.?
>>> R100(R((Integer(5), Integer(6))))
6.?
>>> RIF(CIF(NaN))
[... NaN ...]

```

abs()Return the absolute value of `self`.

EXAMPLES:

```

sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(100)
sage: R(1/3).abs()
0.333333333333333333333333333333333333333333333334?
sage: R(-1/3).abs()
0.333333333333333333333333333333333333333333333334?

```

(continues on next page)

```
sage: R((-1/3, 1/2)).abs()  
1.  
sage: R((-1/3, 1/2)).abs().endpoints()  
(0, 1/2)  
sage: R((-3/2, 1/2)).abs().endpoints()  
(0, 3/2)
```

Return the diameter `self`.

Return whether `self` contains zero.

```

sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(10)
sage: R(10).contains_zero()
False
sage: R((10,11)).contains_zero()
False
sage: R((0,11)).contains_zero()
True
sage: R((-10,11)).contains_zero()
True
sage: R((-10,-1)).contains_zero()
False
sage: R((-10,0)).contains_zero()
True
sage: R(pi).contains_zero()
False

```

```

>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(10))
>>> R(Integer(10)).contains_zero()
False
>>> R((Integer(10), Integer(11))).contains_zero()
False
>>> R((Integer(0), Integer(11))).contains_zero()
True
>>> R((-Integer(10), Integer(11))).contains_zero()
True
>>> R((-Integer(10), -Integer(1))).contains_zero()
False
>>> R((-Integer(10), Integer(0))).contains_zero()
True
>>> R(pi).contains_zero()
False

```

diameter()

Return the diameter self.

EXAMPLES:

```

sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(10)
sage: R(1/4).absolute_diameter()
0
sage: a = R(pi)
sage: a.absolute_diameter()
1/1024
sage: a.upper() - a.lower()
1/1024

```

```

>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(10))

```

(continues on next page)

(continued from previous page)

```

>>> R(Integer(1)/Integer(4)).absolute_diameter()
0
>>> a = R(pi)
>>> a.absolute_diameter()
1/1024
>>> a.upper() - a.lower()
1/1024

```

endpoints()

Return the left and right endpoints of `self`, as a tuple.

EXAMPLES:

```

sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(10)
sage: R(1/4).endpoints()
(1/4, 1/4)
sage: R((1,2)).endpoints()
(1, 2)

```

```

>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(10))
>>> R(Integer(1)/Integer(4)).endpoints()
(1/4, 1/4)
>>> R((Integer(1), Integer(2))).endpoints()
(1, 2)

```

is_negative()

Return whether `self` is definitely negative.

EXAMPLES:

```

sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(100)
sage: R(10).is_negative()
False
sage: R((10,11)).is_negative()
False
sage: R((0,11)).is_negative()
False
sage: R((-10,11)).is_negative()
False
sage: R((-10,-1)).is_negative()
True
sage: R(pi).is_negative()
False

```

```

>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(100))
>>> R(Integer(10)).is_negative()

```

(continues on next page)

(continued from previous page)

```

False
>>> R((Integer(10), Integer(11))).is_negative()
False
>>> R((Integer(0), Integer(11))).is_negative()
False
>>> R((-Integer(10), Integer(11))).is_negative()
False
>>> R((-Integer(10), -Integer(1))).is_negative()
True
>>> R(pi).is_negative()
False

```

is_positive()

Return whether `self` is definitely positive.

EXAMPLES:

```

sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(10)
sage: R(10).is_positive()
True
sage: R((10, 11)).is_positive()
True
sage: R((0, 11)).is_positive()
False
sage: R((-10, 11)).is_positive()
False
sage: R((-10, -1)).is_positive()
False
sage: R(pi).is_positive()
True

```

```

>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(10))
>>> R(Integer(10)).is_positive()
True
>>> R((Integer(10), Integer(11))).is_positive()
True
>>> R((Integer(0), Integer(11))).is_positive()
False
>>> R((-Integer(10), Integer(11))).is_positive()
False
>>> R((-Integer(10), -Integer(1))).is_positive()
False
>>> R(pi).is_positive()
True

```

lower()

Return the lower bound of `self`.

EXAMPLES:

```
sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(50)
sage: R(1/4).lower()
1/4
```

```
>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(50))
>>> R(Integer(1)/Integer(4)).lower()
1/4
```

midpoint()

Return the midpoint of `self`.

EXAMPLES:

```
sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(100)
sage: R(1/4).midpoint()
1/4
sage: R(pi).midpoint()
7964883625991394727376702227905/2535301200456458802993406410752
sage: R(pi).midpoint().n()
3.14159265358979
```

```
>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(100))
>>> R(Integer(1)/Integer(4)).midpoint()
1/4
>>> R(pi).midpoint()
7964883625991394727376702227905/2535301200456458802993406410752
>>> R(pi).midpoint().n()
3.14159265358979
```

mpfi_prec()

Return the precision needed to represent this value as an mpfi interval.

EXAMPLES:

```
sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(10)
sage: R(10).mpfi_prec()
14
sage: R(1000).mpfi_prec()
20
```

```
>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(10))
>>> R(Integer(10)).mpfi_prec()
14
```

(continues on next page)

(continued from previous page)

```
>>> R(Integer(1000)).mpfi_prec()
20
```

sqrt()

Return the square root of self.

EXAMPLES:

```
sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(100)
sage: R(2).sqrt()
1.414213562373095048801688724210?
sage: R((4,9)).sqrt().endpoints()
(2, 3)
```

```
>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(100))
>>> R(Integer(2)).sqrt()
1.414213562373095048801688724210?
>>> R((Integer(4), Integer(9))).sqrt().endpoints()
(2, 3)
```

upper()

Return the upper bound of self.

EXAMPLES:

```
sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(50)
sage: R(1/4).upper()
1/4
```

```
>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(50))
>>> R(Integer(1)/Integer(4)).upper()
1/4
```

sage.rings.real_interval_absolute.**RealIntervalAbsoluteField**(*args, **kwargs)

This field is similar to the *RealIntervalField* except instead of truncating everything to a fixed relative precision, it maintains a fixed absolute precision.

Note that unlike the standard real interval field, elements in this field can have different size and experience coefficient blowup. On the other hand, it avoids precision loss on addition and subtraction. This is useful for, e.g., series computations for special functions.

EXAMPLES:

```
sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(10); R
Real Interval Field with absolute precision 2^-10
sage: R(3/10)
```

(continues on next page)

(continued from previous page)

```
0.300?
sage: R(1000003/10)
100000.300?
sage: R(1e100) + R(1) - R(1e100)
1
```

```
>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(10)); R
Real Interval Field with absolute precision 2^-10
>>> R(Integer(3)/Integer(10))
0.300?
>>> R(Integer(1000003)/Integer(10))
100000.300?
>>> R(RealNumber('1e100')) + R(Integer(1)) - R(RealNumber('1e100'))
1
```

```
class sage.rings.real_interval_absolute.RealIntervalAbsoluteField_class
```

Bases: `Field`

This field is similar to the `RealIntervalField` except instead of truncating everything to a fixed relative precision, it maintains a fixed absolute precision.

Note that unlike the standard real interval field, elements in this field can have different size and experience coefficient blowup. On the other hand, it avoids precision loss on addition and subtraction. This is useful for, e.g., series computations for special functions.

EXAMPLES:

```
sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(10); R
Real Interval Field with absolute precision 2^-10
sage: R(3/10)
0.300?
sage: R(1000003/10)
100000.300?
sage: R(1e100) + R(1) - R(1e100)
1
```

```
>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(10)); R
Real Interval Field with absolute precision 2^-10
>>> R(Integer(3)/Integer(10))
0.300?
>>> R(Integer(1000003)/Integer(10))
100000.300?
>>> R(RealNumber('1e100')) + R(Integer(1)) - R(RealNumber('1e100'))
1
```

`absprec()`

Return the absolute precision of `self`.

EXAMPLES:

```
sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: R = RealIntervalAbsoluteField(100)
sage: R.absprec()
100
sage: RealIntervalAbsoluteField(5).absprec()
5
```

```
>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
>>> R = RealIntervalAbsoluteField(Integer(100))
>>> R.absprec()
100
>>> RealIntervalAbsoluteField(Integer(5)).absprec()
5
```

`sage.rings.real_interval_absolute.shift_ceil( $x$ ,  $shift$ )`

Return $x/2^s$ where s is the value of `shift`, rounded towards $+\infty$. For internal use.

EXAMPLES:

```
sage: from sage.rings.real_interval_absolute import shift_ceil
sage: shift_ceil(15, 2)
4
sage: shift_ceil(-15, 2)
-3
sage: shift_ceil(32, 2)
8
sage: shift_ceil(-32, 2)
-8
```

```
>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import shift_ceil
>>> shift_ceil(Integer(15), Integer(2))
4
>>> shift_ceil(-Integer(15), Integer(2))
-3
>>> shift_ceil(Integer(32), Integer(2))
8
>>> shift_ceil(-Integer(32), Integer(2))
-8
```

`sage.rings.real_interval_absolute.shift_floor( $x$ ,  $shift$ )`

Return $x/2^s$ where s is the value of `shift`, rounded towards $-\infty$. For internal use.

EXAMPLES:

```
sage: from sage.rings.real_interval_absolute import shift_floor
sage: shift_floor(15, 2)
3
sage: shift_floor(-15, 2)
-4
```

```
>>> from sage.all import *
>>> from sage.rings.real_interval_absolute import shift_floor
>>> shift_floor(Integer(15), Integer(2))
3
>>> shift_floor(-Integer(15), Integer(2))
-4
```

2.3 Arbitrary precision complex intervals (parent class)

AUTHORS:

- William Stein wrote `complex_field.py`.
- William Stein (2006-01-26): complete rewrite

Then `complex_field.py` was copied to `complex_interval_field.py` and heavily modified:

- Carl Witty (2007-10-24): rewrite for intervals
- Niles Johnson (2010-08): [Issue #3893](#): `random_element()` should pass on `*args` and `**kwargs`.
- Travis Scrimshaw (2012-10-18): Added documentation to get full coverage.

Note

The `ComplexIntervalField` differs from `ComplexField` in that `ComplexIntervalField` only gives the digits with exact precision, then a `?` signifying that the last digit can have an error of ± 1 .

`sage.rings.complex_interval_field.ComplexIntervalField(prec=53, names=None)`

Return the complex interval field with real and imaginary parts having `prec` bits of precision.

EXAMPLES:

```
sage: ComplexIntervalField()
Complex Interval Field with 53 bits of precision
sage: ComplexIntervalField(100)
Complex Interval Field with 100 bits of precision
sage: ComplexIntervalField(100).base_ring()
Real Interval Field with 100 bits of precision
sage: i = ComplexIntervalField(200).gen()
sage: i^2
-1
sage: i^i
0.207879576350761908546955619834978770033877841631769608075136?
```

```
>>> from sage.all import *
>>> ComplexIntervalField()
Complex Interval Field with 53 bits of precision
>>> ComplexIntervalField(Integer(100))
Complex Interval Field with 100 bits of precision
>>> ComplexIntervalField(Integer(100)).base_ring()
Real Interval Field with 100 bits of precision
>>> i = ComplexIntervalField(Integer(200)).gen()
>>> i**Integer(2)
```

(continues on next page)

(continued from previous page)

```
-1
>>> i**i
0.207879576350761908546955619834978770033877841631769608075136?
```

```
class sage.rings.complex_interval_field.ComplexIntervalField_class(prec=53)
```

Bases: `ComplexIntervalField`

The field of complex (interval) numbers.

EXAMPLES:

```
sage: C = ComplexIntervalField(); C
Complex Interval Field with 53 bits of precision
sage: Q = RationalField()
sage: C(1/3)
0.3333333333333334?
sage: C(1/3, 2)
0.3333333333333334? + 2*I
```

```
>>> from sage.all import *
>>> C = ComplexIntervalField(); C
Complex Interval Field with 53 bits of precision
>>> Q = RationalField()
>>> C(Integer(1)/Integer(3))
0.3333333333333334?
>>> C(Integer(1)/Integer(3), Integer(2))
0.3333333333333334? + 2*I
```

We can also coerce rational numbers and integers into `C`, but coercing a polynomial will raise an exception:

```
sage: Q = RationalField()
sage: C(1/3)
0.3333333333333334?
sage: S.<x> = PolynomialRing(Q)
sage: C(x)
Traceback (most recent call last):
...
TypeError: cannot convert nonconstant polynomial
```

```
>>> from sage.all import *
>>> Q = RationalField()
>>> C(Integer(1)/Integer(3))
0.3333333333333334?
>>> S = PolynomialRing(Q, names=('x',)); (x,) = S._first_ngens(1)
>>> C(x)
Traceback (most recent call last):
...
TypeError: cannot convert nonconstant polynomial
```

This illustrates precision:

```
sage: CIF = ComplexIntervalField(10); CIF(1/3, 2/3)
0.334? + 0.667?*I
```

(continues on next page)

[illegible]

```
sage: saved_z = loads(z.dumps())
sage: saved_z.endpoints() == z.endpoints()
True
sage: loads(CIF.dumps()) == CIF
True
sage: k = ComplexIntervalField(100)
sage: loads(dumps(k)) == k
True
```

```
sage: CIF = ComplexIntervalField(200)
sage: CIF.is_field()
True
sage: CIF.characteristic()
0
sage: CIF.precision()
200
sage: CIF.variable_name()
'I'
sage: CIF == ComplexIntervalField(200)
True
```

2.3. Arbitrary precision complex intervals (parent class) 285

(continued from previous page)

```

sage: CIF == ComplexIntervalField(53)
False
sage: CIF == 1.1
False
sage: CIF = ComplexIntervalField(53)

sage: CIF.category()
Category of infinite fields
sage: TestSuite(CIF).run(skip='_test_gcd_vs_xgcd')

```

```

>>> from sage.all import *
>>> CIF = ComplexIntervalField(Integer(200))
>>> CIF.is_field()
True
>>> CIF.characteristic()
0
>>> CIF.precision()
200
>>> CIF.variable_name()
'I'
>>> CIF == ComplexIntervalField(Integer(200))
True
>>> CIF == ComplexIntervalField(Integer(53))
False
>>> CIF == RealNumber('1.1')
False
>>> CIF = ComplexIntervalField(Integer(53))

>>> CIF.category()
Category of infinite fields
>>> TestSuite(CIF).run(skip='_test_gcd_vs_xgcd')

```

See also

- `sage.rings.real_mpmi`
- `sage.rings.complex_arb.ComplexBallField` (alternative implementation of complex intervals, with more features)

Element

alias of `ComplexIntervalFieldElement`

characteristic()

Return the characteristic of the complex (interval) field, which is 0.

EXAMPLES:

```

sage: CIF.characteristic()
0

```

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

construction()

Return the functorial construction of this complex interval field, namely as the algebraic closure of the real interval field with the same precision.

EXAMPLES:

```
sage: c, S = CIF.construction(); c, S
(AlgebraicClosureFunctor,
 Real Interval Field with 53 bits of precision)
sage: CIF == c(S)
True
```

```
>>> from sage.all import *
>>> c, S = CIF.construction(); c, S
(AlgebraicClosureFunctor,
 Real Interval Field with 53 bits of precision)
>>> CIF == c(S)
True
```

gen($n=0$)

Return the generator of the complex (interval) field.

EXAMPLES:

```
sage: CIF.0
1*I
sage: CIF.gen(0)
1*I
```

```
>>> from sage.all import *
>>> CIF.gen(0)
1*I
>>> CIF.gen(Integer(0))
1*I
```

is_exact()

The complex interval field is not exact.

EXAMPLES:

```
sage: CIF.is_exact()
False
```

```
>>> from sage.all import *
>>> CIF.is_exact()
False
```

is_field($proof=True$)

Return True, since the complex numbers are a field.

EXAMPLES:

```
sage: CIF.is_field()
True
```

```
>>> from sage.all import *
>>> CIF.is_field()
True
```

middle_field()

Return the corresponding *ComplexField* with the same precision as *self*.

EXAMPLES:

```
sage: CIF.middle_field()
Complex Field with 53 bits of precision
sage: ComplexIntervalField(200).middle_field()
Complex Field with 200 bits of precision
```

```
>>> from sage.all import *
>>> CIF.middle_field()
Complex Field with 53 bits of precision
>>> ComplexIntervalField(Integer(200)).middle_field()
Complex Field with 200 bits of precision
```

ngens()

The number of generators of this complex (interval) field as an **R**-algebra.

There is one generator, namely $\sqrt{-1}$.

EXAMPLES:

```
sage: CIF.ngens()
1
```

```
>>> from sage.all import *
>>> CIF.ngens()
1
```

pi()

Return π as an element in the complex (interval) field.

EXAMPLES:

```
sage: ComplexIntervalField(100).pi()
3.14159265358979323846264338328?
```

```
>>> from sage.all import *
>>> ComplexIntervalField(Integer(100)).pi()
3.14159265358979323846264338328?
```

prec()

Return the precision of *self* (in bits).

EXAMPLES:

```
sage: CIF.prec()
53
sage: ComplexIntervalField(200).prec()
200
```

```
>>> from sage.all import *
>>> CIF.prec()
53
>>> ComplexIntervalField(Integer(200)).prec()
200
```

precision()

Return the precision of `self` (in bits).

EXAMPLES:

```
sage: CIF.prec()
53
sage: ComplexIntervalField(200).prec()
200
```

```
>>> from sage.all import *
>>> CIF.prec()
53
>>> ComplexIntervalField(Integer(200)).prec()
200
```

random_element(*args, **kws)

Create a random element of `self`.

This simply chooses the real and imaginary part randomly, passing arguments and keywords to the underlying real interval field.

EXAMPLES:

```
sage: CIF.random_element().parent() is CIF
True
sage: re, im = CIF.random_element(10, 20)
sage: 10 <= re <= 20
True
sage: 10 <= im <= 20
True
```

```
>>> from sage.all import *
>>> CIF.random_element().parent() is CIF
True
>>> re, im = CIF.random_element(Integer(10), Integer(20))
>>> Integer(10) <= re <= Integer(20)
True
>>> Integer(10) <= im <= Integer(20)
True
```

Passes extra positional or keyword arguments through:

```
sage: re, im = CIF.random_element(max=0, min=-5)
sage: -5 <= re <= 0
True
sage: -5 <= im <= 0
True
```

```
>>> from sage.all import *
>>> re, im = CIF.random_element(max=Integer(0), min=-Integer(5))
>>> -Integer(5) <= re <= Integer(0)
True
>>> -Integer(5) <= im <= Integer(0)
True
```

real_field()

Return the underlying *RealIntervalField*.

EXAMPLES:

```
sage: R = CIF.real_field(); R
Real Interval Field with 53 bits of precision
sage: ComplexIntervalField(200).real_field()
Real Interval Field with 200 bits of precision
sage: CIF.real_field() is R
True
```

```
>>> from sage.all import *
>>> R = CIF.real_field(); R
Real Interval Field with 53 bits of precision
>>> ComplexIntervalField(Integer(200)).real_field()
Real Interval Field with 200 bits of precision
>>> CIF.real_field() is R
True
```

scientific_notation (*status=None*)

Set or return the scientific notation printing flag.

If this flag is `True` then complex numbers with this space as parent print using scientific notation.

EXAMPLES:

```
sage: CIF((0.025, 2))
0.025000000000000002? + 2*I
sage: CIF.scientific_notation(True)
sage: CIF((0.025, 2))
2.5000000000000002?e-2 + 2*I
sage: CIF.scientific_notation(False)
sage: CIF((0.025, 2))
0.025000000000000002? + 2*I
```

```
>>> from sage.all import *
>>> CIF((RealNumber('0.025'), Integer(2)))
0.025000000000000002? + 2*I
>>> CIF.scientific_notation(True)
```

(continues on next page)

(continued from previous page)

```
>>> CIF((RealNumber('0.025'), Integer(2)))
2.50000000000000002?e-2 + 2*I
>>> CIF.scientific_notation(False)
>>> CIF((RealNumber('0.025'), Integer(2)))
0.025000000000000002? + 2*I
```

to_prec (*prec*)

Return a complex interval field with the given precision.

EXAMPLES:

```
sage: CIF.to_prec(150)
Complex Interval Field with 150 bits of precision
sage: CIF.to_prec(15)
Complex Interval Field with 15 bits of precision
sage: CIF.to_prec(53) is CIF
True
```

```
>>> from sage.all import *
>>> CIF.to_prec(Integer(150))
Complex Interval Field with 150 bits of precision
>>> CIF.to_prec(Integer(15))
Complex Interval Field with 15 bits of precision
>>> CIF.to_prec(Integer(53)) is CIF
True
```

zeta (*n=2*)Return a primitive *n*-th root of unity.**Todo**Implement *ComplexIntervalFieldElement* multiplicative order and set this output to have multiplicative order *n*.

INPUT:

- *n* – integer (default: 2)

OUTPUT: a complex *n*-th root of unity

EXAMPLES:

```
sage: CIF.zeta(2)
-1
sage: CIF.zeta(5)
0.309016994374948? + 0.9510565162951536?*I
```

```
>>> from sage.all import *
>>> CIF.zeta(Integer(2))
-1
>>> CIF.zeta(Integer(5))
0.309016994374948? + 0.9510565162951536?*I
```

2.4 Arbitrary precision complex intervals

This is a simple complex interval package, using intervals which are axis-aligned rectangles in the complex plane. It has very few special functions, and it does not use any special tricks to keep the size of the intervals down.

AUTHORS:

These authors wrote `complex_mpfpr.pyx` (renamed from `complex_number.pyx`):

```
- William Stein (2006-01-26): complete rewrite
- Joel B. Mohler (2006-12-16): naive rewrite into pyrex
- William Stein(2007-01): rewrite of Mohler's rewrite
```

Then `complex_number.pyx` was copied to `complex_interval.pyx` and heavily modified:

- Carl Witty (2007-10-24): rewrite to become a complex interval package
- Travis Scrimshaw (2012-10-18): Added documentation to get full coverage.

Warning

Mixing symbolic expressions with intervals (in particular, converting constant symbolic expressions to intervals), can lead to incorrect results:

```
sage: ref = ComplexIntervalField(100)(ComplexBallField(100).one().airy_ai())
sage: ref
0.135292416312881415524147423515?
sage: val = CIF(airy_ai(1)); val # known bug
0.13529241631288142?
sage: val.overlaps(ref) # known bug
False
```

```
>>> from sage.all import *
>>> ref = ComplexIntervalField(Integer(100))(ComplexBallField(Integer(100)).one().
↪airy_ai())
>>> ref
0.135292416312881415524147423515?
>>> val = CIF(airy_ai(Integer(1))); val # known bug
0.13529241631288142?
>>> val.overlaps(ref) # known bug
False
```

Todo

Implement `ComplexIntervalFieldElement` multiplicative order similar to `ComplexNumber` multiplicative order with `_set_multiplicative_order(n)` and `ComplexNumber.multiplicative_order()` methods.

```
class sage.rings.complex_interval.ComplexIntervalFieldElement
```

Bases: `FieldElement`

A complex interval.

EXAMPLES:


```
sage: I = CIF.gen()
sage: b = 3/2 + 5/2*I
sage: TestSuite(b).run()
```

```
>>> from sage.all import *
>>> I = CIF.gen()
>>> b = Integer(3)/Integer(2) + Integer(5)/Integer(2)*I
>>> TestSuite(b).run()
```

arg()

Same as `argument()`.

EXAMPLES:

```
sage: i = CIF.0
sage: (i^2).arg()
3.141592653589794?
```

```
>>> from sage.all import *
>>> i = CIF.gen(0)
>>> (i**Integer(2)).arg()
3.141592653589794?
```

argument()

The argument (angle) of the complex number, normalized so that $-\pi < \theta.lower() \leq \pi$.

We raise a `ValueError` if the interval strictly contains 0, or if the interval contains only 0.

Warning

We do not always use the standard branch cut for argument! If the interval crosses the negative real axis, then the argument will be an interval whose lower bound is less than π and whose upper bound is more than π ; in effect, we move the branch cut away from the interval.

EXAMPLES:

```
sage: i = CIF.0
sage: (i^2).argument()
3.141592653589794?
sage: (1+i).argument()
0.785398163397449?
sage: i.argument()
1.570796326794897?
sage: (-i).argument()
-1.570796326794897?
sage: (-1/1000 - i).argument()
-1.571796326461564?
sage: CIF(2).argument()
0
sage: CIF(-2).argument()
3.141592653589794?
```

```

>>> from sage.all import *
>>> i = CIF.gen(0)
>>> (i**Integer(2)).argument()
3.141592653589794?
>>> (Integer(1)+i).argument()
0.785398163397449?
>>> i.argument()
1.570796326794897?
>>> (-i).argument()
-1.570796326794897?
>>> (-Integer(1)/Integer(1000) - i).argument()
-1.571796326461564?
>>> CIF(Integer(2)).argument()
0
>>> CIF(-Integer(2)).argument()
3.141592653589794?

```

Here we see that if the interval crosses the negative real axis, then the argument can exceed π , and we violate the standard interval guarantees in the process:

```

sage: CIF(-2, RIF(-0.1, 0.1)).argument().str(style='brackets')
'[3.0916342578678501 .. 3.1915510493117365]'
sage: CIF(-2, -0.1).argument()
-3.091634257867851?

```

```

>>> from sage.all import *
>>> CIF(-Integer(2), RIF(-RealNumber('0.1'), RealNumber('0.1'))).argument().
    str(style='brackets')
'[3.0916342578678501 .. 3.1915510493117365]'
>>> CIF(-Integer(2), -RealNumber('0.1')).argument()
-3.091634257867851?

```

bisection()

Return the bisection of `self` into four intervals whose union is `self` and intersection is `center()`.

EXAMPLES:

```

sage: z = CIF(RIF(2, 3), RIF(-5, -4))
sage: z.bisection()
(3.? - 5.?*I, 3.? - 5.?*I, 3.? - 5.?*I, 3.? - 5.?*I)
sage: for z in z.bisection():
....:     print(z.real().endpoints())
....:     print(z.imag().endpoints())
(2.000000000000000, 2.500000000000000)
(-5.000000000000000, -4.500000000000000)
(2.500000000000000, 3.000000000000000)
(-5.000000000000000, -4.500000000000000)
(2.000000000000000, 2.500000000000000)
(-4.500000000000000, -4.000000000000000)
(2.500000000000000, 3.000000000000000)
(-4.500000000000000, -4.000000000000000)

sage: # needs sage.symbolic

```

(continues on next page)

(continued from previous page)

```

sage: z = CIF(RIF(sqrt(2), sqrt(3)), RIF(e, pi))
sage: a, b, c, d = z.bisection()
sage: a.intersection(b).intersection(c).intersection(d) == CIF(z.center())
True
sage: zz = a.union(b).union(c).union(c)
sage: zz.real().endpoints() == z.real().endpoints()
True
sage: zz.imag().endpoints() == z.imag().endpoints()
True

```

```

>>> from sage.all import *
>>> z = CIF(RIF(Integer(2), Integer(3)), RIF(-Integer(5), -Integer(4)))
>>> z.bisection()
(3.? - 5.?*I, 3.? - 5.?*I, 3.? - 5.?*I, 3.? - 5.?*I)
>>> for z in z.bisection():
...     print(z.real().endpoints())
...     print(z.imag().endpoints())
(2.000000000000000, 2.500000000000000)
(-5.000000000000000, -4.500000000000000)
(2.500000000000000, 3.000000000000000)
(-5.000000000000000, -4.500000000000000)
(2.000000000000000, 2.500000000000000)
(-4.500000000000000, -4.000000000000000)
(2.500000000000000, 3.000000000000000)
(-4.500000000000000, -4.000000000000000)

>>> # needs sage.symbolic
>>> z = CIF(RIF(sqrt(Integer(2)), sqrt(Integer(3))), RIF(e, pi))
>>> a, b, c, d = z.bisection()
>>> a.intersection(b).intersection(c).intersection(d) == CIF(z.center())
True
>>> zz = a.union(b).union(c).union(c)
>>> zz.real().endpoints() == z.real().endpoints()
True
>>> zz.imag().endpoints() == z.imag().endpoints()
True

```

center()

Return the closest floating-point approximation to the center of the interval.

EXAMPLES:

```

sage: CIF(RIF(1, 2), RIF(3, 4)).center()
1.500000000000000 + 3.500000000000000*I

```

```

>>> from sage.all import *
>>> CIF(RIF(Integer(1), Integer(2)), RIF(Integer(3), Integer(4))).center()
1.500000000000000 + 3.500000000000000*I

```

conjugate()

Return the complex conjugate of this complex number.

EXAMPLES:

```
sage: i = CIF.0
sage: (1+i).conjugate()
1 - 1*I
```

```
>>> from sage.all import *
>>> i = CIF.gen(0)
>>> (Integer(1)+i).conjugate()
1 - 1*I
```

contains_zero()

Return True if self is an interval containing zero.

EXAMPLES:

```
sage: CIF(0).contains_zero()
True
sage: CIF(RIF(-1, 1), 1).contains_zero()
False
```

```
>>> from sage.all import *
>>> CIF(Integer(0)).contains_zero()
True
>>> CIF(RIF(-Integer(1), Integer(1)), Integer(1)).contains_zero()
False
```

cos()

Compute the cosine of this complex interval.

EXAMPLES:

```
sage: CIF(1,1).cos()
0.833730025131149? - 0.988897705762865?*I
sage: CIF(3).cos()
-0.9899924966004455?
sage: CIF(0,2).cos()
3.762195691083632?
```

```
>>> from sage.all import *
>>> CIF(Integer(1),Integer(1)).cos()
0.833730025131149? - 0.988897705762865?*I
>>> CIF(Integer(3)).cos()
-0.9899924966004455?
>>> CIF(Integer(0),Integer(2)).cos()
3.762195691083632?
```

Check that [Issue #17285](#) is fixed:

```
sage: CIF(cos(2/3))
↪needs sage.symbolic
0.7858872607769480?
```

```
>>> from sage.all import *
>>> CIF(cos(Integer(2)/Integer(3)))
```

(continues on next page)

(continued from previous page)

```
↪ # needs sage.symbolic
0.7858872607769480?
```

ALGORITHM:

The implementation uses the following trigonometric identity

$$\cos(x + iy) = \cos(x) \cosh(y) - i \sin(x) \sinh(y)$$

cosh()

Return the hyperbolic cosine of this complex interval.

EXAMPLES:

```
sage: CIF(1,1).cosh()
0.833730025131149? + 0.988897705762865?*I
sage: CIF(2).cosh()
3.762195691083632?
sage: CIF(0,2).cosh()
-0.4161468365471424?
```

```
>>> from sage.all import *
>>> CIF(Integer(1), Integer(1)).cosh()
0.833730025131149? + 0.988897705762865?*I
>>> CIF(Integer(2)).cosh()
3.762195691083632?
>>> CIF(Integer(0), Integer(2)).cosh()
-0.4161468365471424?
```

ALGORITHM:

The implementation uses the following trigonometric identity

$$\cosh(x + iy) = \cos(y) \cosh(x) + i \sin(y) \sinh(x)$$

crosses_log_branch_cut()

Return True if this interval crosses the standard branch cut for `log()` (and hence for exponentiation) and for argument. (Recall that this branch cut is infinitesimally below the negative portion of the real axis.)

EXAMPLES:

```
sage: z = CIF(1.5, 2.5) - CIF(0, 2.5000000000000001); z
1.5000000000000000? + -1.?e-15*I
sage: z.crosses_log_branch_cut()
False
sage: CIF(-2, RIF(-0.1, 0.1)).crosses_log_branch_cut()
True
```

```
>>> from sage.all import *
>>> z = CIF(RealNumber('1.5'), RealNumber('2.5')) - CIF(Integer(0), ↪
↪ RealNumber('2.5000000000000001')); z
1.5000000000000000? + -1.?e-15*I
>>> z.crosses_log_branch_cut()
```

(continues on next page)

(continued from previous page)

```
False
>>> CIF(-Integer(2), RIF(-RealNumber('0.1'), RealNumber('0.1'))).crosses_log_
↳branch_cut()
True
```

diameter()

Return a somewhat-arbitrarily defined “diameter” for this interval.

The diameter of an interval is the maximum of the diameter of the real and imaginary components, where diameter on a real interval is defined as absolute diameter if the interval contains zero, and relative diameter otherwise.

EXAMPLES:

```
sage: CIF(RIF(-1, 1), RIF(13, 17)).diameter()
2.000000000000000
sage: CIF(RIF(-0.1, 0.1), RIF(13, 17)).diameter()
0.2666666666666667
sage: CIF(RIF(-1, 1), 15).diameter()
2.000000000000000
```

```
>>> from sage.all import *
>>> CIF(RIF(-Integer(1), Integer(1)), RIF(Integer(13), Integer(17))).
↳diameter()
2.000000000000000
>>> CIF(RIF(-RealNumber('0.1'), RealNumber('0.1')), RIF(Integer(13),
↳Integer(17))).diameter()
0.2666666666666667
>>> CIF(RIF(-Integer(1), Integer(1)), Integer(15)).diameter()
2.000000000000000
```

edges()

Return the 4 edges of the rectangle in the complex plane defined by this interval as intervals.

OUTPUT: a 4-tuple of complex intervals (left edge, right edge, lower edge, upper edge)

See also

endpoints() which returns the 4 corners of the rectangle.

EXAMPLES:

```
sage: CIF(RIF(1, 2), RIF(3, 4)).edges()
(1 + 4.?*I, 2 + 4.?*I, 2.? + 3*I, 2.? + 4*I)
sage: ComplexIntervalField(20)(-2).log().edges()
(0.69314671? + 3.14160?*I,
 0.69314766? + 3.14160?*I,
 0.693147? + 3.1415902?*I,
 0.693147? + 3.1415940?*I)
```

```
>>> from sage.all import *
>>> CIF(RIF(Integer(1), Integer(2)), RIF(Integer(3), Integer(4))).edges()
```

(continues on next page)

(continued from previous page)

```
(1 + 4.?*I, 2 + 4.?*I, 2.? + 3*I, 2.? + 4*I)
>>> ComplexIntervalField(Integer(20))(-Integer(2)).log().edges()
(0.69314671? + 3.14160?*I,
 0.69314766? + 3.14160?*I,
 0.693147? + 3.1415902?*I,
 0.693147? + 3.1415940?*I)
```

endpoints()

Return the 4 corners of the rectangle in the complex plane defined by this interval.

OUTPUT: a 4-tuple of complex numbers (lower left, upper right, upper left, lower right)

See also

[`edges\(\)`](#) which returns the 4 edges of the rectangle.

EXAMPLES:

```
sage: CIF(RIF(1,2), RIF(3,4)).endpoints()
(1.000000000000000 + 3.000000000000000*I,
 2.000000000000000 + 4.000000000000000*I,
 1.000000000000000 + 4.000000000000000*I,
 2.000000000000000 + 3.000000000000000*I)
sage: ComplexIntervalField(20)(-2).log().endpoints()
(0.69315 + 3.1416*I,
 0.69315 + 3.1416*I,
 0.69315 + 3.1416*I,
 0.69315 + 3.1416*I)
```

```
>>> from sage.all import *
>>> CIF(RIF(Integer(1),Integer(2)), RIF(Integer(3),Integer(4))).endpoints()
(1.000000000000000 + 3.000000000000000*I,
 2.000000000000000 + 4.000000000000000*I,
 1.000000000000000 + 4.000000000000000*I,
 2.000000000000000 + 3.000000000000000*I)
>>> ComplexIntervalField(Integer(20))(-Integer(2)).log().endpoints()
(0.69315 + 3.1416*I,
 0.69315 + 3.1416*I,
 0.69315 + 3.1416*I,
 0.69315 + 3.1416*I)
```

exp()

Compute e^z or $\exp(z)$ where z is the complex number `self`.

EXAMPLES:

```
sage: i = ComplexIntervalField(300).0
sage: z = 1 + i
sage: z.exp()
1.4686939399158851571389675973266042613269567366290087227976756763109369658595
↪1213872272450? + 2.287355287178842391208171906700501808955586256668355680938
↪65811410364716018934540926734485?*I
```

```

>>> from sage.all import *
>>> i = ComplexIntervalField(Integer(300)).gen(0)
>>> z = Integer(1) + i
>>> z.exp()
1.4686939399158851571389675973266042613269567366290087227976756763109369658595
↪1213872272450? + 2.287355287178842391208171906700501808955586256668355680938
↪65811410364716018934540926734485?*I

```

imag()

Return imaginary part of self.

EXAMPLES:

```

sage: i = ComplexIntervalField(100).0
sage: z = 2 + 3*i
sage: x = z.imag(); x
3
sage: x.parent()
Real Interval Field with 100 bits of precision

```

```

>>> from sage.all import *
>>> i = ComplexIntervalField(Integer(100)).gen(0)
>>> z = Integer(2) + Integer(3)*i
>>> x = z.imag(); x
3
>>> x.parent()
Real Interval Field with 100 bits of precision

```

intersection(*other*)

Return the intersection of the two complex intervals self and other.

EXAMPLES:

```

sage: CIF(RIF(1, 3), RIF(1, 3)).intersection(CIF(RIF(2, 4), RIF(2, 4))).
↪str(style='brackets')
'[2.0000000000000000 .. 3.0000000000000000] + [2.0000000000000000 .. 3.
↪0000000000000000]*I'
sage: CIF(RIF(1, 2), RIF(1, 3)).intersection(CIF(RIF(3, 4), RIF(2, 4)))
Traceback (most recent call last):
...
ValueError: intersection of non-overlapping intervals

```

```

>>> from sage.all import *
>>> CIF(RIF(Integer(1), Integer(3)), RIF(Integer(1), Integer(3))).
↪intersection(CIF(RIF(Integer(2), Integer(4)), RIF(Integer(2), Integer(4)))).
↪str(style='brackets')
'[2.0000000000000000 .. 3.0000000000000000] + [2.0000000000000000 .. 3.
↪0000000000000000]*I'
>>> CIF(RIF(Integer(1), Integer(2)), RIF(Integer(1), Integer(3))).
↪intersection(CIF(RIF(Integer(3), Integer(4)), RIF(Integer(2), Integer(4)))).
Traceback (most recent call last):
...
ValueError: intersection of non-overlapping intervals

```


is_NaN()

Return True if this is not-a-number.

EXAMPLES:

```
sage: CIF(2, 1).is_NaN()
False
sage: CIF(NaN).is_NaN()
↪needs sage.symbolic
True
sage: (1 / CIF(0, 0)).is_NaN()
True
```

```
>>> from sage.all import *
>>> CIF(Integer(2), Integer(1)).is_NaN()
False
>>> CIF(NaN).is_NaN()
↪needs sage.symbolic
True
>>> (Integer(1) / CIF(Integer(0), Integer(0))).is_NaN()
True
```

is_exact()

Return whether this complex interval is exact (i.e. contains exactly one complex value).

EXAMPLES:

```
sage: CIF(3).is_exact()
True
sage: CIF(0, 2).is_exact()
True
sage: CIF(-4, 0).sqrt().is_exact()
True
sage: CIF(-5, 0).sqrt().is_exact()
False
sage: CIF(0, 2*pi).is_exact()
↪needs sage.symbolic
False
sage: CIF(e).is_exact()
↪needs sage.symbolic
False
sage: CIF(1e100).is_exact()
True
sage: (CIF(1e100) + 1).is_exact()
False
```

```
>>> from sage.all import *
>>> CIF(Integer(3)).is_exact()
True
>>> CIF(Integer(0), Integer(2)).is_exact()
True
>>> CIF(-Integer(4), Integer(0)).sqrt().is_exact()
True
>>> CIF(-Integer(5), Integer(0)).sqrt().is_exact()
```

(continues on next page)

(continued from previous page)

```
False
>>> CIF(Integer(0), Integer(2)*pi).is_exact()
↪ # needs sage.symbolic
False
>>> CIF(e).is_exact()
↪ needs sage.symbolic
False
>>> CIF(RealNumber('1e100')).is_exact()
True
>>> (CIF(RealNumber('1e100')) + Integer(1)).is_exact()
False
```

is_square()

Return True as $\mathbf{C}$ is algebraically closed.

EXAMPLES:

```
sage: CIF(2, 1).is_square()
True
```

```
>>> from sage.all import *
>>> CIF(Integer(2), Integer(1)).is_square()
True
```

lexico_cmp (*left, right*)

Intervals are compared lexicographically on the 4-tuple: $(x.\text{real}().\text{lower}(), x.\text{real}().\text{upper}(), x.\text{imag}().\text{lower}(), x.\text{imag}().\text{upper}())$

EXAMPLES:

```
sage: a = CIF(RIF(0,1), RIF(0,1))
sage: b = CIF(RIF(0,1), RIF(0,2))
sage: c = CIF(RIF(0,2), RIF(0,2))
sage: a.lexico_cmp(b)
-1
sage: b.lexico_cmp(c)
-1
sage: a.lexico_cmp(c)
-1
sage: a.lexico_cmp(a)
0
sage: b.lexico_cmp(a)
1
```

```
>>> from sage.all import *
>>> a = CIF(RIF(Integer(0), Integer(1)), RIF(Integer(0), Integer(1)))
>>> b = CIF(RIF(Integer(0), Integer(1)), RIF(Integer(0), Integer(2)))
>>> c = CIF(RIF(Integer(0), Integer(2)), RIF(Integer(0), Integer(2)))
>>> a.lexico_cmp(b)
-1
>>> b.lexico_cmp(c)
-1
```

(continues on next page)

(continued from previous page)

```
>>> a.lexico_cmp(c)
-1
>>> a.lexico_cmp(a)
0
>>> b.lexico_cmp(a)
1
```

log (*base=None*)Complex logarithm of z .**Warning**

This does always not use the standard branch cut for complex log! See the docstring for `argument()` to see what we do instead.

EXAMPLES:

```
sage: a = CIF(RIF(3, 4), RIF(13, 14))
sage: a.log().str(style='brackets')
'[2.5908917751460420 .. 2.6782931373360067] + [1.2722973952087170 .. 1.
↪3597029935721503]*I'
sage: a.log().exp().str(style='brackets')
'[2.7954667135098274 .. 4.2819545928390213] + [12.751682453911920 .. 14.
↪237018048974635]*I'
sage: a in a.log().exp()
True
```

```
>>> from sage.all import *
>>> a = CIF(RIF(Integer(3), Integer(4)), RIF(Integer(13), Integer(14)))
>>> a.log().str(style='brackets')
'[2.5908917751460420 .. 2.6782931373360067] + [1.2722973952087170 .. 1.
↪3597029935721503]*I'
>>> a.log().exp().str(style='brackets')
'[2.7954667135098274 .. 4.2819545928390213] + [12.751682453911920 .. 14.
↪237018048974635]*I'
>>> a in a.log().exp()
True
```

If the interval crosses the negative real axis, then we don't use the standard branch cut (and we violate the interval guarantees):

```
sage: CIF(-3, RIF(-1/4, 1/4)).log().str(style='brackets')
'[1.0986122886681095 .. 1.1020725100903968] + [3.0584514217013518 .. 3.
↪2247338854782349]*I'
sage: CIF(-3, -1/4).log()
1.102072510090397? - 3.058451421701352?*I
```

```
>>> from sage.all import *
>>> CIF(-Integer(3), RIF(-Integer(1)/Integer(4), Integer(1)/Integer(4))).
↪log().str(style='brackets')
'[1.0986122886681095 .. 1.1020725100903968] + [3.0584514217013518 .. 3.
```

(continues on next page)

(continued from previous page)

```
↪2247338854782349]*I'
>>> CIF(-Integer(3), -Integer(1)/Integer(4)).log()
1.102072510090397? - 3.058451421701352?*I
```

Usually if an interval contains zero, we raise an exception:

```
sage: CIF(RIF(-1,1), RIF(-1,1)).log()
Traceback (most recent call last):
...
ValueError: Can...t take the argument of interval strictly containing zero
```

```
>>> from sage.all import *
>>> CIF(RIF(-Integer(1), Integer(1)), RIF(-Integer(1), Integer(1))).log()
Traceback (most recent call last):
...
ValueError: Can...t take the argument of interval strictly containing zero
```

But we allow the exact input zero:

```
sage: CIF(0).log()
[-infinity .. -infinity]
```

```
>>> from sage.all import *
>>> CIF(Integer(0)).log()
[-infinity .. -infinity]
```

If a base is passed from another function, we can accommodate this:

```
sage: CIF(-1,1).log(2)
0.5000000000000000? + 3.39927010637040?*I
```

```
>>> from sage.all import *
>>> CIF(-Integer(1), Integer(1)).log(Integer(2))
0.5000000000000000? + 3.39927010637040?*I
```

magnitude()

The largest absolute value of the elements of the interval, rounded away from zero.

OUTPUT: a real number with rounding mode RNDU

EXAMPLES:

```
sage: CIF(RIF(-1,1), RIF(-1,1)).magnitude()
1.41421356237310
sage: CIF(RIF(1,2), RIF(3,4)).magnitude()
4.47213595499958
sage: parent(CIF(1).magnitude())
Real Field with 53 bits of precision and rounding RNDU
```

```
>>> from sage.all import *
>>> CIF(RIF(-Integer(1), Integer(1)), RIF(-Integer(1), Integer(1))).magnitude()
1.41421356237310
```

(continues on next page)

(continued from previous page)

```
>>> CIF(RIF(Integer(1), Integer(2)), RIF(Integer(3), Integer(4))).magnitude()
4.47213595499958
>>> parent(CIF(Integer(1)).magnitude())
Real Field with 53 bits of precision and rounding RNDD
```

mignitude()

The smallest absolute value of the elements of the interval, rounded towards zero.

OUTPUT: a real number with rounding mode RNDD

EXAMPLES:

```
sage: CIF(RIF(-1, 1), RIF(-1, 1)).mignitude()
0.0000000000000000
sage: CIF(RIF(1, 2), RIF(3, 4)).mignitude()
3.16227766016837
sage: parent(CIF(1).mignitude())
Real Field with 53 bits of precision and rounding RNDD
```

```
>>> from sage.all import *
>>> CIF(RIF(-Integer(1), Integer(1)), RIF(-Integer(1), Integer(1))).mignitude()
0.0000000000000000
>>> CIF(RIF(Integer(1), Integer(2)), RIF(Integer(3), Integer(4))).mignitude()
3.16227766016837
>>> parent(CIF(Integer(1)).mignitude())
Real Field with 53 bits of precision and rounding RNDD
```

multiplicative_order()

Return the multiplicative order of this complex number, if known, or raise a `NotImplementedError`.

EXAMPLES:

```
sage: C = CIF
sage: i = C.0
sage: i.multiplicative_order()
4
sage: C(1).multiplicative_order()
1
sage: C(-1).multiplicative_order()
2
sage: (i^2).multiplicative_order()
2
sage: (-i).multiplicative_order()
4
sage: C(2).multiplicative_order()
+Infinity
sage: w = (1 + C(-3).sqrt())/2 ; w
0.5000000000000000? + 0.866025403784439?*I
sage: w.multiplicative_order()
Traceback (most recent call last):
...
NotImplementedError: order of element not known
```

```

>>> from sage.all import *
>>> C = CIF
>>> i = C.gen(0)
>>> i.multiplicative_order()
4
>>> C(Integer(1)).multiplicative_order()
1
>>> C(-Integer(1)).multiplicative_order()
2
>>> (i**Integer(2)).multiplicative_order()
2
>>> (-i).multiplicative_order()
4
>>> C(Integer(2)).multiplicative_order()
+Infinity
>>> w = (Integer(1) + C(-Integer(3)).sqrt())/Integer(2) ; w
0.5000000000000000? + 0.866025403784439?*I
>>> w.multiplicative_order()
Traceback (most recent call last):
...
NotImplementedError: order of element not known

```

norm()

Return the norm of this complex number.

If $c = a + bi$ is a complex number, then the norm of c is defined as the product of c and its complex conjugate:

$$\text{norm}(c) = \text{norm}(a + bi) = c \cdot \bar{c} = a^2 + b^2.$$

The norm of a complex number is different from its absolute value. The absolute value of a complex number is defined to be the square root of its norm. A typical use of the complex norm is in the integral domain $\mathbf{Z}[i]$ of Gaussian integers, where the norm of each Gaussian integer $c = a + bi$ is defined as its complex norm.

See also

- `sage.rings.complex_double.ComplexDoubleElement.norm()`

EXAMPLES:

```

sage: CIF(2, 1).norm()
5
sage: CIF(1, -2).norm()
5

```

```

>>> from sage.all import *
>>> CIF(Integer(2), Integer(1)).norm()
5
>>> CIF(Integer(1), -Integer(2)).norm()
5

```

overlaps(*other*)

Return True if *self* and *other* are intervals with at least one value in common.

EXAMPLES:

```
sage: CIF(0).overlaps(CIF(RIF(0, 1), RIF(-1, 0)))
True
sage: CIF(1).overlaps(CIF(1, 1))
False
```

```
>>> from sage.all import *
>>> CIF(Integer(0)).overlaps(CIF(RIF(Integer(0), Integer(1)), RIF(-Integer(1),
↪ Integer(0))))
True
>>> CIF(Integer(1)).overlaps(CIF(Integer(1), Integer(1)))
False
```

`plot (pointsize=10, **kws)`

Plot a complex interval as a rectangle.

EXAMPLES:

```
sage: sum(plot(CIF(RIF(1/k, 1/k), RIF(-k, k))) for k in [1..10]) #↵
↪needs sage.plot
Graphics object consisting of 20 graphics primitives
```

```
>>> from sage.all import *
>>> sum(plot(CIF(RIF(Integer(1)/k, Integer(1)/k), RIF(-k, k))) for k in↵
↪(ellipsis_range(Integer(1), Ellipsis, Integer(10)))) # needs sage.
↪plot
Graphics object consisting of 20 graphics primitives
```

Exact and nearly exact points are still visible:

```
sage: # needs sage.plot sage.symbolic
sage: plot(CIF(pi, 1), color='red') + plot(CIF(1, e), color='purple') +↵
↪plot(CIF(-1, -1))
Graphics object consisting of 6 graphics primitives
```

```
>>> from sage.all import *
>>> # needs sage.plot sage.symbolic
>>> plot(CIF(pi, Integer(1)), color='red') + plot(CIF(Integer(1), e), color=↵
↪'purple') + plot(CIF(-Integer(1), -Integer(1)))
Graphics object consisting of 6 graphics primitives
```

A demonstration that $z \mapsto z^2$ acts chaotically on $|z| = 1$:

```
sage: # needs sage.plot sage.symbolic
sage: z = CIF(0, 2*pi/1000).exp()
sage: g = Graphics()
sage: for i in range(40):
....:     z = z^2
....:     g += z.plot(color=(1./(40-i), 0, 1))
...
sage: g
Graphics object consisting of 80 graphics primitives
```

```

>>> from sage.all import *
>>> # needs sage.plot sage.symbolic
>>> z = CIF(Integer(0), Integer(2)*pi/Integer(1000)).exp()
>>> g = Graphics()
>>> for i in range(Integer(40)):
...     z = z**Integer(2)
...     g += z.plot(color=(RealNumber('1.')/(Integer(40)-i), Integer(0),
↳Integer(1)))
...
>>> g
Graphics object consisting of 80 graphics primitives

```

prec()

Return precision of this complex number.

EXAMPLES:

```

sage: i = ComplexIntervalField(2000).0
sage: i.prec()
2000

```

```

>>> from sage.all import *
>>> i = ComplexIntervalField(Integer(2000)).gen(0)
>>> i.prec()
2000

```

real()

Return real part of self.

EXAMPLES:

```

sage: i = ComplexIntervalField(100).0
sage: z = 2 + 3*i
sage: x = z.real(); x
2
sage: x.parent()
Real Interval Field with 100 bits of precision

```

```

>>> from sage.all import *
>>> i = ComplexIntervalField(Integer(100)).gen(0)
>>> z = Integer(2) + Integer(3)*i
>>> x = z.real(); x
2
>>> x.parent()
Real Interval Field with 100 bits of precision

```

sin()

Compute the sine of this complex interval.

EXAMPLES:

```

sage: CIF(1,1).sin()
1.298457581415978? + 0.634963914784736?*I

```

(continues on next page)

(continued from previous page)

```
sage: CIF(2).sin()
0.909297426825682?
sage: CIF(0,2).sin()
3.626860407847019?*I
```

```
>>> from sage.all import *
>>> CIF(Integer(1), Integer(1)).sin()
1.298457581415978? + 0.634963914784736?*I
>>> CIF(Integer(2)).sin()
0.909297426825682?
>>> CIF(Integer(0), Integer(2)).sin()
3.626860407847019?*I
```

Check that [Issue #17825](#) is fixed:

```
sage: CIF(sin(2/3))
↪needs sage.symbolic
0.618369803069737? #_
```

```
>>> from sage.all import *
>>> CIF(sin(Integer(2)/Integer(3)))
↪ # needs sage.symbolic
0.618369803069737? _
```

ALGORITHM:

The implementation uses the following trigonometric identity

$$\sin(x + iy) = \sin(x) \cosh(y) + i \cos(x) \sinh(y)$$

sinh()

Return the hyperbolic sine of this complex interval.

EXAMPLES:

```
sage: CIF(1,1).sinh()
0.634963914784736? + 1.298457581415978?*I
sage: CIF(2).sinh()
3.626860407847019?
sage: CIF(0,2).sinh()
0.909297426825682?*I
```

```
>>> from sage.all import *
>>> CIF(Integer(1), Integer(1)).sinh()
0.634963914784736? + 1.298457581415978?*I
>>> CIF(Integer(2)).sinh()
3.626860407847019?
>>> CIF(Integer(0), Integer(2)).sinh()
0.909297426825682?*I
```

ALGORITHM:

The implementation uses the following trigonometric identity

$$\sinh(x + iy) = \cosh(y) \sinh(x) + i \sin(y) \cosh(x)$$

`sqrt` (*all=False*, ***kws*)

The square root function.

Warning

We approximate the standard branch cut along the negative real axis, with $\text{sqrt}(-r^2) = i*r$ for positive real r ; but if the interval crosses the negative real axis, we pick the root with positive imaginary component for the entire interval.

INPUT:

- *all* – boolean (default: `False`); if `True`, return a list of all square roots

EXAMPLES:

```
sage: CIF(-1).sqrt()^2
-1
sage: sqrt(CIF(2))
1.414213562373095?
sage: sqrt(CIF(-1))
1*I
sage: sqrt(CIF(2-I))^2
2.000000000000000? - 1.000000000000000*I
sage: CC(-2-I).sqrt()^2
-2.000000000000000 - 1.000000000000000*I
```

```
>>> from sage.all import *
>>> CIF(-Integer(1)).sqrt()**Integer(2)
-1
>>> sqrt(CIF(Integer(2)))
1.414213562373095?
>>> sqrt(CIF(-Integer(1)))
1*I
>>> sqrt(CIF(Integer(2)-I))**Integer(2)
2.000000000000000? - 1.000000000000000*I
>>> CC(-Integer(2)-I).sqrt()**Integer(2)
-2.000000000000000 - 1.000000000000000*I
```

Here, we select a non-principal root for part of the interval, and violate the standard interval guarantees:

```
sage: CIF(-5, RIF(-1, 1)).sqrt().str(style='brackets')
'[-0.22250788030178321 .. 0.22250788030178296] + [2.2251857651053086 .. 2.
↪2581008643532262]*I'
sage: CIF(-5, -1).sqrt()
0.222507880301783? - 2.247111425095870?I
```

```
>>> from sage.all import *
>>> CIF(-Integer(5), RIF(-Integer(1), Integer(1))).sqrt().str(style='brackets')
↪')
'[-0.22250788030178321 .. 0.22250788030178296] + [2.2251857651053086 .. 2.
↪2581008643532262]*I'
>>> CIF(-Integer(5), -Integer(1)).sqrt()
0.222507880301783? - 2.247111425095870?I
```

```
str (base=10, style=None)
```

Return a string representation of `self`.

EXAMPLES:

```
sage: CIF(1.5).str()
'1.500000000000000?'
sage: CIF(1.5, 2.5).str()
'1.500000000000000? + 2.500000000000000?*I'
sage: CIF(1.5, -2.5).str()
'1.500000000000000? - 2.500000000000000?*I'
sage: CIF(0, -2.5).str()
'-2.500000000000000?*I'
sage: CIF(1.5).str(base=3)
'1.111111111111111111111111111111111111111111111112?'
sage: CIF(1, pi).str(style='brackets') #_
```

←needs sage.symbolic

```
'[1.000000000000000 .. 1.000000000000000] + [3.1415926535897931 .. 3.
```

*←1415926535897936]*I'*

```
>>> from sage.all import *  
>>> CIF(RealNumber('1.5')).str()  
'1.500000000000000?'  
>>> CIF(RealNumber('1.5'), RealNumber('2.5')).str()  
'1.500000000000000? + 2.500000000000000?*I'  
>>> CIF(RealNumber('1.5'), -RealNumber('2.5')).str()  
'1.500000000000000? - 2.500000000000000?*I'  
>>> CIF(Integer(0), -RealNumber('2.5')).str()  
'-2.500000000000000?*I'  
>>> CIF(RealNumber('1.5')).str(base=Integer(3))  
'1.111111111111111111111111111111111111111111111112?'  
>>> CIF(Integer(1), pi).str(style='brackets')  
↪ # needs sage.symbolic  
'[1.000000000000000 .. 1.000000000000000] + [3.1415926535897931 .. 3.  
↪ 1415926535897936]*I'
```

See also

- `RealIntervalFieldElement.str()`

`tan()`

Return the tangent of this complex interval.

EXAMPLES:

```
sage: CIF(1,1).tan()
0.27175258531952? + 1.08392332733870?*I
sage: CIF(2).tan()
-2.185039863261519?
sage: CIF(0,2).tan()
0.964027580075817?*I
```

```
>>> from sage.all import *
>>> CIF(Integer(1), Integer(1)).tan()
0.27175258531952? + 1.08392332733870?*I
>>> CIF(Integer(2)).tan()
-2.185039863261519?
>>> CIF(Integer(0), Integer(2)).tan()
0.964027580075817?*I
```

tanh()

Return the hyperbolic tangent of this complex interval.

EXAMPLES:

```
sage: CIF(1, 1).tanh()
1.08392332733870? + 0.27175258531952?*I
sage: CIF(2).tanh()
0.964027580075817?
sage: CIF(0, 2).tanh()
-2.185039863261519?*I
```

```
>>> from sage.all import *
>>> CIF(Integer(1), Integer(1)).tanh()
1.08392332733870? + 0.27175258531952?*I
>>> CIF(Integer(2)).tanh()
0.964027580075817?
>>> CIF(Integer(0), Integer(2)).tanh()
-2.185039863261519?*I
```

union(*other*)

Return the smallest complex interval including the two complex intervals *self* and *other*.

EXAMPLES:

```
sage: CIF(0).union(CIF(5, 5)).str(style='brackets')
'[0.0000000000000000 .. 5.0000000000000000] + [0.0000000000000000 .. 5.
↪0000000000000000]*I'
```

```
>>> from sage.all import *
>>> CIF(Integer(0)).union(CIF(Integer(5), Integer(5))).str(style='brackets')
'[0.0000000000000000 .. 5.0000000000000000] + [0.0000000000000000 .. 5.
↪0000000000000000]*I'
```

zeta(*a=None*)

Return the image of this interval by the Hurwitz zeta function.

For $a = 1$ (or $a = \text{None}$), this computes the Riemann zeta function.

EXAMPLES:

```
sage: zeta(CIF(2, 3))
0.7980219851462757? - 0.1137443080529385?*I
sage: _.parent()
Complex Interval Field with 53 bits of precision
sage: CIF(2, 3).zeta(1/2)
-1.955171567161496? + 3.123301509220897?*I
```

[illegible]

INPUT:

- `s_real` – string that defines a real number (or something whose string representation defines a number)
- `s_imag` – string that defines a real number (or something whose string representation defines a number)
- `pad` – integer at least 0
- `min_prec` – number will have at least this many bits of precision, no matter what

```
sage: ComplexIntervalFieldElement('2.3')  
2.300000000000000?  
  
sage: ComplexIntervalFieldElement('2.3','1.1')  
2.300000000000000? + 1.100000000000000?*I  
  
sage: ComplexIntervalFieldElement(10)  
10  
  
sage: ComplexIntervalFieldElement(10,10)  
10 + 10*I  
  
sage: ComplexIntervalFieldElement(1.0000000000000000000000000000,2)  
1 + 2*I  
  
sage: ComplexIntervalFieldElement(1,2.0000000000000000000000000000)  
1 + 2*I  
  
sage: ComplexIntervalFieldElement(1.234567890123456789012345, 5.  
↪4321098654321987654321)  
1.234567890123456789012350? + 5.432109865432198765432000?*I
```

(continues on next page)

(continued from previous page)

```
>>> ComplexIntervalFieldElement(RealNumber('1.234567890123456789012345'),
↳RealNumber('5.4321098654321987654321'))
1.234567890123456789012350? + 5.432109865432198765432000?*I
```

`sage.rings.complex_interval.is_ComplexIntervalFieldElement(x)`

Check if `x` is a *ComplexIntervalFieldElement*.

EXAMPLES:

```
sage: from sage.rings.complex_interval import is_ComplexIntervalFieldElement as
↳is_CIFE
sage: is_CIFE(CIF(2))
doctest:warning...
DeprecationWarning: The function is_ComplexIntervalFieldElement is deprecated;
use 'isinstance(..., ComplexIntervalFieldElement)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
sage: is_CIFE(CC(2))
False
```

```
>>> from sage.all import *
>>> from sage.rings.complex_interval import is_ComplexIntervalFieldElement as is_
↳CIFE
>>> is_CIFE(CIF(Integer(2)))
doctest:warning...
DeprecationWarning: The function is_ComplexIntervalFieldElement is deprecated;
use 'isinstance(..., ComplexIntervalFieldElement)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
True
>>> is_CIFE(CC(Integer(2)))
False
```

`sage.rings.complex_interval.make_ComplexIntervalFieldElement0(fld, re, im)`

Construct a *ComplexIntervalFieldElement* for pickling.

2.5 Arbitrary precision real balls

This is a binding to the `arb` module of `FLINT`. It may be useful to refer to its documentation for more details.

Parts of the documentation for this module are copied or adapted from Arb's (now FLINT's) own documentation, licenced (at the time) under the GNU General Public License version 2, or later.

See also

- *Complex balls*
- *Real intervals using MPFI*

2.5.1 Data Structure

Ball arithmetic, also known as mid-rad interval arithmetic, is an extension of floating-point arithmetic in which an error bound is attached to each variable. This allows doing rigorous computations over the real numbers, while avoiding the overhead of traditional (inf-sup) interval arithmetic at high precision, and eliminating much of the need for time-consuming and bug-prone manual error analysis associated with standard floating-point arithmetic.

Sage *RealBall* objects wrap FLINT objects of type `arb_t`. A real ball represents a ball over the real numbers, that is, an interval $[m - r, m + r]$ where the midpoint m and the radius r are (extended) real numbers:

```
sage: RBF(pi)
↪needs sage.symbolic
[3.141592653589793 +/- ...e-16]
sage: RBF(pi).mid(), RBF(pi).rad()
↪needs sage.symbolic
(3.14159265358979, ...e-16)
```

```
>>> from sage.all import *
>>> RBF(pi)
↪needs sage.symbolic
[3.141592653589793 +/- ...e-16]
>>> RBF(pi).mid(), RBF(pi).rad()
↪needs sage.symbolic
(3.14159265358979, ...e-16)
```

The midpoint is represented as an arbitrary-precision floating-point number with arbitrary-precision exponent. The radius is a floating-point number with fixed-precision mantissa and arbitrary-precision exponent.

```
sage: RBF(2)^(2^100)
[2.285367694229514e+381600854690147056244358827360 +/- ...
↪e+381600854690147056244358827344]
```

```
>>> from sage.all import *
>>> RBF(Integer(2)**(Integer(2)**Integer(100)))
[2.285367694229514e+381600854690147056244358827360 +/- ...
↪e+381600854690147056244358827344]
```

RealBallField objects (the parents of real balls) model the field of real numbers represented by balls on which computations are carried out with a certain precision:

```
sage: RBF
Real ball field with 53 bits of precision
```

```
>>> from sage.all import *
>>> RBF
Real ball field with 53 bits of precision
```

It is possible to construct a ball whose parent is the real ball field with precision p but whose midpoint does not fit on p bits. However, the results of operations involving such a ball will (usually) be rounded to its parent's precision:

```
sage: RBF(factorial(50)).mid(), RBF(factorial(50)).rad()
(3.0414093201713378043612608166064768844377641568961e64, 0.00000000)
sage: (RBF(factorial(50)) + 0).mid()
3.04140932017134e64
```

```
>>> from sage.all import *
>>> RBF(factorial(Integer(50))).mid(), RBF(factorial(Integer(50))).rad()
(3.0414093201713378043612608166064768844377641568961e64, 0.00000000)
>>> (RBF(factorial(Integer(50))) + Integer(0)).mid()
3.04140932017134e64
```

2.5.2 Comparison

Warning

In accordance with the semantics of FLINT/Arb, identical *RealBall* objects are understood to give permission for algebraic simplification. This assumption is made to improve performance. For example, setting $z = x*x$ may set z to a ball enclosing the set $\{t^2 : t \in x\}$ and not the (generally larger) set $\{tu : t \in x, u \in x\}$.

Two elements are equal if and only if they are exact and equal (in spite of the above warning, inexact balls are not considered equal to themselves):

```
sage: a = RBF(1)
sage: b = RBF(1)
sage: a is b
False
sage: a == a
True
sage: a == b
True
```

```
>>> from sage.all import *
>>> a = RBF(Integer(1))
>>> b = RBF(Integer(1))
>>> a is b
False
>>> a == a
True
>>> a == b
True
```

```
sage: a = RBF(1/3)
sage: b = RBF(1/3)
sage: a.is_exact()
False
sage: b.is_exact()
False
sage: a is b
False
sage: a == a
False
sage: a == b
False
```



```

>>> from sage.all import *
>>> a = RBF(Integer(1)/Integer(3))
>>> b = RBF(Integer(1)/Integer(3))
>>> a.is_exact()
False
>>> b.is_exact()
False
>>> a is b
False
>>> a == a
False
>>> a == b
False

```

A ball is nonzero in the sense of comparison if and only if it does not contain zero.

```

sage: a = RBF(RIF(-0.5, 0.5))
sage: a != 0
False
sage: b = RBF(1/3)
sage: b != 0
True

```

```

>>> from sage.all import *
>>> a = RBF(RIF(-RealNumber('0.5'), RealNumber('0.5')))
>>> a != Integer(0)
False
>>> b = RBF(Integer(1)/Integer(3))
>>> b != Integer(0)
True

```

However, `bool(b)` returns `False` for a ball `b` only if `b` is exactly zero:

```

sage: bool(a)
True
sage: bool(b)
True
sage: bool(RBF.zero())
False

```

```

>>> from sage.all import *
>>> bool(a)
True
>>> bool(b)
True
>>> bool(RBF.zero())
False

```

A ball `left` is less than a ball `right` if all elements of `left` are less than all elements of `right`.

```

sage: a = RBF(RIF(1, 2))
sage: b = RBF(RIF(3, 4))
sage: a < b

```

(continues on next page)

(continued from previous page)

```

True
sage: a <= b
True
sage: a > b
False
sage: a >= b
False
sage: a = RBF(RIF(1, 3))
sage: b = RBF(RIF(2, 4))
sage: a < b
False
sage: a <= b
False
sage: a > b
False
sage: a >= b
False

```

```

>>> from sage.all import *
>>> a = RBF(RIF(Integer(1), Integer(2)))
>>> b = RBF(RIF(Integer(3), Integer(4)))
>>> a < b
True
>>> a <= b
True
>>> a > b
False
>>> a >= b
False
>>> a = RBF(RIF(Integer(1), Integer(3)))
>>> b = RBF(RIF(Integer(2), Integer(4)))
>>> a < b
False
>>> a <= b
False
>>> a > b
False
>>> a >= b
False

```

Comparisons with Sage symbolic infinities work with some limitations:

```

sage: -infinity < RBF(1) < +infinity
True
sage: -infinity < RBF(infinity)
True
sage: RBF(infinity) < infinity
False
sage: RBF(NaN) < infinity
↳needs sage.symbolic
Traceback (most recent call last):
...

```

#

(continues on next page)

(continued from previous page)

```

ValueError: infinite but not with +/- phase
sage: 1/RBF(0) <= infinity
Traceback (most recent call last):
...
ValueError: infinite but not with +/- phase

```

```

>>> from sage.all import *
>>> -infinity < RBF(Integer(1)) < +infinity
True
>>> -infinity < RBF(infinity)
True
>>> RBF(infinity) < infinity
False
>>> RBF(NaN) < infinity
↳needs sage.symbolic
Traceback (most recent call last):
...
ValueError: infinite but not with +/- phase
>>> Integer(1)/RBF(Integer(0)) <= infinity
Traceback (most recent call last):
...
ValueError: infinite but not with +/- phase

```

Comparisons between elements of real ball fields, however, support special values and should be preferred:

```

sage: RBF(NaN) < RBF(infinity)
↳needs sage.symbolic
False
sage: RBF(0).add_error(infinity) <= RBF(infinity)
True

```

```

>>> from sage.all import *
>>> RBF(NaN) < RBF(infinity)
↳needs sage.symbolic
False
>>> RBF(Integer(0)).add_error(infinity) <= RBF(infinity)
True

```

2.5.3 Classes and Methods

class sage.rings.real_arb.RealBall

Bases: RingElement

Hold one arb_t.

EXAMPLES:

```

sage: a = RealBallField()(RIF(1))
sage: b = a.psi()
sage: b # abs tol 1e-15
[-0.5772156649015329 +/- 4.84e-17]

```

(continues on next page)

(continued from previous page)

```
sage: RIF(b)
-0.577215664901533?
```

```
>>> from sage.all import *
>>> a = RealBallField()(RIF(Integer(1))) # indirect doctest
>>> b = a.psi()
>>> b # abs tol 1e-15
[-0.5772156649015329 +/- 4.84e-17]
>>> RIF(b)
-0.577215664901533?
```

Chi()

Hyperbolic cosine integral.

EXAMPLES:

```
sage: RBF(1).Chi() # abs tol 5e-16
[0.837866940980208 +/- 4.72e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).Chi() # abs tol 5e-16
[0.837866940980208 +/- 4.72e-16]
```

Ci()

Cosine integral.

EXAMPLES:

```
sage: RBF(1).Ci() # abs tol 5e-16
[0.337403922900968 +/- 3.25e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).Ci() # abs tol 5e-16
[0.337403922900968 +/- 3.25e-16]
```

Ei()

Exponential integral.

EXAMPLES:

```
sage: RBF(1).Ei() # abs tol 5e-15
[1.89511781635594 +/- 4.94e-15]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).Ei() # abs tol 5e-15
[1.89511781635594 +/- 4.94e-15]
```

Li()

Offset logarithmic integral.

EXAMPLES:

```
sage: RBF(3).Li() # abs tol 5e-15
[1.11842481454970 +/- 7.61e-15]
```

```
>>> from sage.all import *
>>> RBF(Integer(3)).Li() # abs tol 5e-15
[1.11842481454970 +/- 7.61e-15]
```

Shi()

Hyperbolic sine integral.

EXAMPLES:

```
sage: RBF(1).Shi() # abs tol 5e-15
[1.05725087537573 +/- 2.77e-15]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).Shi() # abs tol 5e-15
[1.05725087537573 +/- 2.77e-15]
```

Si()

Sine integral.

EXAMPLES:

```
sage: RBF(1).Si() # abs tol 1e-15
[0.946083070367183 +/- 9.22e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).Si() # abs tol 1e-15
[0.946083070367183 +/- 9.22e-16]
```

above_abs()

Return an upper bound for the absolute value of this ball.

OUTPUT: a ball with zero radius

EXAMPLES:

```
sage: b = RealBallField(8)(1/3).above_abs()
sage: b
[0.33 +/- ...e-3]
sage: b.is_exact()
True
sage: QQ(b)
171/512
```

```
>>> from sage.all import *
>>> b = RealBallField(Integer(8))(Integer(1)/Integer(3)).above_abs()
>>> b
[0.33 +/- ...e-3]
>>> b.is_exact()
True
>>> QQ(b)
171/512
```

See also`below_abs()`**accuracy()**

Return the effective relative accuracy of this ball measured in bits.

The accuracy is defined as the difference between the position of the top bit in the midpoint and the top bit in the radius, minus one. The result is clamped between plus/minus `maximal_accuracy()`.

EXAMPLES:

```
sage: RBF(pi).accuracy() #_
↪needs sage.symbolic
52
sage: RBF(1).accuracy() == RBF.maximal_accuracy()
True
sage: RBF(NaN).accuracy() == -RBF.maximal_accuracy() #_
↪needs sage.symbolic
True
```

```
>>> from sage.all import *
>>> RBF(pi).accuracy() #_
↪needs sage.symbolic
52
>>> RBF(Integer(1)).accuracy() == RBF.maximal_accuracy()
True
>>> RBF(NaN).accuracy() == -RBF.maximal_accuracy() #_
↪needs sage.symbolic
True
```

See also`maximal_accuracy()`**add_error(amp1)**

Increase the radius of this ball by (an upper bound on) `amp1`.

If `amp1` is negative, the radius is unchanged.

INPUT:

- `amp1` – a real ball (or an object that can be coerced to a real ball)

OUTPUT: a new real ball

EXAMPLES:

```
sage: err = RBF(10^-16)
sage: RBF(1).add_error(err)
[1.0000000000000000 +/- ...e-16]
```

```
>>> from sage.all import *
>>> err = RBF(Integer(10)**-Integer(16))
```

(continues on next page)

(continued from previous page)

```
>>> RBF(Integer(1)).add_error(err)
[1.0000000000000000 +/- ...e-16]
```

agm(*other*)Return the arithmetic-geometric mean of *self* and *other*.

EXAMPLES:

```
sage: RBF(1).agm(1)
1.0000000000000000
sage: RBF(sqrt(2)).agm(1)^(-1)
↪needs sage.symbolic
[0.8346268416740...]
```

#

```
>>> from sage.all import *
>>> RBF(Integer(1)).agm(Integer(1))
1.0000000000000000
>>> RBF(sqrt(Integer(2))).agm(Integer(1))**(-Integer(1))
↪ # needs sage.symbolic
[0.8346268416740...]
```

]

arccos()

Return the arccosine of this ball.

EXAMPLES:

```
sage: RBF(1).arccos()
0
sage: RBF(1, rad=.125r).arccos()
nan
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).arccos()
0
>>> RBF(Integer(1), rad=.125).arccos()
nan
```

arccosh()

Return the inverse hyperbolic cosine of this ball.

EXAMPLES:

```
sage: RBF(2).arccosh()
[1.316957896924817 +/- ...e-16]
sage: RBF(1).arccosh()
0
sage: RBF(0).arccosh()
nan
```

```
>>> from sage.all import *
>>> RBF(Integer(2)).arccosh()
[1.316957896924817 +/- ...e-16]
```

(continues on next page)

(continued from previous page)

```
>>> RBF(Integer(1)).arccosh()
0
>>> RBF(Integer(0)).arccosh()
nan
```

arcsin()

Return the arcsine of this ball.

EXAMPLES:

```
sage: RBF(1).arcsin()
[1.570796326794897 +/- ...e-16]
sage: RBF(1, rad=.125r).arcsin()
nan
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).arcsin()
[1.570796326794897 +/- ...e-16]
>>> RBF(Integer(1), rad=.125).arcsin()
nan
```

arcsinh()

Return the inverse hyperbolic sine of this ball.

EXAMPLES:

```
sage: RBF(1).arcsinh()
[0.881373587019543 +/- ...e-16]
sage: RBF(0).arcsinh()
0
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).arcsinh()
[0.881373587019543 +/- ...e-16]
>>> RBF(Integer(0)).arcsinh()
0
```

arctan()

Return the arctangent of this ball.

EXAMPLES:

```
sage: RBF(1).arctan()
[0.7853981633974483 +/- ...e-17]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).arctan()
[0.7853981633974483 +/- ...e-17]
```

arctanh()

Return the inverse hyperbolic tangent of this ball.

EXAMPLES:


```
sage: RBF(0).arctanh()
0
sage: RBF(1/2).arctanh()
[0.549306144334055 +/- ...e-16]
sage: RBF(1).arctanh()
nan
```

```
>>> from sage.all import *
>>> RBF(Integer(0)).arctanh()
0
>>> RBF(Integer(1)/Integer(2)).arctanh()
[0.549306144334055 +/- ...e-16]
>>> RBF(Integer(1)).arctanh()
nan
```

below_abs (*test_zero=False*)

Return a lower bound for the absolute value of this ball.

INPUT:

- *test_zero* – boolean (default: False); if True, make sure that the returned lower bound is positive, raising an error if the ball contains zero.

OUTPUT: a ball with zero radius

EXAMPLES:

```
sage: RealBallField(8)(1/3).below_abs()
[0.33 +/- ...e-5]
sage: b = RealBallField(8)(1/3).below_abs()
sage: b
[0.33 +/- ...e-5]
sage: b.is_exact()
True
sage: QQ(b)
169/512
```

```
sage: RBF(0).below_abs()
0
sage: RBF(0).below_abs(test_zero=True)
Traceback (most recent call last):
...
ValueError: ball contains zero
```

```
>>> from sage.all import *
>>> RealBallField(Integer(8))(Integer(1)/Integer(3)).below_abs()
[0.33 +/- ...e-5]
>>> b = RealBallField(Integer(8))(Integer(1)/Integer(3)).below_abs()
>>> b
[0.33 +/- ...e-5]
>>> b.is_exact()
True
>>> QQ(b)
169/512
```

(continues on next page)

(continued from previous page)

```
>>> RBF(Integer(0)).below_abs()
0
>>> RBF(Integer(0)).below_abs(test_zero=True)
Traceback (most recent call last):
...
ValueError: ball contains zero
```

See also`above_abs()`**beta** ($a, z=1$)

(Incomplete) beta function.

INPUT:

- a, z – (optional) real balls

OUTPUT:

The lower incomplete beta function $B(\text{self}, a, z)$.With the default value of z , the complete beta function $B(\text{self}, a)$.

EXAMPLES:

```
sage: RBF(sin(3)).beta(RBF(2/3).sqrt()) # abs tol 1e-13 #_
↪needs sage.symbolic
[7.407661629415 +/- 1.07e-13]
sage: RealBallField(100)(7/2).beta(1) # abs tol 1e-30
[0.28571428571428571428571428571 +/- 5.23e-30]
sage: RealBallField(100)(7/2).beta(1, 1/2)
[0.025253813613805268728601584361 +/- 2.53e-31]
```

```
>>> from sage.all import *
>>> RBF(sin(Integer(3))).beta(RBF(Integer(2)/Integer(3)).sqrt()) # abs tol 1e-13 #_
↪1e-13 # needs sage.symbolic
[7.407661629415 +/- 1.07e-13]
>>> RealBallField(Integer(100))(Integer(7)/Integer(2)).beta(Integer(1)) #_
↪abs tol 1e-30
[0.28571428571428571428571428571 +/- 5.23e-30]
>>> RealBallField(Integer(100))(Integer(7)/Integer(2)).beta(Integer(1), Integer(1)/Integer(2))
↪Integer(1)/Integer(2))
[0.025253813613805268728601584361 +/- 2.53e-31]
```

Todo

At the moment `RBF(beta(a,b))` does not work, one needs `RBF(a).beta(b)` for this to work. See [Issue #32851](#) and [Issue #24641](#).

ceil ()

Return the ceil of this ball.

EXAMPLES:

```
sage: RBF(1000+1/3, rad=1.r).ceil()
[1.00e+3 +/- 2.01]
```

```
>>> from sage.all import *
>>> RBF(Integer(1000)+Integer(1)/Integer(3), rad=1.r).ceil()
[1.00e+3 +/- 2.01]
```

center()

Return the center of this ball.

EXAMPLES:

```
sage: RealBallField(16)(1/3).mid()
0.3333
sage: RealBallField(16)(1/3).mid().parent()
Real Field with 16 bits of precision
sage: RealBallField(16)(RBF(1/3)).mid().parent()
Real Field with 53 bits of precision
sage: RBF('inf').mid()
+infinity
```

```
>>> from sage.all import *
>>> RealBallField(Integer(16))(Integer(1)/Integer(3)).mid()
0.3333
>>> RealBallField(Integer(16))(Integer(1)/Integer(3)).mid().parent()
Real Field with 16 bits of precision
>>> RealBallField(Integer(16))(RBF(Integer(1)/Integer(3))).mid().parent()
Real Field with 53 bits of precision
>>> RBF('inf').mid()
+infinity
```

```
sage: b = RBF(2)^(2^1000)
sage: b.mid()
+infinity
```

```
>>> from sage.all import *
>>> b = RBF(Integer(2))**(Integer(2)**Integer(1000))
>>> b.mid()
+infinity
```

See also

rad(), *squash()*

chebyshev_T(n)

Evaluate the Chebyshev polynomial of the first kind T_n at this ball.

EXAMPLES:

```

sage: # needs sage.symbolic
sage: RBF(pi).chebyshev_T(0)
1.0000000000000000
sage: RBF(pi).chebyshev_T(1)
[3.141592653589793 +/- ...e-16]
sage: RBF(pi).chebyshev_T(10**20)
Traceback (most recent call last):
...
ValueError: index too large
sage: RBF(pi).chebyshev_T(-1)
Traceback (most recent call last):
...
ValueError: expected a nonnegative index

```

```

>>> from sage.all import *
>>> # needs sage.symbolic
>>> RBF(pi).chebyshev_T(Integer(0))
1.0000000000000000
>>> RBF(pi).chebyshev_T(Integer(1))
[3.141592653589793 +/- ...e-16]
>>> RBF(pi).chebyshev_T(Integer(10)**Integer(20))
Traceback (most recent call last):
...
ValueError: index too large
>>> RBF(pi).chebyshev_T(-Integer(1))
Traceback (most recent call last):
...
ValueError: expected a nonnegative index

```

chebyshev_U(*n*)

Evaluate the Chebyshev polynomial of the second kind U_n at this ball.

EXAMPLES:

```

sage: # needs sage.symbolic
sage: RBF(pi).chebyshev_U(0)
1.0000000000000000
sage: RBF(pi).chebyshev_U(1)
[6.283185307179586 +/- ...e-16]
sage: RBF(pi).chebyshev_U(10**20)
Traceback (most recent call last):
...
ValueError: index too large
sage: RBF(pi).chebyshev_U(-1)
Traceback (most recent call last):
...
ValueError: expected a nonnegative index

```

```

>>> from sage.all import *
>>> # needs sage.symbolic
>>> RBF(pi).chebyshev_U(Integer(0))
1.0000000000000000
>>> RBF(pi).chebyshev_U(Integer(1))

```

(continues on next page)

(continued from previous page)

```
[6.283185307179586 +/- ...e-16]
>>> RBF(pi).chebyshev_U(Integer(10)**Integer(20))
Traceback (most recent call last):
...
ValueError: index too large
>>> RBF(pi).chebyshev_U(-Integer(1))
Traceback (most recent call last):
...
ValueError: expected a nonnegative index
```

contains_exact (*other*)

Return True *iff* the given number (or ball) *other* is contained in the interval represented by *self*.

If *self* contains NaN, this function always returns True (as it could represent anything, and in particular could represent all the points included in *other*). If *other* contains NaN and *self* does not, it always returns False.

Use *other* in *self* for a test that works for a wider range of inputs but may return false negatives.

EXAMPLES:

```
sage: b = RBF(1)
sage: b.contains_exact(1)
True
sage: b.contains_exact(QQ(1))
True
sage: b.contains_exact(1.)
True
sage: b.contains_exact(b)
True
```

```
>>> from sage.all import *
>>> b = RBF(Integer(1))
>>> b.contains_exact(Integer(1))
True
>>> b.contains_exact(QQ(Integer(1)))
True
>>> b.contains_exact(RealNumber('1.'))
True
>>> b.contains_exact(b)
True
```

```
sage: RBF(1/3).contains_exact(1/3)
True
sage: RBF(sqrt(2)).contains_exact(sqrt(2)) #_
↪needs sage.symbolic
Traceback (most recent call last):
...
TypeError: unsupported type: <class 'sage.symbolic.expression.Expression'>
```

```
>>> from sage.all import *
>>> RBF(Integer(1)/Integer(3)).contains_exact(Integer(1)/Integer(3))
True
```

(continues on next page)

(continued from previous page)

```
>>> RBF(sqrt(Integer(2))).contains_exact(sqrt(Integer(2)))
↳ # needs sage.symbolic
Traceback (most recent call last):
...
TypeError: unsupported type: <class 'sage.symbolic.expression.Expression'>
```

contains_integer()

Return True iff this ball contains any integer.

EXAMPLES:

```
sage: RBF(3.1, 0.1).contains_integer()
True
sage: RBF(3.1, 0.05).contains_integer()
False
```

```
>>> from sage.all import *
>>> RBF(RealNumber('3.1'), RealNumber('0.1')).contains_integer()
True
>>> RBF(RealNumber('3.1'), RealNumber('0.05')).contains_integer()
False
```

contains_zero()

Return True iff this ball contains zero.

EXAMPLES:

```
sage: RBF(0).contains_zero()
True
sage: RBF(RIF(-1, 1)).contains_zero()
True
sage: RBF(1/3).contains_zero()
False
```

```
>>> from sage.all import *
>>> RBF(Integer(0)).contains_zero()
True
>>> RBF(RIF(-Integer(1), Integer(1))).contains_zero()
True
>>> RBF(Integer(1)/Integer(3)).contains_zero()
False
```

cos()

Return the cosine of this ball.

EXAMPLES:

```
sage: RBF(pi).cos()
↳ needs sage.symbolic
[-1.000000000000000 +/- ...e-16]
```

```
>>> from sage.all import *
>>> RBF(pi).cos()
```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
[-1.000000000000000 +/- ...e-16]
```

See also

```
cospi()
```

cos_integral()

Cosine integral.

EXAMPLES:

```
sage: RBF(1).Ci() # abs tol 5e-16
[0.337403922900968 +/- 3.25e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).Ci() # abs tol 5e-16
[0.337403922900968 +/- 3.25e-16]
```

cosh()

Return the hyperbolic cosine of this ball.

EXAMPLES:

```
sage: RBF(1).cosh()
[1.543080634815244 +/- ...e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).cosh()
[1.543080634815244 +/- ...e-16]
```

cosh_integral()

Hyperbolic cosine integral.

EXAMPLES:

```
sage: RBF(1).Chi() # abs tol 5e-16
[0.837866940980208 +/- 4.72e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).Chi() # abs tol 5e-16
[0.837866940980208 +/- 4.72e-16]
```

cot()

Return the cotangent of this ball.

EXAMPLES:

```
sage: RBF(1).cot()
[0.642092615934331 +/- ...e-16]
sage: RBF(pi).cot()
```

#

(continues on next page)

(continued from previous page)

```
↳needs sage.symbolic
nan
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).cot()
[0.642092615934331 +/- ...e-16]
>>> RBF(pi).cot()
↳needs sage.symbolic
nan
```

coth()

Return the hyperbolic cotangent of this ball.

EXAMPLES:

```
sage: RBF(1).coth()
[1.313035285499331 +/- ...e-16]
sage: RBF(0).coth()
nan
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).coth()
[1.313035285499331 +/- ...e-16]
>>> RBF(Integer(0)).coth()
nan
```

csc()

Return the cosecant of this ball.

EXAMPLES:

```
sage: RBF(1).csc()
[1.188395105778121 +/- ...e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).csc()
[1.188395105778121 +/- ...e-16]
```

csch()

Return the hyperbolic cosecant of this ball.

EXAMPLES:

```
sage: RBF(1).csch()
[0.850918128239321 +/- ...e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).csch()
[0.850918128239321 +/- ...e-16]
```

diameter()

Return the diameter of this ball.

EXAMPLES:


```
sage: RBF(1/3).diameter()
1.1102230e-16
sage: RBF(1/3).diameter().parent()
Real Field with 30 bits of precision
sage: RBF(RIF(1.02, 1.04)).diameter()
0.020000000
```

```
>>> from sage.all import *
>>> RBF(Integer(1)/Integer(3)).diameter()
1.1102230e-16
>>> RBF(Integer(1)/Integer(3)).diameter().parent()
Real Field with 30 bits of precision
>>> RBF(RIF(RealNumber('1.02'), RealNumber('1.04'))).diameter()
0.020000000
```

See also

`rad()`, `rad_as_ball()`, `mid()`

endpoints (*rnd=None*)

Return the endpoints of this ball, rounded outwards.

INPUT:

- `rnd` – string; rounding mode for the parent of the resulting floating-point numbers (does not affect their values!), see `sage.rings.real_mpfi.RealIntervalFieldElement.upper()`

OUTPUT: a pair of real numbers

EXAMPLES:

```
sage: RBF(-1/3).endpoints()
(-0.3333333333333334, -0.3333333333333333)
```

```
>>> from sage.all import *
>>> RBF(-Integer(1)/Integer(3)).endpoints()
(-0.3333333333333334, -0.3333333333333333)
```

See also

`lower()`, `upper()`

erf()

Error function.

EXAMPLES:

```
sage: RBF(1/2).erf() # abs tol 1e-16
[0.520499877813047 +/- 6.10e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)/Integer(2)).erf() # abs tol 1e-16
[0.520499877813047 +/- 6.10e-16]
```

erfi()

Imaginary error function.

EXAMPLES:

```
sage: RBF(1/2).erfi()
[0.614952094696511 +/- 2.22e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)/Integer(2)).erfi()
[0.614952094696511 +/- 2.22e-16]
```

exp()

Return the exponential of this ball.

EXAMPLES:

```
sage: RBF(1).exp()
[2.718281828459045 +/- ...e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).exp()
[2.718281828459045 +/- ...e-16]
```

expm1()

Return $\exp(\text{self}) - 1$, computed accurately when *self* is close to zero.

EXAMPLES:

```
sage: eps = RBF(1e-30)
sage: exp(eps) - 1
[+/- ...e-30]
sage: eps.expm1()
[1.0000000000000000e-30 +/- ...e-47]
```

```
>>> from sage.all import *
>>> eps = RBF(RealNumber('1e-30'))
>>> exp(eps) - Integer(1)
[+/- ...e-30]
>>> eps.expm1()
[1.0000000000000000e-30 +/- ...e-47]
```

floor()

Return the floor of this ball.

EXAMPLES:

```
sage: RBF(1000+1/3, rad=1.r).floor()
[1.00e+3 +/- 1.01]
```

```
>>> from sage.all import *
>>> RBF(Integer(1000)+Integer(1)/Integer(3), rad=1.).floor()
[1.00e+3 +/- 1.01]
```

gamma (*a=None*)

Image of this ball by the (upper incomplete) Euler Gamma function.

For *a* real, return the upper incomplete Gamma function $\Gamma(\text{self}, a)$.

For integer and rational arguments, `gamma()` may be faster.

EXAMPLES:

```
sage: RBF(1/2).gamma()
[1.772453850905516 +/- ...e-16]
sage: RBF(gamma(3/2, RBF(2).sqrt())) # abs tol 2e-17
[0.37118875695353 +/- 3.00e-15]
sage: RBF(3/2).gamma_inc(RBF(2).sqrt()) # abs tol 2e-17
[0.37118875695353 +/- 3.00e-15]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)/Integer(2)).gamma()
[1.772453850905516 +/- ...e-16]
>>> RBF(gamma(Integer(3)/Integer(2), RBF(Integer(2)).sqrt())) # abs tol 2e-17
[0.37118875695353 +/- 3.00e-15]
>>> RBF(Integer(3)/Integer(2)).gamma_inc(RBF(Integer(2)).sqrt()) # abs tol 2e-17
[0.37118875695353 +/- 3.00e-15]
```

See also

`gamma()`

gamma_inc (*a=None*)

Image of this ball by the (upper incomplete) Euler Gamma function.

For *a* real, return the upper incomplete Gamma function $\Gamma(\text{self}, a)$.

For integer and rational arguments, `gamma()` may be faster.

EXAMPLES:

```
sage: RBF(1/2).gamma()
[1.772453850905516 +/- ...e-16]
sage: RBF(gamma(3/2, RBF(2).sqrt())) # abs tol 2e-17
[0.37118875695353 +/- 3.00e-15]
sage: RBF(3/2).gamma_inc(RBF(2).sqrt()) # abs tol 2e-17
[0.37118875695353 +/- 3.00e-15]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)/Integer(2)).gamma()
[1.772453850905516 +/- ...e-16]
>>> RBF(gamma(Integer(3)/Integer(2), RBF(Integer(2)).sqrt())) # abs tol 2e-17
[0.37118875695353 +/- 3.00e-15]
```

(continues on next page)

(continued from previous page)

```
>>> RBF(Integer(3)/Integer(2)).gamma_inc(RBF(Integer(2)).sqrt()) # abs tol
↪ 2e-17
[0.37118875695353 +/- 3.00e-15]
```

See also[`gamma\(\)`](#)**gamma_inc_lower(*a*)**

Image of this ball by the lower incomplete Euler Gamma function.

For *a* real, return the lower incomplete Gamma function of $\Gamma(\text{self}, a)$.

EXAMPLES:

```
sage: RBF(gamma_inc_lower(1/2, RBF(2).sqrt()))
[1.608308637729248 +/- 8.14e-16]
sage: RealBallField(100)(7/2).gamma_inc_lower(5)
[2.6966551541863035516887949614 +/- 8.91e-29]
```

```
>>> from sage.all import *
>>> RBF(gamma_inc_lower(Integer(1)/Integer(2), RBF(Integer(2)).sqrt()))
[1.608308637729248 +/- 8.14e-16]
>>> RealBallField(Integer(100))(Integer(7)/Integer(2)).gamma_inc_
↪ lower(Integer(5))
[2.6966551541863035516887949614 +/- 8.91e-29]
```

identical(*other*)

Return True iff *self* and *other* are equal as balls, i.e. have both the same midpoint and radius.

Note that this is not the same thing as testing whether both *self* and *other* certainly represent the same real number, unless either *self* or *other* is exact (and neither contains NaN). To test whether both operands might represent the same mathematical quantity, use [`overlaps\(\)`](#) or [`contains\(\)`](#), depending on the circumstance.

EXAMPLES:

```
sage: RBF(1).identical(RBF(3)-RBF(2))
True
sage: RBF(1, rad=0.25r).identical(RBF(1, rad=0.25r))
True
sage: RBF(1).identical(RBF(1, rad=0.25r))
False
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).identical(RBF(Integer(3))-RBF(Integer(2)))
True
>>> RBF(Integer(1), rad=0.25).identical(RBF(Integer(1), rad=0.25))
True
>>> RBF(Integer(1)).identical(RBF(Integer(1), rad=0.25))
False
```

imag()

Return the imaginary part of this ball.

EXAMPLES:

```
sage: RBF(1/3).imag()
0
```

```
>>> from sage.all import *
>>> RBF(Integer(1)/Integer(3)).imag()
0
```

is_NaN()

Return True if this ball is not-a-number.

EXAMPLES:

```
sage: RBF(NaN).is_NaN() #_
↪needs sage.symbolic
True
sage: RBF(-5).gamma().is_NaN()
True
sage: RBF(infinity).is_NaN()
False
sage: RBF(42, rad=1.r).is_NaN()
False
```

```
>>> from sage.all import *
>>> RBF(NaN).is_NaN() #_
↪needs sage.symbolic
True
>>> RBF(-Integer(5)).gamma().is_NaN()
True
>>> RBF(infinity).is_NaN()
False
>>> RBF(Integer(42), rad=1.).is_NaN()
False
```

is_exact()

Return True iff the radius of this ball is zero.

EXAMPLES:

```
sage: RBF = RealBallField()
sage: RBF(1).is_exact()
True
sage: RBF(RIF(0.1, 0.2)).is_exact()
False
```

```
>>> from sage.all import *
>>> RBF = RealBallField()
>>> RBF(Integer(1)).is_exact()
True
```

(continues on next page)

(continued from previous page)

```
>>> RBF(RIF(RealNumber('0.1'), RealNumber('0.2'))).is_exact()
False
```

is_finite()

Return True iff the midpoint and radius of this ball are both finite floating-point numbers, i.e. not infinities or NaN.

EXAMPLES:

```
sage: (RBF(2)^(2^1000)).is_finite()
True
sage: RBF(oo).is_finite()
False
```

```
>>> from sage.all import *
>>> (RBF(Integer(2)**(Integer(2)**Integer(1000))).is_finite()
True
>>> RBF(oo).is_finite()
False
```

is_infinity()

Return True if this ball contains or may represent a point at infinity.

This is the exact negation of `is_finite()`, used in comparisons with Sage symbolic infinities.

Warning

Contrary to the usual convention, a return value of `True` does not imply that all points of the ball satisfy the predicate. This is due to the way comparisons with symbolic infinities work in sage.

EXAMPLES:

```
sage: RBF(infinity).is_infinity()
True
sage: RBF(-infinity).is_infinity()
True
sage: RBF(NaN).is_infinity()
↪needs sage.symbolic
True
sage: (~RBF(0)).is_infinity()
True
sage: RBF(42, rad=1.r).is_infinity()
False
```

```
>>> from sage.all import *
>>> RBF(infinity).is_infinity()
True
>>> RBF(-infinity).is_infinity()
True
>>> RBF(NaN).is_infinity()
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
True
>>> (~RBF(Integer(0))).is_infinity()
True
>>> RBF(Integer(42), rad=1.).is_infinity()
False
```

is_negative_infinity()

Return True if this ball is the point $-\infty$.

EXAMPLES:

```
sage: RBF(-infinity).is_negative_infinity()
True
```

```
>>> from sage.all import *
>>> RBF(-infinity).is_negative_infinity()
True
```

is_nonzero()

Return True iff zero is not contained in the interval represented by this ball.

Note

This method is not the negation of `is_zero()`: it only returns True if zero is known not to be contained in the ball.

Use `bool(b)` (or, equivalently, `not b.is_zero()`) to check if a ball **b** **may** represent a nonzero number (for instance, to determine the “degree” of a polynomial with ball coefficients).

EXAMPLES:

```
sage: RBF = RealBallField()
sage: RBF(pi).is_nonzero()
↪needs sage.symbolic
True
sage: RBF(RIF(-0.5, 0.5)).is_nonzero()
False
```

```
>>> from sage.all import *
>>> RBF = RealBallField()
>>> RBF(pi).is_nonzero()
↪needs sage.symbolic
True
>>> RBF(RIF(-RealNumber('0.5'), RealNumber('0.5'))).is_nonzero()
False
```

See also

`is_zero()`

is_positive_infinity()

Return True if this ball is the point $+\infty$.

EXAMPLES:

```
sage: RBF(infinity).is_positive_infinity()
True
```

```
>>> from sage.all import *
>>> RBF(infinity).is_positive_infinity()
True
```

is_zero()

Return True iff the midpoint and radius of this ball are both zero.

EXAMPLES:

```
sage: RBF = RealBallField()
sage: RBF(0).is_zero()
True
sage: RBF(RIF(-0.5, 0.5)).is_zero()
False
```

```
>>> from sage.all import *
>>> RBF = RealBallField()
>>> RBF(Integer(0)).is_zero()
True
>>> RBF(RIF(-RealNumber('0.5'), RealNumber('0.5'))).is_zero()
False
```

See also

is_nonzero()

lambert_w()

Return the image of this ball by the Lambert W function.

EXAMPLES:

```
sage: RBF(1).lambert_w()
[0.5671432904097...]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).lambert_w()
[0.5671432904097...]
```

li()

Logarithmic integral.

EXAMPLES:

```
sage: RBF(3).li() # abs tol 5e-15
[2.16358859466719 +/- 4.72e-15]
```



```
>>> from sage.all import *
>>> RBF(Integer(3)).li() # abs tol 5e-15
[2.16358859466719 +/- 4.72e-15]
```

log (*base=None*)

Return the logarithm of this ball.

INPUT:

- *base* – (optional) positive real ball or number; if *None*, return the natural logarithm $\ln(\text{self})$, otherwise, return the general logarithm $\ln(\text{self}) / \ln(\text{base})$

EXAMPLES:

```
sage: RBF(3).log()
[1.098612288668110 +/- ...e-16]
sage: RBF(3).log(2)
[1.58496250072116 +/- ...e-15]
sage: log(RBF(5), 2)
[2.32192809488736 +/- ...e-15]

sage: RBF(-1/3).log()
nan
sage: RBF(3).log(-1)
nan
sage: RBF(2).log(0)
nan
```

```
>>> from sage.all import *
>>> RBF(Integer(3)).log()
[1.098612288668110 +/- ...e-16]
>>> RBF(Integer(3)).log(Integer(2))
[1.58496250072116 +/- ...e-15]
>>> log(RBF(Integer(5)), Integer(2))
[2.32192809488736 +/- ...e-15]

>>> RBF(-Integer(1)/Integer(3)).log()
nan
>>> RBF(Integer(3)).log(-Integer(1))
nan
>>> RBF(Integer(2)).log(Integer(0))
nan
```

log1p ()

Return $\log(1 + \text{self})$, computed accurately when *self* is close to zero.

EXAMPLES:

```
sage: eps = RBF(1e-30)
sage: (1 + eps).log()
[+/- ...e-16]
sage: eps.log1p()
[1.000000000000000e-30 +/- ...e-46]
```

```
>>> from sage.all import *
>>> eps = RBF(RealNumber('1e-30'))
>>> (Integer(1) + eps).log()
[+/- ...e-16]
>>> eps.log1p()
[1.000000000000000e-30 +/- ...e-46]
```

log_gamma()

Return the image of this ball by the logarithmic Gamma function.

The complex branch structure is assumed, so if `self <= 0`, the result is an indeterminate interval.

EXAMPLES:

```
sage: RBF(1/2).log_gamma()
[0.572364942924700 +/- ...e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)/Integer(2)).log_gamma()
[0.572364942924700 +/- ...e-16]
```

log_integral()

Logarithmic integral.

EXAMPLES:

```
sage: RBF(3).li() # abs tol 5e-15
[2.16358859466719 +/- 4.72e-15]
```

```
>>> from sage.all import *
>>> RBF(Integer(3)).li() # abs tol 5e-15
[2.16358859466719 +/- 4.72e-15]
```

log_integral_offset()

Offset logarithmic integral.

EXAMPLES:

```
sage: RBF(3).Li() # abs tol 5e-15
[1.11842481454970 +/- 7.61e-15]
```

```
>>> from sage.all import *
>>> RBF(Integer(3)).Li() # abs tol 5e-15
[1.11842481454970 +/- 7.61e-15]
```

lower(rnd=None)

Return the right endpoint of this ball, rounded downwards.

INPUT:

- `rnd` – string; rounding mode for the parent of the result (does not affect its value!), see `sage.rings.real_mpfi.RealIntervalFieldElement.lower()`

OUTPUT: a real number

EXAMPLES:

```
sage: RBF(-1/3).lower()
-0.3333333333333334
sage: RBF(-1/3).lower().parent()
Real Field with 53 bits of precision and rounding RNDD
```

```
>>> from sage.all import *
>>> RBF(-Integer(1)/Integer(3)).lower()
-0.3333333333333334
>>> RBF(-Integer(1)/Integer(3)).lower().parent()
Real Field with 53 bits of precision and rounding RNDD
```

See also

upper(), *endpoints()*

max(*others)

Return a ball containing the maximum of this ball and the remaining arguments.

EXAMPLES:

```
sage: RBF(-1, rad=.5).max(0)
0

sage: RBF(0, rad=2.).max(RBF(0, rad=1.)).endpoints()
(-1.000000000465662, 2.000000000651926)

sage: RBF(-infinity).max(-3, 1/3)
[0.3333333333333333 +/- ...e-17]

sage: RBF('nan').max(0)
nan
```

```
>>> from sage.all import *
>>> RBF(-Integer(1), rad=RealNumber('.5')).max(Integer(0))
0

>>> RBF(Integer(0), rad=RealNumber('2.')).max(RBF(Integer(0), rad=RealNumber(
↪ '1.'))).endpoints()
(-1.000000000465662, 2.000000000651926)

>>> RBF(-infinity).max(-Integer(3), Integer(1)/Integer(3))
[0.3333333333333333 +/- ...e-17]

>>> RBF('nan').max(Integer(0))
nan
```

See also

min()

mid()

Return the center of this ball.

EXAMPLES:

```
sage: RealBallField(16) (1/3).mid()
0.3333
sage: RealBallField(16) (1/3).mid().parent()
Real Field with 16 bits of precision
sage: RealBallField(16) (RBF(1/3)).mid().parent()
Real Field with 53 bits of precision
sage: RBF('inf').mid()
+infinity
```

```
>>> from sage.all import *
>>> RealBallField(Integer(16)) (Integer(1)/Integer(3)).mid()
0.3333
>>> RealBallField(Integer(16)) (Integer(1)/Integer(3)).mid().parent()
Real Field with 16 bits of precision
>>> RealBallField(Integer(16)) (RBF(Integer(1)/Integer(3))).mid().parent()
Real Field with 53 bits of precision
>>> RBF('inf').mid()
+infinity
```

```
sage: b = RBF(2)^(2^1000)
sage: b.mid()
+infinity
```

```
>>> from sage.all import *
>>> b = RBF(Integer(2)**(Integer(2)**Integer(1000)))
>>> b.mid()
+infinity
```

See also*rad()*, *squash()***min(*others)**

Return a ball containing the minimum of this ball and the remaining arguments.

EXAMPLES:

```
sage: RBF(1, rad=.5).min(0)
0
sage: RBF(0, rad=2.).min(RBF(0, rad=1.)).endpoints()
(-2.000000000651926, 1.000000000465662)
sage: RBF(infinity).min(3, 1/3)
[0.3333333333333333 +/- ...e-17]
```

(continues on next page)

(continued from previous page)

```
sage: RBF('nan').min(0)
nan
```

```
>>> from sage.all import *
>>> RBF(Integer(1), rad=RealNumber('.5')).min(Integer(0))
0

>>> RBF(Integer(0), rad=RealNumber('2.')).min(RBF(Integer(0), rad=RealNumber(
↳ '1.')).endpoints())
(-2.000000000651926, 1.000000000465662)

>>> RBF(infinity).min(Integer(3), Integer(1)/Integer(3))
[0.3333333333333333 +/- ...e-17]

>>> RBF('nan').min(Integer(0))
nan
```

See also[`max\(\)`](#)**nbits()**

Return the minimum precision sufficient to represent this ball exactly.

In other words, return the number of bits needed to represent the absolute value of the mantissa of the midpoint of this ball. The result is 0 if the midpoint is a special value.

EXAMPLES:

```
sage: RBF(1/3).nbits()
53
sage: RBF(1023, .1).nbits()
10
sage: RBF(1024, .1).nbits()
1
sage: RBF(0).nbits()
0
sage: RBF(infinity).nbits()
0
```

```
>>> from sage.all import *
>>> RBF(Integer(1)/Integer(3)).nbits()
53
>>> RBF(Integer(1023), RealNumber('.1')).nbits()
10
>>> RBF(Integer(1024), RealNumber('.1')).nbits()
1
>>> RBF(Integer(0)).nbits()
0
>>> RBF(infinity).nbits()
0
```

overlaps (*other*)

Return True iff self and other have some point in common.

If either self or other contains NaN, this method always returns nonzero (as a NaN could be anything, it could in particular contain any number that is included in the other operand).

EXAMPLES:

```
sage: RBF(pi).overlaps(RBF(pi) + 2**(-100)) #_
↪needs sage.symbolic
True
sage: RBF(pi).overlaps(RBF(3)) #_
↪needs sage.symbolic
False
```

```
>>> from sage.all import *
>>> RBF(pi).overlaps(RBF(pi) + Integer(2)**(-Integer(100))) #_
↪ # needs sage.symbolic
True
>>> RBF(pi).overlaps(RBF(Integer(3))) #_
↪ # needs sage.symbolic
False
```

polylog (*s*)

Return the polylogarithm $\text{Li}_s(\text{self})$.

EXAMPLES:

```
sage: polylog(0, -1) #_
↪needs sage.symbolic
-1/2
sage: RBF(-1).polylog(0)
[-0.500000000000000 +/- ...e-16]
sage: polylog(1, 1/2) #_
↪needs sage.symbolic
-log(1/2)
sage: RBF(1/2).polylog(1)
[0.69314718055995 +/- ...e-15]
sage: RBF(1/3).polylog(1/2)
[0.44210883528067 +/- 6.7...e-15]
sage: RBF(1/3).polylog(RBF(pi)) #_
↪needs sage.symbolic
[0.34728895057225 +/- ...e-15]
```

```
>>> from sage.all import *
>>> polylog(Integer(0), -Integer(1)) #_
↪ # needs sage.symbolic
-1/2
>>> RBF(-Integer(1)).polylog(Integer(0))
[-0.500000000000000 +/- ...e-16]
>>> polylog(Integer(1), Integer(1)/Integer(2)) #_
↪ # needs sage.symbolic
-log(1/2)
>>> RBF(Integer(1)/Integer(2)).polylog(Integer(1))
```

(continues on next page)

(continued from previous page)

```
[0.69314718055995 +/- ...e-15]
>>> RBF(Integer(1)/Integer(3)).polylog(Integer(1)/Integer(2))
[0.44210883528067 +/- 6.7...e-15]
>>> RBF(Integer(1)/Integer(3)).polylog(RLF(pi))
↪ # needs sage.symbolic
[0.34728895057225 +/- ...e-15]
```

psi()

Compute the digamma function with argument `self`.

EXAMPLES:

```
sage: RBF(1).psi() # abs tol 1e-15
[-0.5772156649015329 +/- 4.84e-17]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).psi() # abs tol 1e-15
[-0.5772156649015329 +/- 4.84e-17]
```

rad()

Return the radius of this ball.

EXAMPLES:

```
sage: RBF(1/3).rad()
5.5511151e-17
sage: RBF(1/3).rad().parent()
Real Field with 30 bits of precision
```

```
>>> from sage.all import *
>>> RBF(Integer(1)/Integer(3)).rad()
5.5511151e-17
>>> RBF(Integer(1)/Integer(3)).rad().parent()
Real Field with 30 bits of precision
```

See also

`mid()`, `rad_as_ball()`, `diameter()`

rad_as_ball()

Return an exact ball with center equal to the radius of this ball.

EXAMPLES:

```
sage: rad = RBF(1/3).rad_as_ball()
sage: rad
[5.55111512e-17 +/- ...e-26]
sage: rad.is_exact()
True
sage: rad.parent()
Real ball field with 30 bits of precision
```

```
>>> from sage.all import *
>>> rad = RBF(Integer(1)/Integer(3)).rad_as_ball()
>>> rad
[5.55111512e-17 +/- ...e-26]
>>> rad.is_exact()
True
>>> rad.parent()
Real ball field with 30 bits of precision
```

See also

squash(), *rad()*

real()

Return the real part of this ball.

EXAMPLES:

```
sage: RBF(1/3).real()
[0.333333333333333 +/- 7.04e-17]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)/Integer(3)).real()
[0.333333333333333 +/- 7.04e-17]
```

rgamma()

Return the image of this ball by the function $1/\Gamma$, avoiding division by zero at the poles of the gamma function.

EXAMPLES:

```
sage: RBF(-1).rgamma()
0
sage: RBF(3).rgamma()
0.5000000000000000
```

```
>>> from sage.all import *
>>> RBF(-Integer(1)).rgamma()
0
>>> RBF(Integer(3)).rgamma()
0.5000000000000000
```

rising_factorial(n)

Return the n -th rising factorial of this ball.

The n -th rising factorial of x is equal to $x(x+1)\cdots(x+n-1)$.

For real n , it is a quotient of gamma functions.

EXAMPLES:

```
sage: RBF(1).rising_factorial(5)
120.00000000000000
sage: RBF(1/2).rising_factorial(1/3) # abs tol 1e-14
[0.636849884317974 +/- 8.98e-16]
```



```
>>> from sage.all import *
>>> RBF(Integer(1)).rising_factorial(Integer(5))
120.00000000000000
>>> RBF(Integer(1)/Integer(2)).rising_factorial(Integer(1)/Integer(3)) # abs_
↳tol 1e-14
[0.636849884317974 +/- 8.98e-16]
```

round()

Return a copy of this ball with center rounded to the precision of the parent.

EXAMPLES:

It is possible to create balls whose midpoint is more precise than their parent's nominal precision (see [real_arb](#) for more information):

```
sage: b = RBF(pi.n(100)) #_
↳needs sage.symbolic
sage: b.mid() #_
↳needs sage.symbolic
3.141592653589793238462643383
```

```
>>> from sage.all import *
>>> b = RBF(pi.n(Integer(100))) #_
↳ # needs sage.symbolic
>>> b.mid() #_
↳needs sage.symbolic
3.141592653589793238462643383
```

The `round()` method rounds such a ball to its parent's precision:

```
sage: b.round().mid() #_
↳needs sage.symbolic
3.14159265358979
```

```
>>> from sage.all import *
>>> b.round().mid() #_
↳needs sage.symbolic
3.14159265358979
```

See also

[trim\(\)](#)

rsqrt()

Return the reciprocal square root of `self`.

At high precision, this is faster than computing a square root.

EXAMPLES:

```
sage: RBF(2).rsqrt()
[0.707106781186547 +/- ...e-16]
```

(continues on next page)

(continued from previous page)

```
sage: RBF(0).rsqrt()
nan
```

```
>>> from sage.all import *
>>> RBF(Integer(2)).rsqrt()
[0.707106781186547 +/- ...e-16]
>>> RBF(Integer(0)).rsqrt()
nan
```

sec()

Return the secant of this ball.

EXAMPLES:

```
sage: RBF(1).sec()
[1.850815717680925 +/- ...e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).sec()
[1.850815717680925 +/- ...e-16]
```

sech()

Return the hyperbolic secant of this ball.

EXAMPLES:

```
sage: RBF(1).sech()
[0.648054273663885 +/- ...e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).sech()
[0.648054273663885 +/- ...e-16]
```

sin()

Return the sine of this ball.

EXAMPLES:

```
sage: RBF(pi).sin()
↪needs sage.symbolic
[+/- ...e-16]
```

```
>>> from sage.all import *
>>> RBF(pi).sin()
↪needs sage.symbolic
[+/- ...e-16]
```

See also

`sinpi()`

sin_integral()

Sine integral.

EXAMPLES:

```
sage: RBF(1).Si() # abs tol 1e-15
[0.946083070367183 +/- 9.22e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).Si() # abs tol 1e-15
[0.946083070367183 +/- 9.22e-16]
```

sinh()

Return the hyperbolic sine of this ball.

EXAMPLES:

```
sage: RBF(1).sinh()
[1.175201193643801 +/- ...e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).sinh()
[1.175201193643801 +/- ...e-16]
```

sinh_integral()

Hyperbolic sine integral.

EXAMPLES:

```
sage: RBF(1).Shi() # abs tol 5e-15
[1.05725087537573 +/- 2.77e-15]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).Shi() # abs tol 5e-15
[1.05725087537573 +/- 2.77e-15]
```

sqrt()

Return the square root of this ball.

EXAMPLES:

```
sage: RBF(2).sqrt()
[1.414213562373095 +/- ...e-16]
sage: RBF(-1/3).sqrt()
nan
```

```
>>> from sage.all import *
>>> RBF(Integer(2)).sqrt()
[1.414213562373095 +/- ...e-16]
>>> RBF(-Integer(1)/Integer(3)).sqrt()
nan
```

sqrt1pm1()Return $\sqrt{1 + \text{self}} - 1$, computed accurately when `self` is close to zero.

EXAMPLES:

```
sage: eps = RBF(10^(-20))
sage: (1 + eps).sqrt() - 1
[+/- ...e-16]
sage: eps.sqrt1pm1()
[5.000000000000000e-21 +/- ...e-36]
```

```
>>> from sage.all import *
>>> eps = RBF(Integer(10)**(-Integer(20)))
>>> (Integer(1) + eps).sqrt() - Integer(1)
[+/- ...e-16]
>>> eps.sqrt1pm1()
[5.000000000000000e-21 +/- ...e-36]
```

sqrtpos()

Return the square root of this ball, assuming that it represents a nonnegative number.

Any negative numbers in the input interval are discarded.

EXAMPLES:

```
sage: RBF(2).sqrtpos()
[1.414213562373095 +/- ...e-16]
sage: RBF(-1/3).sqrtpos()
0
sage: RBF(0, rad=2.r).sqrtpos()
[+/- 1.42]
```

```
>>> from sage.all import *
>>> RBF(Integer(2)).sqrtpos()
[1.414213562373095 +/- ...e-16]
>>> RBF(-Integer(1)/Integer(3)).sqrtpos()
0
>>> RBF(Integer(0), rad=2.).sqrtpos()
[+/- 1.42]
```

squash()

Return an exact ball with the same center as this ball.

EXAMPLES:

```
sage: mid = RealBallField(16)(1/3).squash()
sage: mid
[0.3333 +/- ...e-5]
sage: mid.is_exact()
True
sage: mid.parent()
Real ball field with 16 bits of precision
```

```
>>> from sage.all import *
>>> mid = RealBallField(Integer(16))(Integer(1)/Integer(3)).squash()
>>> mid
[0.3333 +/- ...e-5]
```

(continues on next page)

(continued from previous page)

```
>>> mid.is_exact()
True
>>> mid.parent()
Real ball field with 16 bits of precision
```

See also

`mid()`, `rad_as_ball()`

tan()

Return the tangent of this ball.

EXAMPLES:

```
sage: RBF(1).tan()
[1.557407724654902 +/- ...e-16]
sage: RBF(pi/2).tan()
↳needs sage.symbolic
nan
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).tan()
[1.557407724654902 +/- ...e-16]
>>> RBF(pi/Integer(2)).tan()
↳ # needs sage.symbolic
nan
```

tanh()

Return the hyperbolic tangent of this ball.

EXAMPLES:

```
sage: RBF(1).tanh()
[0.761594155955765 +/- ...e-16]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)).tanh()
[0.761594155955765 +/- ...e-16]
```

trim()

Return a trimmed copy of this ball.

Round `self` to a number of bits equal to the `accuracy()` of `self` (as indicated by its radius), plus a few guard bits. The resulting ball is guaranteed to contain `self`, but is more economical if `self` has less than full accuracy.

EXAMPLES:

```
sage: b = RBF(0.111111111111111, rad=.001)
sage: b.mid()
0.111111111111110
sage: b.trim().mid()
0.111111104488373
```

```
>>> from sage.all import *
>>> b = RBF(RealNumber('0.11111111111111'), rad=RealNumber('.001'))
>>> b.mid()
0.111111111111110
>>> b.trim().mid()
0.111111104488373
```

See also*round()***union** (*other*)

Return a ball containing the convex hull of *self* and *other*.

EXAMPLES:

```
sage: RBF(0).union(1).endpoints()
(-9.31322574615479e-10, 1.00000000093133)
```

```
>>> from sage.all import *
>>> RBF(Integer(0)).union(Integer(1)).endpoints()
(-9.31322574615479e-10, 1.00000000093133)
```

upper (*rnd=None*)

Return the right endpoint of this ball, rounded upwards.

INPUT:

- *rnd* – string; rounding mode for the parent of the result (does not affect its value!), see *sage.rings.real_mpfi.RealIntervalFieldElement.upper()*

OUTPUT: a real number

EXAMPLES:

```
sage: RBF(-1/3).upper()
-0.333333333333333
sage: RBF(-1/3).upper().parent()
Real Field with 53 bits of precision and rounding RNDU
```

```
>>> from sage.all import *
>>> RBF(-Integer(1)/Integer(3)).upper()
-0.333333333333333
>>> RBF(-Integer(1)/Integer(3)).upper().parent()
Real Field with 53 bits of precision and rounding RNDU
```

See also*lower()*, *endpoints()***zeta** (*a=None*)

Return the image of this ball by the Hurwitz zeta function.

For $a = 1$ (or $a = \text{None}$), this computes the Riemann zeta function.

Otherwise, it computes the Hurwitz zeta function.

Use `RealBallField.zeta()` to compute the Riemann zeta function of a small integer without first converting it to a real ball.

EXAMPLES:

```
sage: RBF(-1).zeta()
[-0.08333333333333333 +/- ...e-17]
sage: RBF(-1).zeta(1)
[-0.08333333333333333 +/- ...e-17]
sage: RBF(-1).zeta(2)
[-1.0833333333333333 +/- ...e-16]
```

```
>>> from sage.all import *
>>> RBF(-Integer(1)).zeta()
[-0.08333333333333333 +/- ...e-17]
>>> RBF(-Integer(1)).zeta(Integer(1))
[-0.08333333333333333 +/- ...e-17]
>>> RBF(-Integer(1)).zeta(Integer(2))
[-1.0833333333333333 +/- ...e-16]
```

zetaderiv(k)

Return the image of this ball by the k -th derivative of the Riemann zeta function.

For a more flexible interface, see the low-level method `_zeta_series` of polynomials with complex ball coefficients.

EXAMPLES:

```
sage: RBF(1/2).zetaderiv(1)
[-3.92264613920915...]
sage: RBF(2).zetaderiv(3)
[-6.0001458028430...]
```

```
>>> from sage.all import *
>>> RBF(Integer(1)/Integer(2)).zetaderiv(Integer(1))
[-3.92264613920915...]
>>> RBF(Integer(2)).zetaderiv(Integer(3))
[-6.0001458028430...]
```

class `sage.rings.real_arb.RealBallField` (*precision=53*)

Bases: `UniqueRepresentation, RealBallField`

An approximation of the field of real numbers using mid-rad intervals, also known as balls.

INPUT:

- `precision` – integer ≥ 2

EXAMPLES:

```
sage: RBF = RealBallField() # indirect doctest
sage: RBF(1)
1.0000000000000000
```

```
>>> from sage.all import *
>>> RBF = RealBallField() # indirect doctest
>>> RBF(Integer(1))
1.0000000000000000
```

```
sage: (1/2*RBF(1)) + AA(sqrt(2)) - 1 + polygen(QQ, 'x') #
↪needs sage.symbolic
x + [0.914213562373095 +/- ...e-16]
```

```
>>> from sage.all import *
>>> (Integer(1)/Integer(2)*RBF(Integer(1))) + AA(sqrt(Integer(2))) - Integer(1) +
↪polygen(QQ, 'x') # needs sage.symbolic
x + [0.914213562373095 +/- ...e-16]
```

See also

- `sage.rings.real_arb`
- `sage.rings.real_mpfr`
- `sage.rings.real_mpfi` (real intervals represented by their endpoints)
- `sage.rings.complex_arb`

Element

alias of `RealBall`

`algebraic_closure()`

Return the complex ball field with the same precision.

EXAMPLES:

```
sage: from sage.rings.complex_arb import ComplexBallField
sage: RBF.complex_field()
Complex ball field with 53 bits of precision
sage: RealBallField(3).algebraic_closure()
Complex ball field with 3 bits of precision
```

```
>>> from sage.all import *
>>> from sage.rings.complex_arb import ComplexBallField
>>> RBF.complex_field()
Complex ball field with 53 bits of precision
>>> RealBallField(Integer(3)).algebraic_closure()
Complex ball field with 3 bits of precision
```

`bell_number(n)`

Return a ball enclosing the n -th Bell number.

EXAMPLES:

```
sage: [RBF.bell_number(n) for n in range(7)]
[1.0000000000000000,
 1.0000000000000000,
```

(continues on next page)

(continued from previous page)

```

2.0000000000000000,
5.0000000000000000,
15.000000000000000,
52.000000000000000,
203.00000000000000]
sage: RBF.bell_number(-1)
Traceback (most recent call last):
...
ValueError: expected a nonnegative index
sage: RBF.bell_number(10**20)
[5.38270113176282e+1794956117137290721328 +/- ...e+1794956117137290721313]

```

```

>>> from sage.all import *
>>> [RBF.bell_number(n) for n in range(Integer(7))]
[1.0000000000000000,
1.0000000000000000,
2.0000000000000000,
5.0000000000000000,
15.000000000000000,
52.000000000000000,
203.00000000000000]
>>> RBF.bell_number(-Integer(1))
Traceback (most recent call last):
...
ValueError: expected a nonnegative index
>>> RBF.bell_number(Integer(10)**Integer(20))
[5.38270113176282e+1794956117137290721328 +/- ...e+1794956117137290721313]

```

bernoulli(*n*)

Return a ball enclosing the *n*-th Bernoulli number.

EXAMPLES:

```

sage: [RBF.bernoulli(n) for n in range(4)]
[1.0000000000000000, -0.5000000000000000, [0.1666666666666667 +/- ...e-17], 0]
sage: RBF.bernoulli(2**20)
[-1.823002872104961e+5020717 +/- ...e+5020701]
sage: RBF.bernoulli(2**1000)
Traceback (most recent call last):
...
ValueError: argument too large

```

```

>>> from sage.all import *
>>> [RBF.bernoulli(n) for n in range(Integer(4))]
[1.0000000000000000, -0.5000000000000000, [0.1666666666666667 +/- ...e-17], 0]
>>> RBF.bernoulli(Integer(2)**Integer(20))
[-1.823002872104961e+5020717 +/- ...e+5020701]
>>> RBF.bernoulli(Integer(2)**Integer(1000))
Traceback (most recent call last):
...
ValueError: argument too large

```

catalan_constant()

Return a ball enclosing the Catalan constant.

EXAMPLES:

```
sage: RBF.catalan_constant()
[0.915965594177219 +/- ...e-16]
sage: RealBallField(128).catalan_constant()
[0.91596559417721901505460351493238411077 +/- ...e-39]
```

```
>>> from sage.all import *
>>> RBF.catalan_constant()
[0.915965594177219 +/- ...e-16]
>>> RealBallField(Integer(128)).catalan_constant()
[0.91596559417721901505460351493238411077 +/- ...e-39]
```

characteristic()

Real ball fields have characteristic zero.

EXAMPLES:

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

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

complex_field()

Return the complex ball field with the same precision.

EXAMPLES:

```
sage: from sage.rings.complex_arb import ComplexBallField
sage: RBF.complex_field()
Complex ball field with 53 bits of precision
sage: RealBallField(3).algebraic_closure()
Complex ball field with 3 bits of precision
```

```
>>> from sage.all import *
>>> from sage.rings.complex_arb import ComplexBallField
>>> RBF.complex_field()
Complex ball field with 53 bits of precision
>>> RealBallField(Integer(3)).algebraic_closure()
Complex ball field with 3 bits of precision
```

construction()

Return the construction of a real ball field as a completion of the rationals.

EXAMPLES:

```
sage: RBF = RealBallField(42)
sage: functor, base = RBF.construction()
sage: functor, base
```

(continues on next page)

(continued from previous page)

```
(Completion[+Infinity, prec=42], Rational Field)
sage: functor(base) is RBF
True
```

```
>>> from sage.all import *
>>> RBF = RealBallField(Integer(42))
>>> functor, base = RBF.construction()
>>> functor, base
(Completion[+Infinity, prec=42], Rational Field)
>>> functor(base) is RBF
True
```

cospi(x)

Return a ball enclosing $\cos(\pi x)$.

This works even if x itself is not a ball, and may be faster or more accurate where x is a rational number.

EXAMPLES:

```
sage: RBF.cospi(1)
-1.0000000000000000
sage: RBF.cospi(1/3)
0.5000000000000000
```

```
>>> from sage.all import *
>>> RBF.cospi(Integer(1))
-1.0000000000000000
>>> RBF.cospi(Integer(1)/Integer(3))
0.5000000000000000
```

See also

[`cos\(\)`](#)

double_factorial(n)

Return a ball enclosing the n -th double factorial.

EXAMPLES:

```
sage: [RBF.double_factorial(n) for n in range(7)]
[1.0000000000000000,
 1.0000000000000000,
 2.0000000000000000,
 3.0000000000000000,
 8.0000000000000000,
15.0000000000000000,
48.0000000000000000]
sage: RBF.double_factorial(2**20)
[1.448372990...e+2928836 +/- ...]
sage: RBF.double_factorial(2**1000)
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```

ValueError: argument too large
sage: RBF.double_factorial(-1)
Traceback (most recent call last):
...
ValueError: expected a nonnegative index

```

```

>>> from sage.all import *
>>> [RBF.double_factorial(n) for n in range(Integer(7))]
[1.0000000000000000,
 1.0000000000000000,
 2.0000000000000000,
 3.0000000000000000,
 8.0000000000000000,
15.0000000000000000,
48.0000000000000000]
>>> RBF.double_factorial(Integer(2)**Integer(20))
[1.448372990...e+2928836 +/- ...]
>>> RBF.double_factorial(Integer(2)**Integer(1000))
Traceback (most recent call last):
...
ValueError: argument too large
>>> RBF.double_factorial(-Integer(1))
Traceback (most recent call last):
...
ValueError: expected a nonnegative index

```

euler_constant()

Return a ball enclosing the Euler constant.

EXAMPLES:

```

sage: RBF.euler_constant() # abs tol 1e-15
[0.5772156649015329 +/- 9.00e-17]
sage: RealBallField(128).euler_constant()
[0.57721566490153286060651209008240243104 +/- ...e-39]

```

```

>>> from sage.all import *
>>> RBF.euler_constant() # abs tol 1e-15
[0.5772156649015329 +/- 9.00e-17]
>>> RealBallField(Integer(128)).euler_constant()
[0.57721566490153286060651209008240243104 +/- ...e-39]

```

fibonacci(n)

Return a ball enclosing the n -th Fibonacci number.

EXAMPLES:

```

sage: [RBF.fibonacci(n) for n in range(7)]
[0,
 1.0000000000000000,
 1.0000000000000000,
 2.0000000000000000,

```

(continues on next page)

(continued from previous page)

```

3.0000000000000000,
5.0000000000000000,
8.0000000000000000]
sage: RBF.fibonacci(-2)
-1.0000000000000000
sage: RBF.fibonacci(10**20)
[3.78202087472056e+20898764024997873376 +/- ...e+20898764024997873361]

```

```

>>> from sage.all import *
>>> [RBF.fibonacci(n) for n in range(Integer(7))]
[0,
1.0000000000000000,
1.0000000000000000,
2.0000000000000000,
3.0000000000000000,
5.0000000000000000,
8.0000000000000000]
>>> RBF.fibonacci(-Integer(2))
-1.0000000000000000
>>> RBF.fibonacci(Integer(10)**Integer(20))
[3.78202087472056e+20898764024997873376 +/- ...e+20898764024997873361]

```

gamma (x)

Return a ball enclosing the gamma function of x .

This works even if x itself is not a ball, and may be more efficient in the case where x is an integer or a rational number.

EXAMPLES:

```

sage: RBF.gamma(5)
24.000000000000000
sage: RBF.gamma(10**20)
[1.932849514310098...+1956570551809674817225 +/- ...]
sage: RBF.gamma(1/3)
[2.678938534707747 +/- ...e-16]
sage: RBF.gamma(-5)
nan

```

```

>>> from sage.all import *
>>> RBF.gamma(Integer(5))
24.000000000000000
>>> RBF.gamma(Integer(10)**Integer(20))
[1.932849514310098...+1956570551809674817225 +/- ...]
>>> RBF.gamma(Integer(1)/Integer(3))
[2.678938534707747 +/- ...e-16]
>>> RBF.gamma(-Integer(5))
nan

```

See also

[*gamma\(\)*](#)

gens()

EXAMPLES:

```
sage: RBF.gens()
(1.0000000000000000,)
sage: RBF.gens_dict()
{'1.0000000000000000': 1.0000000000000000}
```

```
>>> from sage.all import *
>>> RBF.gens()
(1.0000000000000000,)
>>> RBF.gens_dict()
{'1.0000000000000000': 1.0000000000000000}
```

is_exact()

Real ball fields are not exact.

EXAMPLES:

```
sage: RealBallField().is_exact()
False
```

```
>>> from sage.all import *
>>> RealBallField().is_exact()
False
```

log2()Return a ball enclosing $\log(2)$.

EXAMPLES:

```
sage: RBF.log2()
[0.6931471805599453 +/- ...e-17]
sage: RealBallField(128).log2()
[0.69314718055994530941723212145817656807 +/- ...e-39]
```

```
>>> from sage.all import *
>>> RBF.log2()
[0.6931471805599453 +/- ...e-17]
>>> RealBallField(Integer(128)).log2()
[0.69314718055994530941723212145817656807 +/- ...e-39]
```

maximal_accuracy()

Return the relative accuracy of exact elements measured in bits.

OUTPUT: integer

EXAMPLES:

```
sage: RBF.maximal_accuracy()
9223372036854775807 # 64-bit
2147483647          # 32-bit
```

```
>>> from sage.all import *
>>> RBF.maximal_accuracy()
9223372036854775807 # 64-bit
2147483647          # 32-bit
```

See also

RealBall.accuracy()

pi()

Return a ball enclosing π .

EXAMPLES:

```
sage: RBF.pi()
[3.141592653589793 +/- ...e-16]
sage: RealBallField(128).pi()
[3.1415926535897932384626433832795028842 +/- ...e-38]
```

```
>>> from sage.all import *
>>> RBF.pi()
[3.141592653589793 +/- ...e-16]
>>> RealBallField(Integer(128)).pi()
[3.1415926535897932384626433832795028842 +/- ...e-38]
```

prec()

Return the bit precision used for operations on elements of this field.

EXAMPLES:

```
sage: RealBallField().precision()
53
```

```
>>> from sage.all import *
>>> RealBallField().precision()
53
```

precision()

Return the bit precision used for operations on elements of this field.

EXAMPLES:

```
sage: RealBallField().precision()
53
```

```
>>> from sage.all import *
>>> RealBallField().precision()
53
```

sinpi(x)

Return a ball enclosing $\sin(\pi x)$.

This works even if x itself is not a ball, and may be faster or more accurate where x is a rational number.

EXAMPLES:

```
sage: RBF.sinpi(1)
0
sage: RBF.sinpi(1/3)
[0.866025403784439 +/- ...e-16]
sage: RBF.sinpi(1 + 2^(-100))
[-2.478279624546525e-30 +/- ...e-46]
```

```
>>> from sage.all import *
>>> RBF.sinpi(Integer(1))
0
>>> RBF.sinpi(Integer(1)/Integer(3))
[0.866025403784439 +/- ...e-16]
>>> RBF.sinpi(Integer(1) + Integer(2)**(-Integer(100)))
[-2.478279624546525e-30 +/- ...e-46]
```

See also

[`sin\(\)`](#)

some_elements()

Real ball fields contain exact balls, inexact balls, infinities, and more.

EXAMPLES:

```
sage: RBF.some_elements()
[0, 1.000000000000000, [0.3333333333333333 +/- ...e-17],
[-4.733045976388941e+363922934236666733021124 +/- ...
↪e+363922934236666733021108],
[+/- inf], [+/- inf], [+/- inf], nan]
```

```
>>> from sage.all import *
>>> RBF.some_elements()
[0, 1.000000000000000, [0.3333333333333333 +/- ...e-17],
[-4.733045976388941e+363922934236666733021124 +/- ...
↪e+363922934236666733021108],
[+/- inf], [+/- inf], [+/- inf], nan]
```

zeta(s)

Return a ball enclosing the Riemann zeta function of s .

This works even if s itself is not a ball, and may be more efficient in the case where s is an integer.

EXAMPLES:

```
sage: RBF.zeta(3)
[1.202056903159594 +/- ...e-16]
sage: RBF.zeta(1)
nan
sage: RBF.zeta(1/2)
[-1.460354508809587 +/- ...e-16]
```



```
>>> from sage.all import *
>>> RBF.zeta(Integer(3))
[1.202056903159594 +/- ...e-16]
>>> RBF.zeta(Integer(1))
nan
>>> RBF.zeta(Integer(1)/Integer(2))
[-1.460354508809587 +/- ...e-16]
```

See also[`zeta\(\)`](#)

`sage.rings.real_arb.create_RealBall` (*parent, serialized*)

Create a RealBall from a serialized representation.

2.6 Arbitrary precision complex balls

This is an incomplete interface to the [acb module of FLINT](#); it may be useful to refer to its documentation for more details.

Parts of the documentation for this module are copied or adapted from Arb's (now FLINT's) own documentation, licenced at the time under the GNU General Public License version 2, or later.

See also

- [Real balls](#)
- [Complex interval field \(using MPFI\)](#)
- [Complex intervals \(using MPFI\)](#)

2.6.1 Data Structure

A [ComplexBall](#) represents a complex number with error bounds. It wraps an object of type `acb_t`, which consists of a pair of real number balls representing the real and imaginary part with separate error bounds. (See the documentation of `sage.rings.real_arb` for more information.)

A [ComplexBall](#) thus represents a rectangle $[m_1 - r_1, m_1 + r_1] + [m_2 - r_2, m_2 + r_2]i$ in the complex plane. This is used instead of a disk or square representation (consisting of a complex floating-point midpoint with a single radius), since it allows implementing many operations more conveniently by splitting into ball operations on the real and imaginary parts. It also allows tracking when complex numbers have an exact (for example exactly zero) real part and an inexact imaginary part, or vice versa.

The parents of complex balls are instances of [ComplexBallField](#). The name CBF is bound to the complex ball field with the default precision of 53 bits:

```
sage: CBF is ComplexBallField() is ComplexBallField(53)
True
```

```
>>> from sage.all import *
>>> CBF is ComplexBallField() is ComplexBallField(Integer(53))
True
```

2.6.2 Comparison

Warning

In accordance with the semantics of FLINT/Arb, identical `ComplexBall` objects are understood to give permission for algebraic simplification. This assumption is made to improve performance. For example, setting `z = x*x` sets `z` to a ball enclosing the set $\{t^2 : t \in x\}$ and not the (generally larger) set $\{tu : t \in x, u \in x\}$.

Two elements are equal if and only if they are exact and equal (in spite of the above warning, inexact balls are not considered equal to themselves):

```
sage: a = CBF(1, 2)
sage: b = CBF(1, 2)
sage: a is b
False
sage: a == a
True
sage: a == b
True
```

```
>>> from sage.all import *
>>> a = CBF(Integer(1), Integer(2))
>>> b = CBF(Integer(1), Integer(2))
>>> a is b
False
>>> a == a
True
>>> a == b
True
```

```
sage: a = CBF(1/3, 1/5)
sage: b = CBF(1/3, 1/5)
sage: a.is_exact()
False
sage: b.is_exact()
False
sage: a is b
False
sage: a == a
False
sage: a == b
False
```

```
>>> from sage.all import *
>>> a = CBF(Integer(1)/Integer(3), Integer(1)/Integer(5))
>>> b = CBF(Integer(1)/Integer(3), Integer(1)/Integer(5))
>>> a.is_exact()
False
>>> b.is_exact()
False
>>> a is b
```

(continues on next page)

(continued from previous page)

```
False
>>> a == a
False
>>> a == b
False
```

A ball is nonzero in the sense of usual comparison if and only if it does not contain zero:

```
sage: a = CBF(RIF(-0.5, 0.5))
sage: a != 0
False
sage: b = CBF(1/3, 1/5)
sage: b != 0
True
```

```
>>> from sage.all import *
>>> a = CBF(RIF(-RealNumber('0.5'), RealNumber('0.5')))
>>> a != Integer(0)
False
>>> b = CBF(Integer(1)/Integer(3), Integer(1)/Integer(5))
>>> b != Integer(0)
True
```

However, `bool(b)` returns `False` for a ball `b` only if `b` is exactly zero:

```
sage: bool(a)
True
sage: bool(b)
True
sage: bool(CBF.zero())
False
```

```
>>> from sage.all import *
>>> bool(a)
True
>>> bool(b)
True
>>> bool(CBF.zero())
False
```

2.6.3 Coercion

Automatic coercions work as expected:

```
sage: # needs sage.symbolic
sage: bpol = 1/3*CBF(i) + AA(sqrt(2))
sage: bpol += polygen(RealBallField(20), 'x') + QQbar(i)
sage: bpol
x + [1.41421 +/- ...e-6] + [1.33333 +/- ...e-6]*I
sage: bpol.parent()
Univariate Polynomial Ring in x over Complex ball field with 20 bits of precision
```

(continues on next page)

(continued from previous page)

```
sage: bpol/3
([0.333333 +/- ...e-7])*x + [0.47140 +/- ...e-6] + [0.44444 +/- ...e-6]*I
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> bpol = Integer(1)/Integer(3)*CBF(i) + AA(sqrt(Integer(2)))
>>> bpol += polygen(RealBallField(Integer(20)), 'x') + QQbar(i)
>>> bpol
x + [1.41421 +/- ...e-6] + [1.33333 +/- ...e-6]*I
>>> bpol.parent()
Univariate Polynomial Ring in x over Complex ball field with 20 bits of precision
>>> bpol/Integer(3)
([0.333333 +/- ...e-7])*x + [0.47140 +/- ...e-6] + [0.44444 +/- ...e-6]*I
```

2.6.4 Classes and Methods

class sage.rings.complex_arb.ComplexBall

Bases: RingElement

Hold one acb_t of the FLINT library.

EXAMPLES:

```
sage: a = ComplexBallField()(1, 1)
sage: a
1.0000000000000000 + 1.0000000000000000*I
```

```
>>> from sage.all import *
>>> a = ComplexBallField()(Integer(1), Integer(1))
>>> a
1.0000000000000000 + 1.0000000000000000*I
```

Chi()

Return the hyperbolic cosine integral with argument self.

EXAMPLES:

```
sage: CBF(1, 1).Chi() # abs tol 1e-15
[0.882172180555936 +/- 5.89e-16] + [1.28354719327494 +/- 1.01e-15]*I
sage: CBF(0).Chi()
nan + nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).Chi() # abs tol 1e-15
[0.882172180555936 +/- 5.89e-16] + [1.28354719327494 +/- 1.01e-15]*I
>>> CBF(Integer(0)).Chi()
nan + nan*I
```

Ci()

Return the cosine integral with argument self.

EXAMPLES:

```
sage: CBF(1, 1).Ci() # abs tol 5e-16
[0.882172180555936 +/- 5.89e-16] + [0.287249133519956 +/- 3.37e-16]*I
sage: CBF(0).Ci()
nan + nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).Ci() # abs tol 5e-16
[0.882172180555936 +/- 5.89e-16] + [0.287249133519956 +/- 3.37e-16]*I
>>> CBF(Integer(0)).Ci()
nan + nan*I
```

Ei()

Return the exponential integral with argument `self`.

EXAMPLES:

```
sage: CBF(1, 1).Ei() # abs tol 6e-15
[1.76462598556385 +/- 6.03e-15] + [2.38776985151052 +/- 4.23e-15]*I
sage: CBF(0).Ei()
nan...
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).Ei() # abs tol 6e-15
[1.76462598556385 +/- 6.03e-15] + [2.38776985151052 +/- 4.23e-15]*I
>>> CBF(Integer(0)).Ei()
nan...
```

Li()

Offset logarithmic integral.

EXAMPLES:

```
sage: CBF(0).Li()
[-1.045163780117493 +/- ...e-16]
sage: li(0).n() #_
↳needs sage.symbolic
0.0000000000000000
sage: Li(0).n() #_
↳needs sage.symbolic
-1.04516378011749
```

```
>>> from sage.all import *
>>> CBF(Integer(0)).Li()
[-1.045163780117493 +/- ...e-16]
>>> li(Integer(0)).n() _
↳ # needs sage.symbolic
0.0000000000000000
>>> Li(Integer(0)).n() _
↳ # needs sage.symbolic
-1.04516378011749
```

Shi()

Return the hyperbolic sine integral with argument `self`.

EXAMPLES:

```
sage: CBF(1, 1).Shi() # abs tol 3e-15
[0.88245380500792 +/- 3.36e-15] + [1.10422265823558 +/- 2.48e-15]*I
sage: CBF(0).Shi()
0
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).Shi() # abs tol 3e-15
[0.88245380500792 +/- 3.36e-15] + [1.10422265823558 +/- 2.48e-15]*I
>>> CBF(Integer(0)).Shi()
0
```

Si()

Return the sine integral with argument `self`.

EXAMPLES:

```
sage: CBF(1, 1).Si() # abs tol 3e-15
[1.10422265823558 +/- 2.48e-15] + [0.88245380500792 +/- 3.36e-15]*I
sage: CBF(0).Si()
0
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).Si() # abs tol 3e-15
[1.10422265823558 +/- 2.48e-15] + [0.88245380500792 +/- 3.36e-15]*I
>>> CBF(Integer(0)).Si()
0
```

above_abs()

Return an upper bound for the absolute value of this complex ball.

OUTPUT: a ball with zero radius

EXAMPLES:

```
sage: b = ComplexBallField(8)(1+i).above_abs()
sage: b
[1.4 +/- 0.0219]
sage: b.is_exact()
True
sage: QQ(b)*128
182
```

```
>>> from sage.all import *
>>> b = ComplexBallField(Integer(8))(Integer(1)+i).above_abs()
>>> b
[1.4 +/- 0.0219]
>>> b.is_exact()
True
>>> QQ(b)*Integer(128)
182
```

See also`below_abs()`**accuracy()**

Return the effective relative accuracy of this ball measured in bits.

This is computed as if calling `accuracy()` on the real ball whose midpoint is the larger out of the real and imaginary midpoints of this complex ball, and whose radius is the larger out of the real and imaginary radii of this complex ball.

EXAMPLES:

```
sage: CBF(exp(I*pi/3)).accuracy() #_
↪needs sage.symbolic
51
sage: CBF(I/2).accuracy() == CBF.base().maximal_accuracy()
True
sage: CBF('nan', 'inf').accuracy() == -CBF.base().maximal_accuracy()
True
```

```
>>> from sage.all import *
>>> CBF(exp(I*pi/Integer(3))).accuracy() #_
↪      # needs sage.symbolic
51
>>> CBF(I/Integer(2)).accuracy() == CBF.base().maximal_accuracy()
True
>>> CBF('nan', 'inf').accuracy() == -CBF.base().maximal_accuracy()
True
```

See also`maximal_accuracy()`**add_error(ampl)**

Increase the radii of the real and imaginary parts by (an upper bound on) `ampl`.

If `ampl` is negative, the radii remain unchanged.

INPUT:

- `ampl` – a **real** ball (or an object that can be coerced to a real ball)

OUTPUT: a new complex ball

EXAMPLES:

```
sage: CBF(1+i).add_error(10^-16)
[1.0000000000000000 +/- ...e-16] + [1.0000000000000000 +/- ...e-16]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1)+i).add_error(Integer(10)**-Integer(16))
[1.0000000000000000 +/- ...e-16] + [1.0000000000000000 +/- ...e-16]*I
```

agm1()

Return the arithmetic-geometric mean of 1 and `self`.

The arithmetic-geometric mean is defined such that the function is continuous in the complex plane except for a branch cut along the negative half axis (where it is continuous from above). This corresponds to always choosing an “optimal” branch for the square root in the arithmetic-geometric mean iteration.

EXAMPLES:

```
sage: CBF(0, -1).agm1()
[0.599070117367796 +/- 3.9...e-16] + [-0.599070117367796 +/- 5.5...e-16]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(0), -Integer(1)).agm1()
[0.599070117367796 +/- 3.9...e-16] + [-0.599070117367796 +/- 5.5...e-16]*I
```

airy()

Return the Airy functions A_i , A_i' , B_i , B_i' with argument `self`, evaluated simultaneously.

EXAMPLES:

```
sage: CBF(10*pi).airy() #_
↳needs sage.symbolic
([1.2408955946101e-52 +/- ...e-66],
 [-6.965048886977e-52 +/- ...e-65],
 [2.2882956833435e+50 +/- ...e+36],
 [1.2807602335816e+51 +/- ...e+37])
sage: ai, aip, bi, bip = CBF(1,2).airy()
sage: (ai * bip - bi * aip) * CBF(pi) #_
↳needs sage.symbolic
[1.00000000000000 +/- ...e-15] + [+/- ...e-16]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(10)*pi).airy() _
↳ # needs sage.symbolic
([1.2408955946101e-52 +/- ...e-66],
 [-6.965048886977e-52 +/- ...e-65],
 [2.2882956833435e+50 +/- ...e+36],
 [1.2807602335816e+51 +/- ...e+37])
>>> ai, aip, bi, bip = CBF(Integer(1),Integer(2)).airy()
>>> (ai * bip - bi * aip) * CBF(pi) #_
↳needs sage.symbolic
[1.00000000000000 +/- ...e-15] + [+/- ...e-16]*I
```

airy_ai()

Return the Airy function A_i with argument `self`.

EXAMPLES:

```
sage: CBF(1,2).airy_ai()
[-0.2193862549814276 +/- ...e-17] + [-0.1753859114081094 +/- ...e-17]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1),Integer(2)).airy_ai()
[-0.2193862549814276 +/- ...e-17] + [-0.1753859114081094 +/- ...e-17]*I
```


airy_ai_prime()

Return the Airy function derivative Ai' with argument `self`.

EXAMPLES:

```
sage: CBF(1,2).airy_ai_prime()
[0.1704449781789148 +/- ...e-17] + [0.387622439413295 +/- ...e-16]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(2)).airy_ai_prime()
[0.1704449781789148 +/- ...e-17] + [0.387622439413295 +/- ...e-16]*I
```

airy_bi()

Return the Airy function Bi with argument `self`.

EXAMPLES:

```
sage: CBF(1,2).airy_bi()
[0.0488220324530612 +/- ...e-17] + [0.1332740579917484 +/- ...e-17]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(2)).airy_bi()
[0.0488220324530612 +/- ...e-17] + [0.1332740579917484 +/- ...e-17]*I
```

airy_bi_prime()

Return the Airy function derivative Bi' with argument `self`.

EXAMPLES:

```
sage: CBF(1,2).airy_bi_prime()
[-0.857239258605362 +/- ...e-16] + [0.4955063363095674 +/- ...e-17]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(2)).airy_bi_prime()
[-0.857239258605362 +/- ...e-16] + [0.4955063363095674 +/- ...e-17]*I
```

arccos (analytic=False)

Return the arccosine of this ball.

INPUT:

- `analytic` – boolean (default: `False`); if `True`, return an indeterminate (not-a-number) value when the input ball touches the branch cut

EXAMPLES:

```
sage: CBF(1+i).arccos()
[0.90455689430238 +/- ...e-15] + [-1.06127506190504 +/- ...e-15]*I
sage: CBF(-1).arccos()
[3.141592653589793 +/- ...e-16]
sage: CBF(-1).arccos(analytic=True)
nan + nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1)+i).arccos()
```

(continues on next page)

(continued from previous page)

```
[0.90455689430238 +/- ...e-15] + [-1.06127506190504 +/- ...e-15]*I
>>> CBF(-Integer(1)).arccos()
[3.141592653589793 +/- ...e-16]
>>> CBF(-Integer(1)).arccos(analytic=True)
nan + nan*I
```

arccosh (*analytic=False*)

Return the hyperbolic arccosine of this ball.

INPUT:

- *analytic* – boolean (default: False); if True, return an indeterminate (not-a-number) value when the input ball touches the branch cut

EXAMPLES:

```
sage: CBF(1+i).arccosh()
[1.061275061905035 +/- ...e-16] + [0.904556894302381 +/- ...e-16]*I
sage: CBF(-2).arccosh()
[1.316957896924817 +/- ...e-16] + [3.141592653589793 +/- ...e-16]*I
sage: CBF(-2).arccosh(analytic=True)
nan + nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1)+i).arccosh()
[1.061275061905035 +/- ...e-16] + [0.904556894302381 +/- ...e-16]*I
>>> CBF(-Integer(2)).arccosh()
[1.316957896924817 +/- ...e-16] + [3.141592653589793 +/- ...e-16]*I
>>> CBF(-Integer(2)).arccosh(analytic=True)
nan + nan*I
```

arcsin (*analytic=False*)

Return the arcsine of this ball.

INPUT:

- *analytic* – boolean (default: False); if True, return an indeterminate (not-a-number) value when the input ball touches the branch cut

EXAMPLES:

```
sage: CBF(1+i).arcsin()
[0.66623943249252 +/- ...e-15] + [1.06127506190504 +/- ...e-15]*I
sage: CBF(1, RIF(0, 1/1000)).arcsin()
[1.6 +/- 0.0619] + [+/- 0.0322]*I
sage: CBF(1, RIF(0, 1/1000)).arcsin(analytic=True)
nan + nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1)+i).arcsin()
[0.66623943249252 +/- ...e-15] + [1.06127506190504 +/- ...e-15]*I
>>> CBF(Integer(1), RIF(Integer(0), Integer(1)/Integer(1000))).arcsin()
[1.6 +/- 0.0619] + [+/- 0.0322]*I
>>> CBF(Integer(1), RIF(Integer(0), Integer(1)/Integer(1000))).
```

(continues on next page)

(continued from previous page)

```
↪ arcsin(analytic=True)
nan + nan*I
```

arcsinh (*analytic=False*)

Return the hyperbolic arcsine of this ball.

INPUT:

- *analytic* – boolean (default: False); if True, return an indeterminate (not-a-number) value when the input ball touches the branch cut

EXAMPLES:

```
sage: CBF(1+i).arcsinh()
[1.06127506190504 +/- ...e-15] + [0.66623943249252 +/- ...e-15]*I
sage: CBF(2*i).arcsinh()
[1.31695789692482 +/- ...e-15] + [1.570796326794897 +/- ...e-16]*I
sage: CBF(2*i).arcsinh(analytic=True)
nan + nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1)+i).arcsinh()
[1.06127506190504 +/- ...e-15] + [0.66623943249252 +/- ...e-15]*I
>>> CBF(Integer(2)*i).arcsinh()
[1.31695789692482 +/- ...e-15] + [1.570796326794897 +/- ...e-16]*I
>>> CBF(Integer(2)*i).arcsinh(analytic=True)
nan + nan*I
```

arctan (*analytic=False*)

Return the arctangent of this ball.

INPUT:

- *analytic* – boolean (default: False); if True, return an indeterminate (not-a-number) value when the input ball touches the branch cut

EXAMPLES:

```
sage: CBF(1+i).arctan()
[1.017221967897851 +/- ...e-16] + [0.4023594781085251 +/- ...e-17]*I
sage: CBF(i).arctan()
nan + nan*I
sage: CBF(2*i).arctan()
[1.570796326794897 +/- ...e-16] + [0.549306144334055 +/- ...e-16]*I
sage: CBF(2*i).arctan(analytic=True)
nan + nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1)+i).arctan()
[1.017221967897851 +/- ...e-16] + [0.4023594781085251 +/- ...e-17]*I
>>> CBF(i).arctan()
nan + nan*I
>>> CBF(Integer(2)*i).arctan()
[1.570796326794897 +/- ...e-16] + [0.549306144334055 +/- ...e-16]*I
```

(continues on next page)

(continued from previous page)

```
>>> CBF(Integer(2)*i).arctan(analytic=True)
nan + nan*I
```

arctanh (*analytic=False*)

Return the hyperbolic arctangent of this ball.

INPUT:

- *analytic* – boolean (default: False); if True, return an indeterminate (not-a-number) value when the input ball touches the branch cut

EXAMPLES:

```
sage: CBF(1+i).arctanh()
[0.4023594781085251 +/- ...e-17] + [1.017221967897851 +/- ...e-16]*I
sage: CBF(-2).arctanh()
[-0.549306144334055 +/- ...e-16] + [1.570796326794897 +/- ...e-16]*I
sage: CBF(-2).arctanh(analytic=True)
nan + nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1)+i).arctanh()
[0.4023594781085251 +/- ...e-17] + [1.017221967897851 +/- ...e-16]*I
>>> CBF(-Integer(2)).arctanh()
[-0.549306144334055 +/- ...e-16] + [1.570796326794897 +/- ...e-16]*I
>>> CBF(-Integer(2)).arctanh(analytic=True)
nan + nan*I
```

arg ()

Return the argument of this complex ball.

EXAMPLES:

```
sage: CBF(1 + i).arg()
[0.7853981633974483 +/- ...e-17]
sage: CBF(-1).arg()
[3.141592653589793 +/- ...e-16]
sage: CBF(-1).arg().parent()
Real ball field with 53 bits of precision
```

```
>>> from sage.all import *
>>> CBF(Integer(1) + i).arg()
[0.7853981633974483 +/- ...e-17]
>>> CBF(-Integer(1)).arg()
[3.141592653589793 +/- ...e-16]
>>> CBF(-Integer(1)).arg().parent()
Real ball field with 53 bits of precision
```

barnes_g ()Return the Barnes G-function of *self*.

EXAMPLES:

```
sage: CBF(-4).barnes_g()
0
sage: CBF(8).barnes_g()
24883200.00000000
sage: CBF(500,10).barnes_g()
[4.54078781e+254873 +/- ...e+254864] + [8.65835455e+254873 +/- ...e+254864]*I
```

```
>>> from sage.all import *
>>> CBF(-Integer(4)).barnes_g()
0
>>> CBF(Integer(8)).barnes_g()
24883200.00000000
>>> CBF(Integer(500), Integer(10)).barnes_g()
[4.54078781e+254873 +/- ...e+254864] + [8.65835455e+254873 +/- ...e+254864]*I
```

below_abs (*test_zero=False*)

Return a lower bound for the absolute value of this complex ball.

INPUT:

- *test_zero* – boolean (default: False); if True, make sure that the returned lower bound is positive, raising an error if the ball contains zero.

OUTPUT: a ball with zero radius

EXAMPLES:

```
sage: b = ComplexBallField(8)(1+i).below_abs()
sage: b
[1.4 +/- 0.0141]
sage: b.is_exact()
True
sage: QQ(b)*128
181
sage: (CBF(1/3) - 1/3).below_abs()
0
sage: (CBF(1/3) - 1/3).below_abs(test_zero=True)
Traceback (most recent call last):
...
ValueError: ball contains zero
```

```
>>> from sage.all import *
>>> b = ComplexBallField(Integer(8))(Integer(1)+i).below_abs()
>>> b
[1.4 +/- 0.0141]
>>> b.is_exact()
True
>>> QQ(b)*Integer(128)
181
>>> (CBF(Integer(1)/Integer(3)) - Integer(1)/Integer(3)).below_abs()
0
>>> (CBF(Integer(1)/Integer(3)) - Integer(1)/Integer(3)).below_abs(test_
↪ zero=True)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: ball contains zero
```

See also

[`above\_abs\(\)`](#)

bessel_I(nu)

Return the modified Bessel function of the first kind with argument `self` and index `nu`.

EXAMPLES:

```
sage: CBF(1, 1).bessel_I(1)
[0.365028028827088 +/- ...e-16] + [0.614160334922903 +/- ...e-16]*I
sage: CBF(100, -100).bessel_I(1/3)
[5.4362189595644e+41 +/- ...e+27] + [7.1989436985321e+41 +/- ...e+27]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).bessel_I(Integer(1))
[0.365028028827088 +/- ...e-16] + [0.614160334922903 +/- ...e-16]*I
>>> CBF(Integer(100), -Integer(100)).bessel_I(Integer(1)/Integer(3))
[5.4362189595644e+41 +/- ...e+27] + [7.1989436985321e+41 +/- ...e+27]*I
```

bessel_J(nu)

Return the Bessel function of the first kind with argument `self` and index `nu`.

EXAMPLES:

```
sage: CBF(1, 1).bessel_J(1)
[0.614160334922903 +/- ...e-16] + [0.365028028827088 +/- ...e-16]*I
sage: CBF(100, -100).bessel_J(1/3)
[1.108431870251e+41 +/- ...e+28] + [-8.952577603125e+41 +/- ...e+28]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).bessel_J(Integer(1))
[0.614160334922903 +/- ...e-16] + [0.365028028827088 +/- ...e-16]*I
>>> CBF(Integer(100), -Integer(100)).bessel_J(Integer(1)/Integer(3))
[1.108431870251e+41 +/- ...e+28] + [-8.952577603125e+41 +/- ...e+28]*I
```

bessel_J_Y(nu)

Return the Bessel function of the first and second kind with argument `self` and index `nu`, computed simultaneously.

EXAMPLES:

```
sage: J, Y = CBF(1, 1).bessel_J_Y(1)
sage: J - CBF(1, 1).bessel_J(1)
[+/- ...e-16] + [+/- ...e-16]*I
sage: Y - CBF(1, 1).bessel_Y(1)
[+/- ...e-14] + [+/- ...e-14]*I
```

```
>>> from sage.all import *
>>> J, Y = CBF(Integer(1), Integer(1)).bessel_J_Y(Integer(1))
>>> J - CBF(Integer(1), Integer(1)).bessel_J(Integer(1))
[+/- ...e-16] + [+/- ...e-16]*I
>>> Y - CBF(Integer(1), Integer(1)).bessel_Y(Integer(1))
[+/- ...e-14] + [+/- ...e-14]*I
```

bessel_K(*nu*)

Return the modified Bessel function of the second kind with argument *self* and index *nu*.

EXAMPLES:

```
sage: CBF(1, 1).bessel_K(0)
[0.08019772694652 +/- ...e-15] + [-0.357277459285330 +/- ...e-16]*I
sage: CBF(1, 1).bessel_K(1)
[0.02456830552374 +/- ...e-15] + [-0.45971947380119 +/- ...e-15]*I
sage: CBF(100, 100).bessel_K(QQbar(i))
[3.8693896656383e-45 +/- ...e-59] + [5.507100423418e-46 +/- ...e-59]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).bessel_K(Integer(0))
[0.08019772694652 +/- ...e-15] + [-0.357277459285330 +/- ...e-16]*I
>>> CBF(Integer(1), Integer(1)).bessel_K(Integer(1))
[0.02456830552374 +/- ...e-15] + [-0.45971947380119 +/- ...e-15]*I
>>> CBF(Integer(100), Integer(100)).bessel_K(QQbar(i))
[3.8693896656383e-45 +/- ...e-59] + [5.507100423418e-46 +/- ...e-59]*I
```

bessel_Y(*nu*)

Return the Bessel function of the second kind with argument *self* and index *nu*.

EXAMPLES:

```
sage: CBF(1, 1).bessel_Y(1)
[-0.6576945355913 +/- ...e-14] + [0.6298010039929 +/- ...e-14]*I
sage: CBF(100, -100).bessel_Y(1/3)
[-8.952577603125e+41 +/- ...e+28] + [-1.108431870251e+41 +/- ...e+28]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).bessel_Y(Integer(1))
[-0.6576945355913 +/- ...e-14] + [0.6298010039929 +/- ...e-14]*I
>>> CBF(Integer(100), -Integer(100)).bessel_Y(Integer(1)/Integer(3))
[-8.952577603125e+41 +/- ...e+28] + [-1.108431870251e+41 +/- ...e+28]*I
```

chebyshev_T(*n*)

Return the Chebyshev function of the first kind of order *n* evaluated at *self*.

EXAMPLES:

```
sage: CBF(1/3).chebyshev_T(20)
[0.8710045668809 +/- ...e-14]
sage: CBF(1/3).chebyshev_T(CBF(5, 1))
[1.84296854518763 +/- ...e-15] + [0.20053614301799 +/- ...e-15]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1)/Integer(3)).chebyshev_T(Integer(20))
[0.8710045668809 +/- ...e-14]
>>> CBF(Integer(1)/Integer(3)).chebyshev_T(CBF(Integer(5), Integer(1)))
[1.84296854518763 +/- ...e-15] + [0.20053614301799 +/- ...e-15]*I
```

chebyshev_U(*n*)

Return the Chebyshev function of the second kind of order *n* evaluated at *self*.

EXAMPLES:

```
sage: CBF(1/3).chebyshev_U(20)
[0.6973126541184 +/- ...e-14]
sage: CBF(1/3).chebyshev_U(CBF(5,1))
[1.75884964893425 +/- ...e-15] + [0.7497317165104 +/- ...e-14]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1)/Integer(3)).chebyshev_U(Integer(20))
[0.6973126541184 +/- ...e-14]
>>> CBF(Integer(1)/Integer(3)).chebyshev_U(CBF(Integer(5), Integer(1)))
[1.75884964893425 +/- ...e-15] + [0.7497317165104 +/- ...e-14]*I
```

conjugate()

Return the complex conjugate of this ball.

EXAMPLES:

```
sage: CBF(-2 + I/3).conjugate()
-2.0000000000000000 + [-0.3333333333333333 +/- ...e-17]*I
```

```
>>> from sage.all import *
>>> CBF(-Integer(2) + I/Integer(3)).conjugate()
-2.0000000000000000 + [-0.3333333333333333 +/- ...e-17]*I
```

contains_exact(*other*)

Return True *iff* *other* is contained in *self*.

Use *other* in *self* for a test that works for a wider range of inputs but may return false negatives.

INPUT:

- *other* – *ComplexBall*, *Integer*, or *Rational*

EXAMPLES:

```
sage: CBF(RealBallField(100)(1/3), 0).contains_exact(1/3)
True
sage: CBF(1).contains_exact(1)
True
sage: CBF(1).contains_exact(CBF(1))
True

sage: CBF(sqrt(2)).contains_exact(sqrt(2))
↪needs sage.symbolic
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: unsupported type: <class 'sage.symbolic.expression.Expression'>
```

```
>>> from sage.all import *
>>> CBF(RealBallField(Integer(100))(Integer(1)/Integer(3)), Integer(0)).
↳contains_exact(Integer(1)/Integer(3))
True
>>> CBF(Integer(1)).contains_exact(Integer(1))
True
>>> CBF(Integer(1)).contains_exact(CBF(Integer(1)))
True

>>> CBF(sqrt(Integer(2))).contains_exact(sqrt(Integer(2)))
↳# needs sage.symbolic
Traceback (most recent call last):
...
TypeError: unsupported type: <class 'sage.symbolic.expression.Expression'>
```

contains_integer()

Return True iff this ball contains any integer.

EXAMPLES:

```
sage: CBF(3, RBF(0.1)).contains_integer()
False
sage: CBF(3, RBF(0.1, 0.1)).contains_integer()
True
```

```
>>> from sage.all import *
>>> CBF(Integer(3), RBF(RealNumber('0.1'))).contains_integer()
False
>>> CBF(Integer(3), RBF(RealNumber('0.1'), RealNumber('0.1'))).contains_
↳integer()
True
```

contains_zero()

Return True iff this ball contains zero.

EXAMPLES:

```
sage: CBF(0).contains_zero()
True
sage: CBF(RIF(-1, 1)).contains_zero()
True
sage: CBF(i).contains_zero()
False
```

```
>>> from sage.all import *
>>> CBF(Integer(0)).contains_zero()
True
>>> CBF(RIF(-Integer(1), Integer(1))).contains_zero()
True
```

(continues on next page)

(continued from previous page)

```
>>> CBF(i).contains_zero()
False
```

cos()

Return the cosine of this ball.

EXAMPLES:

```
sage: CBF(i*pi).cos()
↪needs sage.symbolic
[11.59195327552152 +/- ...e-15]
```

```
>>> from sage.all import *
>>> CBF(i*pi).cos()
↪needs sage.symbolic
[11.59195327552152 +/- ...e-15]
```

cos_integral()

Return the cosine integral with argument self.

EXAMPLES:

```
sage: CBF(1, 1).Ci() # abs tol 5e-16
[0.882172180555936 +/- 5.89e-16] + [0.287249133519956 +/- 3.37e-16]*I
sage: CBF(0).Ci()
nan + nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).Ci() # abs tol 5e-16
[0.882172180555936 +/- 5.89e-16] + [0.287249133519956 +/- 3.37e-16]*I
>>> CBF(Integer(0)).Ci()
nan + nan*I
```

cosh()

Return the hyperbolic cosine of this ball.

EXAMPLES:

```
sage: CBF(1, 1).cosh()
[0.833730025131149 +/- ...e-16] + [0.988897705762865 +/- ...e-16]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).cosh()
[0.833730025131149 +/- ...e-16] + [0.988897705762865 +/- ...e-16]*I
```

cosh_integral()

Return the hyperbolic cosine integral with argument self.

EXAMPLES:

```
sage: CBF(1, 1).Chi() # abs tol 1e-15
[0.882172180555936 +/- 5.89e-16] + [1.28354719327494 +/- 1.01e-15]*I
sage: CBF(0).Chi()
nan + nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).Chi() # abs tol 1e-15
[0.882172180555936 +/- 5.89e-16] + [1.28354719327494 +/- 1.01e-15]*I
>>> CBF(Integer(0)).Chi()
nan + nan*I
```

cot()

Return the cotangent of this ball.

EXAMPLES:

```
sage: CBF(pi, 1/10).cot() #_
↪needs sage.symbolic
[+/- ...e-14] + [-10.03331113225399 +/- ...e-15]*I
sage: CBF(pi).cot() #_
↪needs sage.symbolic
nan
```

```
>>> from sage.all import *
>>> CBF(pi, Integer(1)/Integer(10)).cot() #_
↪ # needs sage.symbolic
[+/- ...e-14] + [-10.03331113225399 +/- ...e-15]*I
>>> CBF(pi).cot() #_
↪needs sage.symbolic
nan
```

coth()

Return the hyperbolic cotangent of this ball.

EXAMPLES:

```
sage: CBF(1, 1).coth() #_
[0.868014142895925 +/- ...e-16] + [-0.2176215618544027 +/- ...e-17]*I
sage: CBF(0, pi).coth() #_
↪needs sage.symbolic
nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).coth() #_
[0.868014142895925 +/- ...e-16] + [-0.2176215618544027 +/- ...e-17]*I
>>> CBF(Integer(0), pi).coth() #_
↪ # needs sage.symbolic
nan*I
```

csc()

Return the cosecant of this ball.

EXAMPLES:

```
sage: CBF(1, 1).csc()
[0.621518017170428 +/- ...e-16] + [-0.303931001628426 +/- ...e-16]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).csc()
[0.621518017170428 +/- ...e-16] + [-0.303931001628426 +/- ...e-16]*I
```

csch()

Return the hyperbolic cosecant of this ball.

EXAMPLES:

```
sage: CBF(1, 1).csch()
[0.303931001628426 +/- ...e-16] + [-0.621518017170428 +/- ...e-16]*I
sage: CBF(i*pi).csch()
↳needs sage.symbolic
nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).csch()
[0.303931001628426 +/- ...e-16] + [-0.621518017170428 +/- ...e-16]*I
>>> CBF(i*pi).csch()
↳needs sage.symbolic
nan*I
```

cube()

Return the cube of this ball.

The result is computed efficiently using two real squarings, two real multiplications, and scalar operations.

EXAMPLES:

```
sage: CBF(1, 1).cube()
-2.0000000000000000 + 2.0000000000000000*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).cube()
-2.0000000000000000 + 2.0000000000000000*I
```

diameter()

Return the diameter of this ball.

EXAMPLES:

```
sage: CBF(1 + i).diameter()
0.00000000
sage: CBF(i/3).diameter()
2.2204460e-16
sage: CBF(i/3).diameter().parent()
Real Field with 30 bits of precision
sage: CBF(CIF(RIF(1.02, 1.04), RIF(2.1, 2.2))).diameter()
0.20000000
```

```
>>> from sage.all import *
>>> CBF(Integer(1) + i).diameter()
0.00000000
>>> CBF(i/Integer(3)).diameter()
```

(continues on next page)

(continued from previous page)

```

2.2204460e-16
>>> CBF(i/Integer(3)).diameter().parent()
Real Field with 30 bits of precision
>>> CBF(CIF(RIF(RealNumber('1.02'), RealNumber('1.04')), RIF(RealNumber('2.1
↵'), RealNumber('2.2')))).diameter()
0.20000000

```

See also

`rad()`, `mid()`

eisenstein(*n*)

Return the first *n* entries in the sequence of Eisenstein series $G_4(\tau)$, $G_6(\tau)$, $G_8(\tau)$, ... where *tau* is given by `self`. The output is a list.

EXAMPLES:

```

sage: a, b, c, d = 2, 5, 1, 3
sage: tau = CBF(1,3)
sage: tau.eisenstein(4)
[[2.1646498507193 +/- ...e-14],
 [2.0346794456073 +/- ...e-14],
 [2.0081609898081 +/- ...e-14],
 [2.0019857082706 +/- ...e-14]]
sage: ((a*tau+b)/(c*tau+d)).eisenstein(3)[2]
[331011.2004330 +/- ...e-8] + [-711178.1655746 +/- ...e-8]*I
sage: (c*tau+d)^8 * tau.eisenstein(3)[2]
[331011.20043304 +/- ...e-9] + [-711178.1655746 +/- ...e-8]*I

```

```

>>> from sage.all import *
>>> a, b, c, d = Integer(2), Integer(5), Integer(1), Integer(3)
>>> tau = CBF(Integer(1), Integer(3))
>>> tau.eisenstein(Integer(4))
[[2.1646498507193 +/- ...e-14],
 [2.0346794456073 +/- ...e-14],
 [2.0081609898081 +/- ...e-14],
 [2.0019857082706 +/- ...e-14]]
>>> ((a*tau+b)/(c*tau+d)).eisenstein(Integer(3))[Integer(2)]
[331011.2004330 +/- ...e-8] + [-711178.1655746 +/- ...e-8]*I
>>> (c*tau+d)**Integer(8) * tau.eisenstein(Integer(3))[Integer(2)]
[331011.20043304 +/- ...e-9] + [-711178.1655746 +/- ...e-8]*I

```

elliptic_e()

Return the complete elliptic integral of the second kind evaluated at *m* given by `self`.

EXAMPLES:

```

sage: CBF(2,3).elliptic_e()
[1.472797144959 +/- ...e-13] + [-1.231604783936 +/- ...e-14]*I

```

```
>>> from sage.all import *
>>> CBF(Integer(2), Integer(3)).elliptic_e()
[1.472797144959 +/- ...e-13] + [-1.231604783936 +/- ...e-14]*I
```

elliptic_e_inc(m)

Return the incomplete elliptic integral of the second kind evaluated at m .

See `elliptic_e()` for the corresponding complete integral

INPUT:

- m – complex ball

EXAMPLES:

```
sage: CBF(1,2).elliptic_e_inc(CBF(0,1))
[1.906576998914 +/- ...e-13] + [3.6896645289411 +/- ...e-14]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(2)).elliptic_e_inc(CBF(Integer(0), Integer(1)))
[1.906576998914 +/- ...e-13] + [3.6896645289411 +/- ...e-14]*I
```

At parameter $\pi/2$ it is a complete integral:

```
sage: phi = CBF(1,1)
sage: (CBF.pi()/2).elliptic_e_inc(phi)
[1.2838409578982 +/- ...e-14] + [-0.5317843366915 +/- ...e-14]*I
sage: phi.elliptic_e()
[1.2838409578982 +/- 5...e-14] + [-0.5317843366915 +/- 3...e-14]*I

sage: phi = CBF(2, 3/7)
sage: (CBF.pi()/2).elliptic_e_inc(phi)
[0.787564350925 +/- ...e-13] + [-0.686896129145 +/- ...e-13]*I
sage: phi.elliptic_e()
[0.7875643509254 +/- ...e-14] + [-0.686896129145 +/- ...e-13]*I
```

```
>>> from sage.all import *
>>> phi = CBF(Integer(1), Integer(1))
>>> (CBF.pi()/Integer(2)).elliptic_e_inc(phi)
[1.2838409578982 +/- ...e-14] + [-0.5317843366915 +/- ...e-14]*I
>>> phi.elliptic_e()
[1.2838409578982 +/- 5...e-14] + [-0.5317843366915 +/- 3...e-14]*I

>>> phi = CBF(Integer(2), Integer(3)/Integer(7))
>>> (CBF.pi()/Integer(2)).elliptic_e_inc(phi)
[0.787564350925 +/- ...e-13] + [-0.686896129145 +/- ...e-13]*I
>>> phi.elliptic_e()
[0.7875643509254 +/- ...e-14] + [-0.686896129145 +/- ...e-13]*I
```

elliptic_f(m)

Return the incomplete elliptic integral of the first kind evaluated at m .

See `elliptic_k()` for the corresponding complete integral

INPUT:

- m – complex ball

EXAMPLES:

```
sage: CBF(1,2).elliptic_f(CBF(0,1))
[0.6821522911854 +/- ...e-14] + [1.2482780628143 +/- ...e-14]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(2)).elliptic_f(CBF(Integer(0), Integer(1)))
[0.6821522911854 +/- ...e-14] + [1.2482780628143 +/- ...e-14]*I
```

At parameter $\pi/2$ it is a complete integral:

```
sage: phi = CBF(1,1)
sage: (CBF.pi()/2).elliptic_f(phi)
[1.5092369540513 +/- ...e-14] + [0.6251464152027 +/- ...e-15]*I
sage: phi.elliptic_k()
[1.50923695405127 +/- ...e-15] + [0.62514641520270 +/- ...e-15]*I
```

```
sage: phi = CBF(2, 3/7)
sage: (CBF.pi()/2).elliptic_f(phi)
[1.3393589639094 +/- ...e-14] + [1.1104369690719 +/- ...e-14]*I
sage: phi.elliptic_k()
[1.33935896390938 +/- ...e-15] + [1.11043696907194 +/- ...e-15]*I
```

```
>>> from sage.all import *
>>> phi = CBF(Integer(1), Integer(1))
>>> (CBF.pi()/Integer(2)).elliptic_f(phi)
[1.5092369540513 +/- ...e-14] + [0.6251464152027 +/- ...e-15]*I
>>> phi.elliptic_k()
[1.50923695405127 +/- ...e-15] + [0.62514641520270 +/- ...e-15]*I

>>> phi = CBF(Integer(2), Integer(3)/Integer(7))
>>> (CBF.pi()/Integer(2)).elliptic_f(phi)
[1.3393589639094 +/- ...e-14] + [1.1104369690719 +/- ...e-14]*I
>>> phi.elliptic_k()
[1.33935896390938 +/- ...e-15] + [1.11043696907194 +/- ...e-15]*I
```

elliptic_invariants()

Return the lattice invariants (g_2 , g_3).

EXAMPLES:

```
sage: CBF(0,1).elliptic_invariants()
([189.07272012923 +/- ...e-12], [+/- ...e-12])
sage: CBF(sqrt(2)/2, sqrt(2)/2).elliptic_invariants()
↳needs sage.symbolic
([+/- ...e-12] + [-332.5338031465...]*I,
 [1254.46842157... ] + [1254.46842157...]*I)
```

```
>>> from sage.all import *
>>> CBF(Integer(0), Integer(1)).elliptic_invariants()
([189.07272012923 +/- ...e-12], [+/- ...e-12])
>>> CBF(sqrt(Integer(2))/Integer(2), sqrt(Integer(2))/Integer(2)).elliptic_
```

(continues on next page)

(continued from previous page)

```

↪invariants() # needs sage.symbolic
([+/- ...e-12] + [-332.5338031465...]*I,
 [1254.46842157... + [1254.46842157...]*I)

```

elliptic_k()

Return the complete elliptic integral of the first kind evaluated at m given by `self`.

EXAMPLES:

```

sage: CBF(2,3).elliptic_k()
[1.04291329192852 +/- ...e-15] + [0.62968247230864 +/- ...e-15]*I

```

```

>>> from sage.all import *
>>> CBF(Integer(2),Integer(3)).elliptic_k()
[1.04291329192852 +/- ...e-15] + [0.62968247230864 +/- ...e-15]*I

```

elliptic_p(tau, n=None)

Return the Weierstrass elliptic function with lattice parameter τ , evaluated at `self`. The function is doubly periodic in `self` with periods 1 and τ , which should lie in the upper half plane.

If n is given, return a list containing the first n terms in the Taylor expansion at `self`. In particular, with $n = 2$, compute the Weierstrass elliptic function together with its derivative, which generate the field of elliptic functions with periods 1 and τ .

EXAMPLES:

```

sage: tau = CBF(1,4)
sage: z = CBF(sqrt(2), sqrt(3)) #_
↪needs sage.symbolic
sage: z.elliptic_p(tau) #_
↪needs sage.symbolic
[-3.28920996772709 +/- ...e-15] + [-0.0003673767302933 +/- ...e-17]*I
sage: (z + tau).elliptic_p(tau) #_
↪needs sage.symbolic
[-3.28920996772709 +/- ...e-15] + [-0.000367376730293 +/- ...e-16]*I
sage: (z + 1).elliptic_p(tau) #_
↪needs sage.symbolic
[-3.28920996772709 +/- ...e-15] + [-0.0003673767302933 +/- ...e-17]*I

sage: z.elliptic_p(tau, 3) #_
↪needs sage.symbolic
[[-3.28920996772709 +/- ...e-15] + [-0.0003673767302933 +/- ...e-17]*I,
 [0.002473055794309 +/- ...e-16] + [0.003859554040267 +/- ...e-16]*I,
 [-0.01299087561709 +/- ...e-15] + [0.00725027521915 +/- ...e-15]*I]
sage: (z + 3 + 4*tau).elliptic_p(tau, 3) #_
↪needs sage.symbolic
[[-3.28920996772709 +/- ...e-15] + [-0.00036737673029 +/- ...e-15]*I,
 [0.0024730557943 +/- ...e-14] + [0.0038595540403 +/- ...e-14]*I,
 [-0.01299087562 +/- ...e-12] + [0.00725027522 +/- ...e-12]*I]

```

```

>>> from sage.all import *
>>> tau = CBF(Integer(1),Integer(4))
>>> z = CBF(sqrt(Integer(2)), sqrt(Integer(3)))

```

(continues on next page)

(continued from previous page)

```

↪          # needs sage.symbolic
>>> z.elliptic_p(tau)                                     #_
↪needs sage.symbolic
[-3.28920996772709 +/- ...e-15] + [-0.0003673767302933 +/- ...e-17]*I
>>> (z + tau).elliptic_p(tau)                             #_
↪needs sage.symbolic
[-3.28920996772709 +/- ...e-15] + [-0.000367376730293 +/- ...e-16]*I
>>> (z + Integer(1)).elliptic_p(tau)                     _
↪          # needs sage.symbolic
[-3.28920996772709 +/- ...e-15] + [-0.0003673767302933 +/- ...e-17]*I

>>> z.elliptic_p(tau, Integer(3))                         _
↪          # needs sage.symbolic
[[-3.28920996772709 +/- ...e-15] + [-0.0003673767302933 +/- ...e-17]*I,
 [0.002473055794309 +/- ...e-16] + [0.003859554040267 +/- ...e-16]*I,
 [-0.01299087561709 +/- ...e-15] + [0.00725027521915 +/- ...e-15]*I]
>>> (z + Integer(3) + Integer(4)*tau).elliptic_p(tau, Integer(3)) _
↪          # needs sage.symbolic
[[-3.28920996772709 +/- ...e-15] + [-0.00036737673029 +/- ...e-15]*I,
 [0.0024730557943 +/- ...e-14] + [0.0038595540403 +/- ...e-14]*I,
 [-0.01299087562 +/- ...e-12] + [0.00725027522 +/- ...e-12]*I]

```

elliptic_pi(*m*)

Return the complete elliptic integral of the third kind evaluated at *m* given by *self*.

EXAMPLES:

```

sage: CBF(2,3).elliptic_pi(CBF(1,1))
[0.2702999736198... ] + [0.715676058329...]*I

```

```

>>> from sage.all import *
>>> CBF(Integer(2),Integer(3)).elliptic_pi(CBF(Integer(1),Integer(1)))
[0.2702999736198... ] + [0.715676058329...]*I

```

elliptic_pi_inc(*phi*, *m*)

Return the Legendre incomplete elliptic integral of the third kind.

See: [elliptic_pi\(\)](#) for the complete integral.

INPUT:

- *phi* – complex ball
- *m* – complex ball

EXAMPLES:

```

sage: CBF(1,2).elliptic_pi_inc(CBF(0,1), CBF(2,-3))
[0.05738864021418 +/- ...e-15] + [0.55557494549951 +/- ...e-15]*I

```

```

>>> from sage.all import *
>>> CBF(Integer(1),Integer(2)).elliptic_pi_inc(CBF(Integer(0),Integer(1)),_
↪CBF(Integer(2),-Integer(3)))
[0.05738864021418 +/- ...e-15] + [0.55557494549951 +/- ...e-15]*I

```

At parameter $\pi/2$ it is a complete integral:

```
sage: n = CBF(1,1)
sage: m = CBF(-2/3, 3/5)
sage: n.elliptic_pi_inc(CBF.pi()/2, m) # this is a regression, see
↪:issue:28623
nan + nan*I
sage: n.elliptic_pi(m)
[0.8934793755173...] + [0.957078687107...]*I

sage: n = CBF(2, 3/7)
sage: m = CBF(-1/3, 2/9)
sage: n.elliptic_pi_inc(CBF.pi()/2, m) # this is a regression, see
↪:issue:28623
nan + nan*I
sage: n.elliptic_pi(m)
[0.296958874641...] + [1.318879533273...]*I
```

```
>>> from sage.all import *
>>> n = CBF(Integer(1), Integer(1))
>>> m = CBF(-Integer(2)/Integer(3), Integer(3)/Integer(5))
>>> n.elliptic_pi_inc(CBF.pi()/Integer(2), m) # this is a regression, see
↪:issue:28623
nan + nan*I
>>> n.elliptic_pi(m)
[0.8934793755173...] + [0.957078687107...]*I

>>> n = CBF(Integer(2), Integer(3)/Integer(7))
>>> m = CBF(-Integer(1)/Integer(3), Integer(2)/Integer(9))
>>> n.elliptic_pi_inc(CBF.pi()/Integer(2), m) # this is a regression, see
↪:issue:28623
nan + nan*I
>>> n.elliptic_pi(m)
[0.296958874641...] + [1.318879533273...]*I
```

elliptic_rf(y, z)

Return the Carlson symmetric elliptic integral of the first kind evaluated at (self, y, z).

INPUT:

- y – complex ball
- z – complex ball

EXAMPLES:

```
sage: CBF(0,1).elliptic_rf(CBF(-1/2,1), CBF(-1,-1))
[1.469800396738515 +/- ...e-16] + [-0.2358791199824196 +/- ...e-17]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(0), Integer(1)).elliptic_rf(CBF(-Integer(1)/Integer(2),
↪Integer(1)), CBF(-Integer(1), -Integer(1)))
[1.469800396738515 +/- ...e-16] + [-0.2358791199824196 +/- ...e-17]*I
```

elliptic_rg(y, z)

Return the Carlson symmetric elliptic integral of the second kind evaluated at (self, y, z).

INPUT:

- y – complex ball
- z – complex ball

EXAMPLES:

```
sage: CBF(0,1).elliptic_rg(CBF(-1/2,1), CBF(-1,-1))
[0.1586786770922370 +/- ...e-17] + [0.2239733128130531 +/- ...e-17]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(0), Integer(1)).elliptic_rg(CBF(-Integer(1)/Integer(2),
↪ Integer(1)), CBF(-Integer(1), -Integer(1)))
[0.1586786770922370 +/- ...e-17] + [0.2239733128130531 +/- ...e-17]*I
```

elliptic_rj(y, z, p)

Return the Carlson symmetric elliptic integral of the third kind evaluated at (self, y, z).

INPUT:

- y – complex ball
- z – complex ball
- p – complex bamm

EXAMPLES:

```
sage: CBF(0,1).elliptic_rj(CBF(-1/2,1), CBF(-1,-1), CBF(2))
[1.00438675628573...] + [-0.24516268343916...]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(0), Integer(1)).elliptic_rj(CBF(-Integer(1)/Integer(2),
↪ Integer(1)), CBF(-Integer(1), -Integer(1)), CBF(Integer(2)))
[1.00438675628573...] + [-0.24516268343916...]*I
```

elliptic_roots()

Return the lattice roots ($e1, e2, e3$) of $4z^3 - g_2z - g_3$.

EXAMPLES:

```
sage: e1, e2, e3 = CBF(0,1).elliptic_roots()
sage: e1, e2, e3
([6.8751858180204 +/- ...e-14],
 [+/- ...e-14],
 [-6.8751858180204 +/- ...e-14])
sage: g2, g3 = CBF(0,1).elliptic_invariants()
sage: 4 * e1^3 - g2 * e1 - g3
[+/- ...e-11]
```

```
>>> from sage.all import *
>>> e1, e2, e3 = CBF(Integer(0), Integer(1)).elliptic_roots()
>>> e1, e2, e3
([6.8751858180204 +/- ...e-14],
 [+/- ...e-14],
 [-6.8751858180204 +/- ...e-14])
```

(continues on next page)

(continued from previous page)

```
>>> g2, g3 = CBF(Integer(0), Integer(1)).elliptic_invariants()
>>> Integer(4) * e1**Integer(3) - g2 * e1 - g3
[+/- ...e-11]
```

elliptic_sigma(tau)

Return the value of the Weierstrass sigma function at (self, tau).

EXAMPLES:

```
- ``tau`` -- a complex ball with positive imaginary part
```

EXAMPLES:

```
sage: CBF(1,1).elliptic_sigma(CBF(1,3))
[-0.543073363596 +/- ...e-13] + [3.6357291186244 +/- ...e-14]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).elliptic_sigma(CBF(Integer(1), Integer(3)))
[-0.543073363596 +/- ...e-13] + [3.6357291186244 +/- ...e-14]*I
```

elliptic_zeta(tau)

Return the value of the Weierstrass zeta function at (self, tau).

EXAMPLES:

```
- ``tau`` -- a complex ball with positive imaginary part
```

EXAMPLES:

```
sage: CBF(1,1).elliptic_zeta(CBF(1,3))
[3.2898676194970 +/- ...e-14] + [0.1365414361782 +/- ...e-14]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).elliptic_zeta(CBF(Integer(1), Integer(3)))
[3.2898676194970 +/- ...e-14] + [0.1365414361782 +/- ...e-14]*I
```

erf()

Return the error function with argument self.

EXAMPLES:

```
sage: CBF(1, 1).erf()
[1.316151281697947 +/- ...e-16] + [0.1904534692378347 +/- ...e-17]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).erf()
[1.316151281697947 +/- ...e-16] + [0.1904534692378347 +/- ...e-17]*I
```

erfc()

Compute the complementary error function with argument self.

EXAMPLES:

```
sage: CBF(20).erfc() # abs tol 1e-190
[5.39586561160790e-176 +/- 6.73e-191]
sage: CBF(100, 100).erfc()
[0.00065234366376858 +/- ...e-18] + [-0.00393572636292141 +/- ...e-18]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(20)).erfc() # abs tol 1e-190
[5.39586561160790e-176 +/- 6.73e-191]
>>> CBF(Integer(100), Integer(100)).erfc()
[0.00065234366376858 +/- ...e-18] + [-0.00393572636292141 +/- ...e-18]*I
```

exp()

Return the exponential of this ball.

See also

[exppii\(\)](#)

EXAMPLES:

```
sage: CBF(i*pi).exp() #_
↪needs sage.symbolic
[-1.000000000000000 +/- ...e-16] + [+/- ...e-16]*I
```

```
>>> from sage.all import *
>>> CBF(i*pi).exp() #_
↪needs sage.symbolic
[-1.000000000000000 +/- ...e-16] + [+/- ...e-16]*I
```

exp_integral_e(s)

Return the image of this ball by the generalized exponential integral with index s .

EXAMPLES:

```
sage: CBF(1+i).exp_integral_e(1)
[0.00028162445198 +/- ...e-15] + [-0.17932453503936 +/- ...e-15]*I
sage: CBF(1+i).exp_integral_e(QQbar(i))
[-0.10396361883964 +/- ...e-15] + [-0.16268401277783 +/- ...e-15]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1)+i).exp_integral_e(Integer(1))
[0.00028162445198 +/- ...e-15] + [-0.17932453503936 +/- ...e-15]*I
>>> CBF(Integer(1)+i).exp_integral_e(QQbar(i))
[-0.10396361883964 +/- ...e-15] + [-0.16268401277783 +/- ...e-15]*I
```

exppii()

Return $\exp(\pi i \cdot \text{self})$.

EXAMPLES:

```
sage: CBF(1/2).exppii()
1.000000000000000*I
```

(continues on next page)

(continued from previous page)

```
sage: CBF(0, -1/pi).expzii()
↳needs sage.symbolic
[2.71828182845904 +/- ...e-15]
```

```
>>> from sage.all import *
>>> CBF(Integer(1)/Integer(2)).expzii()
1.000000000000000*I
>>> CBF(Integer(0), -Integer(1)/pi).expzii()
↳ # needs sage.symbolic
[2.71828182845904 +/- ...e-15]
```

gamma ($z=None$)

Return the image of this ball by the Euler Gamma function (if $z = None$) or the incomplete Gamma function (otherwise).

EXAMPLES:

```
sage: CBF(1, 1).gamma() # abs tol 1e-15
[0.498015668118356 +/- 1.26e-16] + [-0.1549498283018107 +/- 8.43e-17]*I
sage: CBF(-1).gamma()
nan
sage: CBF(1, 1).gamma(0) # abs tol 1e-15
[0.498015668118356 +/- 1.26e-16] + [-0.1549498283018107 +/- 8.43e-17]*I
sage: CBF(1, 1).gamma(100)
[-3.6143867454139e-45 +/- ...e-59] + [-3.7022961377791e-44 +/- ...e-58]*I
sage: CBF(1, 1).gamma(CLF(i)) # abs tol 1e-14
[0.328866841935004 +/- 7.07e-16] + [-0.189749450456210 +/- 9.05e-16]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).gamma() # abs tol 1e-15
[0.498015668118356 +/- 1.26e-16] + [-0.1549498283018107 +/- 8.43e-17]*I
>>> CBF(-Integer(1)).gamma()
nan
>>> CBF(Integer(1), Integer(1)).gamma(Integer(0)) # abs tol 1e-15
[0.498015668118356 +/- 1.26e-16] + [-0.1549498283018107 +/- 8.43e-17]*I
>>> CBF(Integer(1), Integer(1)).gamma(Integer(100))
[-3.6143867454139e-45 +/- ...e-59] + [-3.7022961377791e-44 +/- ...e-58]*I
>>> CBF(Integer(1), Integer(1)).gamma(CLF(i)) # abs tol 1e-14
[0.328866841935004 +/- 7.07e-16] + [-0.189749450456210 +/- 9.05e-16]*I
```

gamma_inc ($z=None$)

Return the image of this ball by the Euler Gamma function (if $z = None$) or the incomplete Gamma function (otherwise).

EXAMPLES:

```
sage: CBF(1, 1).gamma() # abs tol 1e-15
[0.498015668118356 +/- 1.26e-16] + [-0.1549498283018107 +/- 8.43e-17]*I
sage: CBF(-1).gamma()
nan
sage: CBF(1, 1).gamma(0) # abs tol 1e-15
[0.498015668118356 +/- 1.26e-16] + [-0.1549498283018107 +/- 8.43e-17]*I
```

(continues on next page)

(continued from previous page)

```
sage: CBF(1, 1).gamma(100)
[-3.6143867454139e-45 +/- ...e-59] + [-3.7022961377791e-44 +/- ...e-58]*I
sage: CBF(1, 1).gamma(CLF(i)) # abs tol 1e-14
[0.328866841935004 +/- 7.07e-16] + [-0.189749450456210 +/- 9.05e-16]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).gamma() # abs tol 1e-15
[0.498015668118356 +/- 1.26e-16] + [-0.1549498283018107 +/- 8.43e-17]*I
>>> CBF(-Integer(1)).gamma()
nan
>>> CBF(Integer(1), Integer(1)).gamma(Integer(0)) # abs tol 1e-15
[0.498015668118356 +/- 1.26e-16] + [-0.1549498283018107 +/- 8.43e-17]*I
>>> CBF(Integer(1), Integer(1)).gamma(Integer(100))
[-3.6143867454139e-45 +/- ...e-59] + [-3.7022961377791e-44 +/- ...e-58]*I
>>> CBF(Integer(1), Integer(1)).gamma(CLF(i)) # abs tol 1e-14
[0.328866841935004 +/- 7.07e-16] + [-0.189749450456210 +/- 9.05e-16]*I
```

gegenbauer_C(n, m)

Return the Gegenbauer polynomial (or function) $C_n^m(z)$ evaluated at `self`.

EXAMPLES:

```
sage: CBF(-10).gegenbauer_C(7, 1/2)
[-263813415.6250000 +/- ...e-8]
```

```
>>> from sage.all import *
>>> CBF(-Integer(10)).gegenbauer_C(Integer(7), Integer(1)/Integer(2))
[-263813415.6250000 +/- ...e-8]
```

hermite_H(n)

Return the Hermite function (or polynomial) of order `n` evaluated at `self`.

EXAMPLES:

```
sage: CBF(10).hermite_H(1)
20.000000000000000
sage: CBF(10).hermite_H(30)
[8.0574670961707e+37 +/- ...e+23]
```

```
>>> from sage.all import *
>>> CBF(Integer(10)).hermite_H(Integer(1))
20.000000000000000
>>> CBF(Integer(10)).hermite_H(Integer(30))
[8.0574670961707e+37 +/- ...e+23]
```

hypergeometric(a, b, regularized=False)

Return the generalized hypergeometric function of `self`.

INPUT:

- `a` – upper parameters; list of complex numbers that coerce into this ball's parent
- `b` – lower parameters; list of complex numbers that coerce into this ball's parent
- `regularized` – if `True`, the regularized generalized hypergeometric function is computed

OUTPUT: the generalized hypergeometric function defined by

$${}_pF_q(a_1, \dots, a_p; b_1, \dots, b_q; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k \dots (a_p)_k}{(b_1)_k \dots (b_q)_k} \frac{z^k}{k!}$$

extended using analytic continuation or regularization when the sum does not converge.

The regularized generalized hypergeometric function

$${}_pF_q(a_1, \dots, a_p; b_1, \dots, b_q; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k \dots (a_p)_k}{\Gamma(b_1 + k) \dots \Gamma(b_q + k)} \frac{z^k}{k!}$$

is well-defined even when the lower parameters are nonpositive integers. Currently, this is only supported for some p and q .

EXAMPLES:

```
sage: CBF(1, pi/2).hypergeometric([], []) #_
↪needs sage.symbolic
[+/- ...e-16] + [2.71828182845904 +/- ...e-15]*I

sage: CBF(1, pi).hypergeometric([1/4], [1/4]) #_
↪needs sage.symbolic
[-2.7182818284590 +/- ...e-14] + [+/- ...e-14]*I

sage: CBF(1000, 1000).hypergeometric([10], [AA(sqrt(2))]) #_
↪needs sage.symbolic
[9.79300951360e+454 +/- ...e+442] + [5.522579106816e+455 +/- ...e+442]*I
sage: CBF(1000, 1000).hypergeometric([100], [AA(sqrt(2))]) #_
↪needs sage.symbolic
[1.27967355557e+590 +/- ...e+578] + [-9.32333491987e+590 +/- ...e+578]*I

sage: CBF(0, 1).hypergeometric([], [1/2, 1/3, 1/4])
[-3.7991962344383 +/- ...e-14] + [23.878097177805 +/- ...e-13]*I

sage: CBF(0).hypergeometric([1], [])
1.0000000000000000
sage: CBF(1, 1).hypergeometric([1], [])
1.0000000000000000*I

sage: CBF(2+3*I).hypergeometric([1/4, 1/3], [1/2]) # abs tol 1e-14
[0.7871684267473 +/- 6.79e-14] + [0.2749254173721 +/- 8.82e-14]*I
sage: CBF(2+3*I).hypergeometric([1/4, 1/3], [1/2], regularized=True)
[0.4441122268685 +/- 3...e-14] + [0.1551100567338 +/- 5...e-14]*I

sage: CBF(5).hypergeometric([2, 3], [-5])
nan + nan*I
sage: CBF(5).hypergeometric([2, 3], [-5], regularized=True)
[5106.925964355 +/- ...e-10]

sage: CBF(2016).hypergeometric([], [2/3]) # abs tol 1e+26
[2.0256426923278e+38 +/- 9.59e+24]
sage: CBF(-2016).hypergeometric([], [2/3], regularized=True)
[-0.0005428550847 +/- ...e-14]
```

(continues on next page)

(continued from previous page)

```
sage: CBF(-7).hypergeometric([4], [])
0.0002441406250000000
```

```
sage: CBF(0, 3).hypergeometric([CBF(1,1)], [-4], regularized=True)
[239.514000752841 +/- ...e-13] + [105.175157349015 +/- ...e-13]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), pi/Integer(2)).hypergeometric([], [])
↪ # needs sage.symbolic
[+/- ...e-16] + [2.71828182845904 +/- ...e-15]*I

>>> CBF(Integer(1), pi).hypergeometric([Integer(1)/Integer(4)], [Integer(1)/
↪ Integer(4)])
↪ # needs sage.symbolic
[-2.7182818284590 +/- ...e-14] + [+/- ...e-14]*I

>>> CBF(Integer(1000), Integer(1000)).hypergeometric([Integer(10)],
↪ [AA(sqrt(Integer(2)))]
↪ # needs sage.symbolic
[9.79300951360e+454 +/- ...e+442] + [5.522579106816e+455 +/- ...e+442]*I
>>> CBF(Integer(1000), Integer(1000)).hypergeometric([Integer(100)],
↪ [AA(sqrt(Integer(2)))]
↪ # needs sage.symbolic
[1.27967355557e+590 +/- ...e+578] + [-9.3233491987e+590 +/- ...e+578]*I

>>> CBF(Integer(0), Integer(1)).hypergeometric([], [Integer(1)/Integer(2),
↪ Integer(1)/Integer(3), Integer(1)/Integer(4)])
[-3.7991962344383 +/- ...e-14] + [23.878097177805 +/- ...e-13]*I

>>> CBF(Integer(0)).hypergeometric([Integer(1)], [])
1.0000000000000000

>>> CBF(Integer(1), Integer(1)).hypergeometric([Integer(1)], [])
1.0000000000000000*I

>>> CBF(Integer(2)+Integer(3)*I).hypergeometric([Integer(1)/Integer(4),
↪ Integer(1)/Integer(3)], [Integer(1)/Integer(2)]) # abs tol 1e-14
[0.7871684267473 +/- 6.79e-14] + [0.2749254173721 +/- 8.82e-14]*I
>>> CBF(Integer(2)+Integer(3)*I).hypergeometric([Integer(1)/Integer(4),
↪ Integer(1)/Integer(3)], [Integer(1)/Integer(2)], regularized=True)
[0.4441122268685 +/- 3...e-14] + [0.1551100567338 +/- 5...e-14]*I

>>> CBF(Integer(5)).hypergeometric([Integer(2), Integer(3)], [-Integer(5)])
nan + nan*I
>>> CBF(Integer(5)).hypergeometric([Integer(2), Integer(3)], [-Integer(5)],
↪ regularized=True)
[5106.925964355 +/- ...e-10]

>>> CBF(Integer(2016)).hypergeometric([], [Integer(2)/Integer(3)]) # abs tol
↪ 1e+26
[2.0256426923278e+38 +/- 9.59e+24]
>>> CBF(-Integer(2016)).hypergeometric([], [Integer(2)/Integer(3)],
↪ regularized=True)
[-0.0005428550847 +/- ...e-14]

>>> CBF(-Integer(7)).hypergeometric([Integer(4)], [])
```

(continues on next page)

(continued from previous page)

```
0.0002441406250000000

>>> CBF(Integer(0), Integer(3)).hypergeometric([CBF(Integer(1), Integer(1))],
↪ [-Integer(4)], regularized=True)
[239.514000752841 +/- ...e-13] + [105.175157349015 +/- ...e-13]*I
```

hypergeometric_U(a, b)

Return the Tricomi confluent hypergeometric function $U(a, b, \text{self})$ of this ball.

EXAMPLES:

```
sage: CBF(1000, 1000).hypergeometric_U(RLF(pi), -100) #_
↪ needs sage.symbolic
[-7.261605907166e-11 +/- ...e-24] + [-7.928136216391e-11 +/- ...e-24]*I
sage: CBF(1000, 1000).hypergeometric_U(0, -100)
1.0000000000000000
```

```
>>> from sage.all import *
>>> CBF(Integer(1000), Integer(1000)).hypergeometric_U(RLF(pi), -
↪ Integer(100)) # needs sage.symbolic
[-7.261605907166e-11 +/- ...e-24] + [-7.928136216391e-11 +/- ...e-24]*I
>>> CBF(Integer(1000), Integer(1000)).hypergeometric_U(Integer(0), -
↪ Integer(100))
1.0000000000000000
```

identical(other)

Return whether `self` and `other` represent the same ball.

INPUT:

- `other` – a *ComplexBall*

OUTPUT:

Return True iff `self` and `other` are equal as sets, i.e. if their real and imaginary parts each have the same midpoint and radius.

Note that this is not the same thing as testing whether both `self` and `other` certainly represent the complex real number, unless either `self` or `other` is exact (and neither contains NaN). To test whether both operands might represent the same mathematical quantity, use *overlaps()* or *in*, depending on the circumstance.

EXAMPLES:

```
sage: CBF(1, 1/3).identical(1 + CBF(0, 1)/3)
True
sage: CBF(1, 1).identical(1 + CBF(0, 1/3)*3)
False
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)/Integer(3)).identical(Integer(1) +_
↪ CBF(Integer(0), Integer(1))/Integer(3))
True
>>> CBF(Integer(1), Integer(1)).identical(Integer(1) + CBF(Integer(0),_
↪ Integer(1)/Integer(3))*Integer(3))
False
```

imag()

Return the imaginary part of this ball.

OUTPUT: a *RealBall*

EXAMPLES:

```
sage: a = CBF(1/3, 1/5)
sage: a.imag()
[0.20000000000000000 +/- ...e-17]
sage: a.imag().parent()
Real ball field with 53 bits of precision
```

```
>>> from sage.all import *
>>> a = CBF(Integer(1)/Integer(3), Integer(1)/Integer(5))
>>> a.imag()
[0.20000000000000000 +/- ...e-17]
>>> a.imag().parent()
Real ball field with 53 bits of precision
```

is_NaN()

Return True iff either the real or the imaginary part is not-a-number.

EXAMPLES:

```
sage: CBF(NaN).is_NaN()
↪needs sage.symbolic
True
sage: CBF(-5).gamma().is_NaN()
True
sage: CBF(oo).is_NaN()
False
sage: CBF(42+I).is_NaN()
False
```

```
>>> from sage.all import *
>>> CBF(NaN).is_NaN()
↪needs sage.symbolic
True
>>> CBF(-Integer(5)).gamma().is_NaN()
True
>>> CBF(oo).is_NaN()
False
>>> CBF(Integer(42)+I).is_NaN()
False
```

is_exact()

Return True iff the radius of this ball is zero.

EXAMPLES:

```
sage: CBF(1).is_exact()
True
sage: CBF(1/3, 1/3).is_exact()
False
```

```
>>> from sage.all import *
>>> CBF(Integer(1)).is_exact()
True
>>> CBF(Integer(1)/Integer(3), Integer(1)/Integer(3)).is_exact()
False
```

is_nonzero()

Return True iff zero is not contained in the interval represented by this ball.

Note

This method is not the negation of `is_zero()`: it only returns True if zero is known not to be contained in the ball.

Use `bool(b)` (or, equivalently, `not b.is_zero()`) to check if a ball `b` **may** represent a nonzero number (for instance, to determine the “degree” of a polynomial with ball coefficients).

EXAMPLES:

```
sage: CBF(pi, 1/3).is_nonzero() #_
↪needs sage.symbolic
True
sage: CBF(RIF(-0.5, 0.5), 1/3).is_nonzero()
True
sage: CBF(1/3, RIF(-0.5, 0.5)).is_nonzero()
True
sage: CBF(RIF(-0.5, 0.5), RIF(-0.5, 0.5)).is_nonzero()
False
```

```
>>> from sage.all import *
>>> CBF(pi, Integer(1)/Integer(3)).is_nonzero() #_
↪# needs sage.symbolic
True
>>> CBF(RIF(-RealNumber('0.5'), RealNumber('0.5')), Integer(1)/Integer(3)).is_
↪nonzero()
True
>>> CBF(Integer(1)/Integer(3), RIF(-RealNumber('0.5'), RealNumber('0.5'))).is_
↪nonzero()
True
>>> CBF(RIF(-RealNumber('0.5'), RealNumber('0.5')), RIF(-RealNumber('0.5'),
↪RealNumber('0.5'))).is_nonzero()
False
```

See also

`is_zero()`

is_real()

Return True iff the imaginary part of this ball is exactly zero.

EXAMPLES:

```
sage: CBF(1/3, 0).is_real()
True
sage: (CBF(i/3) - CBF(1, 1/3)).is_real()
False
sage: CBF('inf').is_real()
True
```

```
>>> from sage.all import *
>>> CBF(Integer(1)/Integer(3), Integer(0)).is_real()
True
>>> (CBF(i/Integer(3)) - CBF(Integer(1), Integer(1)/Integer(3))).is_real()
False
>>> CBF('inf').is_real()
True
```

is_zero()

Return True iff the midpoint and radius of this ball are both zero.

EXAMPLES:

```
sage: CBF(0).is_zero()
True
sage: CBF(RIF(-0.5, 0.5)).is_zero()
False
```

```
>>> from sage.all import *
>>> CBF(Integer(0)).is_zero()
True
>>> CBF(RIF(-RealNumber('0.5'), RealNumber('0.5'))).is_zero()
False
```

See also

[`is\_nonzero\(\)`](#)

jacobi_P(*n*, *a*, *b*)

Return the Jacobi polynomial (or function) $P_n^{(a,b)}(z)$ evaluated at `self`.

EXAMPLES:

```
sage: CBF(5, -6).jacobi_P(8, CBF(1, 2), CBF(2, 3))
[-920983000.45982 +/- ...e-6] + [6069919969.92857 +/- ...e-6]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(5), -Integer(6)).jacobi_P(Integer(8), CBF(Integer(1),
↪ Integer(2)), CBF(Integer(2), Integer(3)))
[-920983000.45982 +/- ...e-6] + [6069919969.92857 +/- ...e-6]*I
```

jacobi_theta(*tau*)

Return the four Jacobi theta functions evaluated at the argument `self` (representing z) and the parameter `tau` which should lie in the upper half plane.

The following definitions are used:

$$\theta_1(z, \tau) = 2q_{1/4} \sum_{n=0}^{\infty} (-1)^n q^{n(n+1)} \sin((2n+1)\pi z)$$

$$\theta_2(z, \tau) = 2q_{1/4} \sum_{n=0}^{\infty} q^{n(n+1)} \cos((2n+1)\pi z)$$

$$\theta_3(z, \tau) = 1 + 2 \sum_{n=1}^{\infty} q^{n^2} \cos(2n\pi z)$$

$$\theta_4(z, \tau) = 1 + 2 \sum_{n=1}^{\infty} (-1)^n q^{n^2} \cos(2n\pi z)$$

where $q = \exp(\pi i \tau)$ and $q_{1/4} = \exp(\pi i \tau / 4)$. Note that z is multiplied by π ; some authors omit this factor.

EXAMPLES:

```
sage: CBF(3, -1/2).jacobi_theta(CBF(1/4, 2))
([-0.186580562274757 +/- ...e-16] + [0.93841744788594 +/- ...e-15]*I,
 [-1.02315311037951 +/- ...e-15] + [-0.203600094532010 +/- ...e-16]*I,
 [1.030613911309632 +/- ...e-16] + [0.030613917822067 +/- ...e-16]*I,
 [0.969386075665498 +/- ...e-16] + [-0.030613917822067 +/- ...e-16]*I)
```

```
sage: CBF(3, -1/2).jacobi_theta(CBF(1/4, -2))
(nan + nan*I, nan + nan*I, nan + nan*I, nan + nan*I)
```

```
sage: CBF(0).jacobi_theta(CBF(0, 1))
(0,
 [0.913579138156117 +/- ...e-16],
 [1.086434811213308 +/- ...e-16],
 [0.913579138156117 +/- ...e-16])
```

```
>>> from sage.all import *
>>> CBF(Integer(3), -Integer(1)/Integer(2)).jacobi_theta(CBF(Integer(1)/
↪Integer(4), Integer(2)))
([-0.186580562274757 +/- ...e-16] + [0.93841744788594 +/- ...e-15]*I,
 [-1.02315311037951 +/- ...e-15] + [-0.203600094532010 +/- ...e-16]*I,
 [1.030613911309632 +/- ...e-16] + [0.030613917822067 +/- ...e-16]*I,
 [0.969386075665498 +/- ...e-16] + [-0.030613917822067 +/- ...e-16]*I)

>>> CBF(Integer(3), -Integer(1)/Integer(2)).jacobi_theta(CBF(Integer(1)/
↪Integer(4), -Integer(2)))
(nan + nan*I, nan + nan*I, nan + nan*I, nan + nan*I)

>>> CBF(Integer(0)).jacobi_theta(CBF(Integer(0), Integer(1)))
(0,
 [0.913579138156117 +/- ...e-16],
 [1.086434811213308 +/- ...e-16],
 [0.913579138156117 +/- ...e-16])
```

laguerre_L($n, m=0$)

Return the Laguerre polynomial (or function) $L_n^m(z)$ evaluated at `self`.

EXAMPLES:

```
sage: CBF(10).laguerre_L(3)
[-45.66666666666666 +/- ...e-14]
sage: CBF(10).laguerre_L(3, 2)
[-6.66666666666667 +/- ...e-13]
sage: CBF(5,7).laguerre_L(CBF(2,3), CBF(1,-2)) # abs tol 1e-9
[5515.3150302713 +/- 5.02e-11] + [-12386.9428452714 +/- 6.21e-11]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(10)).laguerre_L(Integer(3))
[-45.66666666666666 +/- ...e-14]
>>> CBF(Integer(10)).laguerre_L(Integer(3), Integer(2))
[-6.66666666666667 +/- ...e-13]
>>> CBF(Integer(5), Integer(7)).laguerre_L(CBF(Integer(2), Integer(3)), -
↪ CBF(Integer(1), -Integer(2))) # abs tol 1e-9
[5515.3150302713 +/- 5.02e-11] + [-12386.9428452714 +/- 6.21e-11]*I
```

`lambert_w(branch=0)`

Return the image of this ball by the specified branch of the Lambert W function.

EXAMPLES:

```
sage: CBF(1 + I).lambert_w()
[0.6569660692304... ] + [0.3254503394134...]*I
sage: CBF(1 + I).lambert_w(2)
[-2.1208839379437... ] + [11.600137110774...]*I
sage: CBF(1 + I).lambert_w(2^100)
[-70.806021532123... ] + [7.9648836259913...]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1) + I).lambert_w()
[0.6569660692304... ] + [0.3254503394134...]*I
>>> CBF(Integer(1) + I).lambert_w(Integer(2))
[-2.1208839379437... ] + [11.600137110774...]*I
>>> CBF(Integer(1) + I).lambert_w(Integer(2)**Integer(100))
[-70.806021532123... ] + [7.9648836259913...]*I
```

`legendre_P(n, m=0, type=2)`

Return the Legendre function of the first kind $P_n^m(z)$ evaluated at `self`.

The `type` parameter can be either 2 or 3. This selects between different branch cut conventions. The definitions of the “type 2” and “type 3” functions are the same as those used by *Mathematica* and *mpmath*.

EXAMPLES:

```
sage: CBF(1/2).legendre_P(5)
[0.0898437500000000 +/- 7...e-17]
sage: CBF(1,2).legendre_P(CBF(2,3), CBF(0,1))
[0.10996180744364 +/- ...e-15] + [0.14312767804055 +/- ...e-15]*I
sage: CBF(-10).legendre_P(5, 325/100)
[-22104403.487377 +/- ...e-7] + [53364750.687392 +/- ...e-7]*I
sage: CBF(-10).legendre_P(5, 325/100, type=3)
[-57761589.914581 +/- ...e-7] + [+/- ...e-7]*I
```

```

>>> from sage.all import *
>>> CBF(Integer(1)/Integer(2)).legendre_P(Integer(5))
[0.0898437500000000 +/- 7...e-17]
>>> CBF(Integer(1), Integer(2)).legendre_P(CBF(Integer(2), Integer(3)),
↪ CBF(Integer(0), Integer(1)))
[0.10996180744364 +/- ...e-15] + [0.14312767804055 +/- ...e-15]*I
>>> CBF(-Integer(10)).legendre_P(Integer(5), Integer(325)/Integer(100))
[-22104403.487377 +/- ...e-7] + [53364750.687392 +/- ...e-7]*I
>>> CBF(-Integer(10)).legendre_P(Integer(5), Integer(325)/Integer(100),
↪ type=Integer(3))
[-57761589.914581 +/- ...e-7] + [+/- ...e-7]*I

```

legendre_Q($n, m=0, type=2$)

Return the Legendre function of the second kind $Q_n^m(z)$ evaluated at `self`.

The `type` parameter can be either 2 or 3. This selects between different branch cut conventions. The definitions of the “type 2” and “type 3” functions are the same as those used by *Mathematica* and *mpmath*.

EXAMPLES:

```

sage: CBF(1/2).legendre_Q(5)
[0.55508089057168 +/- ...e-15]
sage: CBF(1,2).legendre_Q(CBF(2,3), CBF(0,1))
[0.167678710 +/- ...e-10] + [-0.161558598 +/- ...e-10]*I
sage: CBF(-10).legendre_Q(5, 325/100)
[-83825154.36008 +/- ...e-6] + [-34721515.80396 +/- ...e-6]*I
sage: CBF(-10).legendre_Q(5, 325/100, type=3)
[-4.797306921692e-6 +/- ...e-19] + [-4.797306921692e-6 +/- ...e-19]*I

```

```

>>> from sage.all import *
>>> CBF(Integer(1)/Integer(2)).legendre_Q(Integer(5))
[0.55508089057168 +/- ...e-15]
>>> CBF(Integer(1), Integer(2)).legendre_Q(CBF(Integer(2), Integer(3)),
↪ CBF(Integer(0), Integer(1)))
[0.167678710 +/- ...e-10] + [-0.161558598 +/- ...e-10]*I
>>> CBF(-Integer(10)).legendre_Q(Integer(5), Integer(325)/Integer(100))
[-83825154.36008 +/- ...e-6] + [-34721515.80396 +/- ...e-6]*I
>>> CBF(-Integer(10)).legendre_Q(Integer(5), Integer(325)/Integer(100),
↪ type=Integer(3))
[-4.797306921692e-6 +/- ...e-19] + [-4.797306921692e-6 +/- ...e-19]*I

```

li(`offset=False`)

Return the logarithmic integral with argument `self`.

If `offset` is `True`, return the offset logarithmic integral.

EXAMPLES:

```

sage: CBF(1, 1).li() # abs tol 6e-15
[0.61391166922120 +/- 6.23e-15] + [2.05958421419258 +/- 5.59e-15]*I
sage: CBF(0).li()
0
sage: CBF(0).li(offset=True)
[-1.045163780117493 +/- ...e-16]

```

(continues on next page)

(continued from previous page)

```

sage: li(0).n()
↳needs sage.symbolic
0.0000000000000000
sage: Li(0).n()
↳needs sage.symbolic
-1.04516378011749

```

```

>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).li() # abs tol 6e-15
[0.61391166922120 +/- 6.23e-15] + [2.05958421419258 +/- 5.59e-15]*I
>>> CBF(Integer(0)).li()
0
>>> CBF(Integer(0)).li(offset=True)
[-1.045163780117493 +/- ...e-16]
>>> li(Integer(0)).n()
↳ # needs sage.symbolic
0.0000000000000000
>>> Li(Integer(0)).n()
↳ # needs sage.symbolic
-1.04516378011749

```

log (*base=None, analytic=False*)

General logarithm (principal branch).

INPUT:

- *base* – (optional) complex ball or number; if *None*, return the principal branch of the natural logarithm $\ln(\text{self})$, otherwise, return the general logarithm $\ln(\text{self})/\ln(\text{base})$
- *analytic* – boolean (default: *False*); if *True*, return an indeterminate (not-a-number) value when the input ball touches the branch cut (with respect to *self*)

EXAMPLES:

```

sage: CBF(2*I).log()
[0.693147180559945 +/- ...e-16] + [1.570796326794897 +/- ...e-16]*I
sage: CBF(-1).log()
[3.141592653589793 +/- ...e-16]*I

sage: CBF(2*I).log(2)
[1.000000000000000 +/- ...e-16] + [2.26618007091360 +/- ...e-15]*I
sage: CBF(2*I).log(CBF(i))
[1.000000000000000 +/- ...e-16] + [-0.441271200305303 +/- ...e-16]*I

sage: CBF('inf').log()
[+/- inf]
sage: CBF(2).log(0)
nan + nan*I

sage: CBF(-1).log(2)
[4.53236014182719 +/- ...e-15]*I
sage: CBF(-1).log(2, analytic=True)
nan + nan*I

```

(continues on next page)

(continued from previous page)

```
sage: CBF(-1, RBF(0, rad=.1r)).log(analytic=False)
[+/- ...e-3] + [+/- 3.15]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(2)*i).log()
[0.693147180559945 +/- ...e-16] + [1.570796326794897 +/- ...e-16]*I
>>> CBF(-Integer(1)).log()
[3.141592653589793 +/- ...e-16]*I

>>> CBF(Integer(2)*i).log(Integer(2))
[1.000000000000000 +/- ...e-16] + [2.26618007091360 +/- ...e-15]*I
>>> CBF(Integer(2)*i).log(CBF(i))
[1.000000000000000 +/- ...e-16] + [-0.441271200305303 +/- ...e-16]*I

>>> CBF('inf').log()
[+/- inf]
>>> CBF(Integer(2)).log(Integer(0))
nan + nan*I

>>> CBF(-Integer(1)).log(Integer(2))
[4.53236014182719 +/- ...e-15]*I
>>> CBF(-Integer(1)).log(Integer(2), analytic=True)
nan + nan*I
>>> CBF(-Integer(1), RBF(Integer(0), rad=.1)).log(analytic=False)
[+/- ...e-3] + [+/- 3.15]*I
```

log1p (*analytic=False*)

Return $\log(1 + \text{self})$, computed accurately when *self* is close to zero.

INPUT:

- *analytic* – boolean (default: False); if True, return an indeterminate (not-a-number) value when the input ball touches the branch cut

EXAMPLES:

```
sage: eps = RBF(1e-50)
sage: CBF(1+eps, eps).log()
[+/- ...e-16] + [1.000000000000000e-50 +/- ...e-66]*I
sage: CBF(eps, eps).log1p()
[1.000000000000000e-50 +/- ...e-68] + [1.000000000000000e-50 +/- ...e-66]*I
sage: CBF(-3/2).log1p(analytic=True)
nan + nan*I
```

```
>>> from sage.all import *
>>> eps = RBF(RealNumber('1e-50'))
>>> CBF(Integer(1)+eps, eps).log()
[+/- ...e-16] + [1.000000000000000e-50 +/- ...e-66]*I
>>> CBF(eps, eps).log1p()
[1.000000000000000e-50 +/- ...e-68] + [1.000000000000000e-50 +/- ...e-66]*I
>>> CBF(-Integer(3)/Integer(2)).log1p(analytic=True)
nan + nan*I
```

log_barnes_g()

Return the logarithmic Barnes G-function of `self`.

EXAMPLES:

```
sage: CBF(10^100).log_barnes_g()
[1.14379254649702e+202 +/- ...e+187]
sage: CBF(0,1000).log_barnes_g()
[-2702305.04929258 +/- ...e-9] + [-790386.325561423 +/- ...e-10]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(10)**Integer(100)).log_barnes_g()
[1.14379254649702e+202 +/- ...e+187]
>>> CBF(Integer(0),Integer(1000)).log_barnes_g()
[-2702305.04929258 +/- ...e-9] + [-790386.325561423 +/- ...e-10]*I
```

log_gamma(analytic=False)

Return the image of this ball by the logarithmic Gamma function.

The branch cut of the logarithmic gamma function is placed on the negative half-axis, which means that $\log_gamma(z) + \log z = \log_gamma(z+1)$ holds for all z , whereas $\log_gamma(z) \neq \log(\gamma(z))$ in general.

INPUT:

- `analytic` – boolean (default: `False`); if `True`, return an indeterminate (not-a-number) value when the input ball touches the branch cut

EXAMPLES:

```
sage: CBF(1000, 1000).log_gamma()
[5466.22252162990 +/- ...e-12] + [7039.33429191119 +/- ...e-12]*I
sage: CBF(-1/2).log_gamma()
[1.265512123484645 +/- ...e-16] + [-3.141592653589793 +/- ...e-16]*I
sage: CBF(-1).log_gamma()
nan + ...*I
sage: CBF(-3/2).log_gamma() # abs tol 1e-14
[0.860047015376481 +/- 3.82e-16] + [-6.283185307179586 +/- 6.77e-16]*I
sage: CBF(-3/2).log_gamma(analytic=True)
nan + nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1000), Integer(1000)).log_gamma()
[5466.22252162990 +/- ...e-12] + [7039.33429191119 +/- ...e-12]*I
>>> CBF(-Integer(1)/Integer(2)).log_gamma()
[1.265512123484645 +/- ...e-16] + [-3.141592653589793 +/- ...e-16]*I
>>> CBF(-Integer(1)).log_gamma()
nan + ...*I
>>> CBF(-Integer(3)/Integer(2)).log_gamma() # abs tol 1e-14
[0.860047015376481 +/- 3.82e-16] + [-6.283185307179586 +/- 6.77e-16]*I
>>> CBF(-Integer(3)/Integer(2)).log_gamma(analytic=True)
nan + nan*I
```

log_integral(offset=False)

Return the logarithmic integral with argument `self`.

If `offset` is `True`, return the offset logarithmic integral.

EXAMPLES:

```
sage: CBF(1, 1).li() # abs tol 6e-15
[0.61391166922120 +/- 6.23e-15] + [2.05958421419258 +/- 5.59e-15]*I
sage: CBF(0).li()
0
sage: CBF(0).li(offset=True)
[-1.045163780117493 +/- ...e-16]
sage: li(0).n() #_
↪needs sage.symbolic
0.0000000000000000
sage: Li(0).n() #_
↪needs sage.symbolic
-1.04516378011749
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).li() # abs tol 6e-15
[0.61391166922120 +/- 6.23e-15] + [2.05958421419258 +/- 5.59e-15]*I
>>> CBF(Integer(0)).li()
0
>>> CBF(Integer(0)).li(offset=True)
[-1.045163780117493 +/- ...e-16]
>>> li(Integer(0)).n() #_
↪ # needs sage.symbolic
0.0000000000000000
>>> Li(Integer(0)).n() #_
↪ # needs sage.symbolic
-1.04516378011749
```

`log_integral_offset()`

Offset logarithmic integral.

EXAMPLES:

```
sage: CBF(0).Li()
[-1.045163780117493 +/- ...e-16]
sage: li(0).n() #_
↪needs sage.symbolic
0.0000000000000000
sage: Li(0).n() #_
↪needs sage.symbolic
-1.04516378011749
```

```
>>> from sage.all import *
>>> CBF(Integer(0)).Li()
[-1.045163780117493 +/- ...e-16]
>>> li(Integer(0)).n() #_
↪ # needs sage.symbolic
0.0000000000000000
>>> Li(Integer(0)).n() #_
↪ # needs sage.symbolic
-1.04516378011749
```

mid()

Return the midpoint of this ball.

OUTPUT:

ComplexNumber, floating-point complex number formed by the centers of the real and imaginary parts of this ball.

EXAMPLES:

```

sage: CBF(1/3, 1).mid()
0.3333333333333333 + 1.000000000000000*I
sage: CBF(1/3, 1).mid().parent()
Complex Field with 53 bits of precision
sage: CBF('inf', 'nan').mid()
+infinity + NaN*I
sage: CBF('nan', 'inf').mid()
NaN + +infinity*I
sage: CBF('nan').mid()
NaN
sage: CBF('inf').mid()
+infinity
sage: CBF(0, 'inf').mid()
+infinity*I

```

```

>>> from sage.all import *
>>> CBF(Integer(1)/Integer(3), Integer(1)).mid()
0.3333333333333333 + 1.000000000000000*I
>>> CBF(Integer(1)/Integer(3), Integer(1)).mid().parent()
Complex Field with 53 bits of precision
>>> CBF('inf', 'nan').mid()
+infinity + NaN*I
>>> CBF('nan', 'inf').mid()
NaN + +infinity*I
>>> CBF('nan').mid()
NaN
>>> CBF('inf').mid()
+infinity
>>> CBF(Integer(0), 'inf').mid()
+infinity*I

```

See also*squash()***modular_delta()**Return the modular discriminant with *tau* given by *self*.

EXAMPLES:

```

sage: CBF(0,1).modular_delta()
[0.0017853698506421 +/- ...e-17]
sage: a, b, c, d = 2, 5, 1, 3
sage: tau = CBF(1,3)

```

(continues on next page)

(continued from previous page)

```
sage: ((a*tau+b)/(c*tau+d)).modular_delta()
[0.20921376655 +/- ...e-12] + [1.57611925523 +/- ...e-12]*I
sage: (c*tau+d)^12 * tau.modular_delta()
[0.20921376654986 +/- ...e-15] + [1.5761192552253 +/- ...e-14]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(0), Integer(1)).modular_delta()
[0.0017853698506421 +/- ...e-17]
>>> a, b, c, d = Integer(2), Integer(5), Integer(1), Integer(3)
>>> tau = CBF(Integer(1), Integer(3))
>>> ((a*tau+b)/(c*tau+d)).modular_delta()
[0.20921376655 +/- ...e-12] + [1.57611925523 +/- ...e-12]*I
>>> (c*tau+d)**Integer(12) * tau.modular_delta()
[0.20921376654986 +/- ...e-15] + [1.5761192552253 +/- ...e-14]*I
```

modular_eta()Return the Dedekind eta function with *tau* given by *self*.

EXAMPLES:

```
sage: CBF(0,1).modular_eta()
[0.768225422326057 +/- ...e-16]
sage: CBF(12,1).modular_eta()
[-0.768225422326057 +/- ...e-16]
```

```
>>> from sage.all import *
>>> CBF(Integer(0), Integer(1)).modular_eta()
[0.768225422326057 +/- ...e-16]
>>> CBF(Integer(12), Integer(1)).modular_eta()
[-0.768225422326057 +/- ...e-16]
```

modular_j()Return the modular j-invariant with *tau* given by *self*.

EXAMPLES:

```
sage: CBF(0,1).modular_j()
[1728.00000000000 +/- ...e-11]
```

```
>>> from sage.all import *
>>> CBF(Integer(0), Integer(1)).modular_j()
[1728.00000000000 +/- ...e-11]
```

modular_lambda()Return the modular lambda function with *tau* given by *self*.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: tau = CBF(sqrt(2), pi)
sage: tau.modular_lambda()
[-0.00022005123884157 +/- ...e-18] + [-0.00079787346459944 +/- ...e-18]*I
```

(continues on next page)

(continued from previous page)

```
sage: (tau + 2).modular_lambda()
[-0.00022005123884157 +/- ...e-18] + [-0.00079787346459944 +/- ...e-18]*I
sage: (tau / (1 - 2*tau)).modular_lambda()
[-0.00022005123884 +/- ...e-15] + [-0.00079787346460 +/- ...e-15]*I
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> tau = CBF(sqrt(Integer(2)), pi)
>>> tau.modular_lambda()
[-0.00022005123884157 +/- ...e-18] + [-0.00079787346459944 +/- ...e-18]*I
>>> (tau + Integer(2)).modular_lambda()
[-0.00022005123884157 +/- ...e-18] + [-0.00079787346459944 +/- ...e-18]*I
>>> (tau / (Integer(1) - Integer(2)*tau)).modular_lambda()
[-0.00022005123884 +/- ...e-15] + [-0.00079787346460 +/- ...e-15]*I
```

nbits()

Return the minimum precision sufficient to represent this ball exactly.

More precisely, the output is the number of bits needed to represent the absolute value of the mantissa of both the real and the imaginary part of the midpoint.

EXAMPLES:

```
sage: CBF(17, 1023).nbits()
10
sage: CBF(1/3, NaN).nbits()
↳needs sage.symbolic
53
sage: CBF(NaN).nbits()
↳needs sage.symbolic
0
```

```
>>> from sage.all import *
>>> CBF(Integer(17), Integer(1023)).nbits()
10
>>> CBF(Integer(1)/Integer(3), NaN).nbits()
↳
↳needs sage.symbolic
53
>>> CBF(NaN).nbits()
↳needs sage.symbolic
0
```

overlaps(other)

Return True iff self and other have some point in common.

INPUT:

- other – a *ComplexBall*

EXAMPLES:

```
sage: CBF(1, 1).overlaps(1 + CBF(0, 1/3)*3)
True
sage: CBF(1, 1).overlaps(CBF(1, 'nan'))
```

(continues on next page)

(continued from previous page)

```
True
sage: CBF(1, 1).overlaps(CBF(0, 'nan'))
False
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).overlaps(Integer(1) + CBF(Integer(0),
↳ Integer(1)/Integer(3))*Integer(3))
True
>>> CBF(Integer(1), Integer(1)).overlaps(CBF(Integer(1), 'nan'))
True
>>> CBF(Integer(1), Integer(1)).overlaps(CBF(Integer(0), 'nan'))
False
```

polylog(*s*)

Return the polylogarithm $\text{Li}_s(\text{self})$.

EXAMPLES:

```
sage: CBF(2).polylog(1)
[+/- ...e-15] + [-3.14159265358979 +/- ...e-15]*I
sage: CBF(1, 1).polylog(CBF(1, 1))
[0.3708160030469 +/- ...e-14] + [2.7238016577979 +/- ...e-14]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(2)).polylog(Integer(1))
[+/- ...e-15] + [-3.14159265358979 +/- ...e-15]*I
>>> CBF(Integer(1), Integer(1)).polylog(CBF(Integer(1), Integer(1)))
[0.3708160030469 +/- ...e-14] + [2.7238016577979 +/- ...e-14]*I
```

pow(*expo*, *analytic=False*)

Raise this ball to the power of *expo*.

INPUT:

- *analytic* – boolean (default: `False`); if `True`, return an indeterminate (not-a-number) value when the exponent is not an integer and the base ball touches the branch cut of the logarithm

EXAMPLES:

```
sage: CBF(-1).pow(CBF(i))
[0.0432139182637723 +/- ...e-17]
sage: CBF(-1).pow(CBF(i), analytic=True)
nan + nan*I
sage: CBF(-10).pow(-2)
[0.01000000000000000 +/- ...e-18]
sage: CBF(-10).pow(-2, analytic=True)
[0.01000000000000000 +/- ...e-18]
```

```
>>> from sage.all import *
>>> CBF(-Integer(1)).pow(CBF(i))
[0.0432139182637723 +/- ...e-17]
>>> CBF(-Integer(1)).pow(CBF(i), analytic=True)
nan + nan*I
```

(continues on next page)

(continued from previous page)

```
>>> CBF(-Integer(10)).pow(-Integer(2))
[0.010000000000000000 +/- ...e-18]
>>> CBF(-Integer(10)).pow(-Integer(2), analytic=True)
[0.010000000000000000 +/- ...e-18]
```

psi (*n=None*)Compute the digamma function with argument *self*.If *n* is provided, compute the polygamma function of order *n* and argument *self*.

EXAMPLES:

```
sage: CBF(1, 1).psi()
[0.0946503206224770 +/- ...e-17] + [1.076674047468581 +/- ...e-16]*I
sage: CBF(-1).psi()
nan
sage: CBF(1,1).psi(10)
[56514.8269344249 +/- ...e-11] + [56215.1218005823 +/- ...e-11]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).psi()
[0.0946503206224770 +/- ...e-17] + [1.076674047468581 +/- ...e-16]*I
>>> CBF(-Integer(1)).psi()
nan
>>> CBF(Integer(1), Integer(1)).psi(Integer(10))
[56514.8269344249 +/- ...e-11] + [56215.1218005823 +/- ...e-11]*I
```

rad ()

Return an upper bound for the error radius of this ball.

OUTPUT:

A *RealNumber* of the same precision as the radii of real balls.**Warning**

Unlike a *RealBall*, a *ComplexBall* is *not* defined by its midpoint and radius. (Instances of *ComplexBall* are actually rectangles, not balls.)

EXAMPLES:

```
sage: CBF(1 + i).rad()
0.00000000
sage: CBF(i/3).rad()
1.1102230e-16
sage: CBF(i/3).rad().parent()
Real Field with 30 bits of precision
```

```
>>> from sage.all import *
>>> CBF(Integer(1) + i).rad()
0.00000000
>>> CBF(i/Integer(3)).rad()
```

(continues on next page)

(continued from previous page)

```
1.1102230e-16
>>> CBF(i/Integer(3)).rad().parent()
Real Field with 30 bits of precision
```

See also

diameter(), *mid()*

real()

Return the real part of this ball.

OUTPUT: a *RealBall*

EXAMPLES:

```
sage: a = CBF(1/3, 1/5)
sage: a.real()
[0.3333333333333333 +/- ...e-17]
sage: a.real().parent()
Real ball field with 53 bits of precision
```

```
>>> from sage.all import *
>>> a = CBF(Integer(1)/Integer(3), Integer(1)/Integer(5))
>>> a.real()
[0.3333333333333333 +/- ...e-17]
>>> a.real().parent()
Real ball field with 53 bits of precision
```

rgamma()

Compute the reciprocal gamma function with argument *self*.

EXAMPLES:

```
sage: CBF(6).rgamma()
[0.008333333333333333 +/- ...e-18]
sage: CBF(-1).rgamma()
0
```

```
>>> from sage.all import *
>>> CBF(Integer(6)).rgamma()
[0.008333333333333333 +/- ...e-18]
>>> CBF(-Integer(1)).rgamma()
0
```

rising_factorial(*n*)

Return the *n*-th rising factorial of this ball.

The *n*-th rising factorial of *x* is equal to $x(x+1)\cdots(x+n-1)$.

For complex *n*, it is a quotient of gamma functions.

EXAMPLES:

```
>>> from sage.all import *
>>> CBF(Integer(1)).rising_factorial(Integer(5))
120.00000000000000
>>> CBF(Integer(1)/Integer(3), Integer(1)/Integer(2)).rising_
↪factorial(Integer(300))
[-3.87949484514e+612 +/- 5...e+600] + [-3.52042209763e+612 +/- 5...e+600]*I
>>> CBF(Integer(1)).rising_factorial(-Integer(1))
nan
>>> CBF(Integer(1)).rising_factorial(Integer(2)**Integer(64))
[+/- ...e+347382171326740403407]
>>> ComplexBallField(Integer(128))(Integer(1)).rising_
↪factorial(Integer(2)**Integer(64))
[2.34369112679686134...e+347382171305201285713 +/- ...]
>>> CBF(Integer(1)/Integer(2)).rising_factorial(CBF(Integer(2), Integer(3))) #_
↪abs tol 1e-15
[-0.123060451458124 +/- 3.06e-16] + [0.0406412631676552 +/- 7.57e-17]*I
```

Return a copy of this ball rounded to the precision of the parent.

It is possible to create balls whose midpoint is more precise than their parent's nominal precision (see [real_arb](#) for more information):

[illegible]

The `round()` method rounds such a ball to its parent's precision:

```
sage: b.round().mid()
↳needs sage.symbolic
0.5000000000000000 + 0.866025403784439*I
```

```
>>> from sage.all import *
>>> b.round().mid()
↳needs sage.symbolic
0.5000000000000000 + 0.866025403784439*I
```

See also

`trim()`

rsqrt (*analytic=False*)

Return the reciprocal square root of `self`.

If either the real or imaginary part is exactly zero, only a single real reciprocal square root is needed.

INPUT:

- `analytic` – boolean (default: `False`); if `True`, return an indeterminate (not-a-number) value when the input ball touches the branch cut

EXAMPLES:

```
sage: CBF(-2).rsqrt()
[-0.707106781186547 +/- ...e-16]*I
sage: CBF(-2).rsqrt(analytic=True)
nan + nan*I
sage: CBF(0, 1/2).rsqrt()
1.0000000000000000 - 1.0000000000000000*I
sage: CBF(0).rsqrt()
nan + nan*I
```

```
>>> from sage.all import *
>>> CBF(-Integer(2)).rsqrt()
[-0.707106781186547 +/- ...e-16]*I
>>> CBF(-Integer(2)).rsqrt(analytic=True)
nan + nan*I
>>> CBF(Integer(0), Integer(1)/Integer(2)).rsqrt()
1.0000000000000000 - 1.0000000000000000*I
>>> CBF(Integer(0)).rsqrt()
nan + nan*I
```

sec ()

Return the secant of this ball.

EXAMPLES:

```
sage: CBF(1, 1).sec()
[0.498337030555187 +/- ...e-16] + [0.591083841721045 +/- ...e-16]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).sec()
[0.498337030555187 +/- ...e-16] + [0.591083841721045 +/- ...e-16]*I
```

sech()

Return the hyperbolic secant of this ball.

EXAMPLES:

```
sage: CBF(pi/2, 1/10).sech() #_
↪needs sage.symbolic
[0.397174529918189 +/- ...e-16] + [-0.0365488656274242 +/- ...e-17]*I
```

```
>>> from sage.all import *
>>> CBF(pi/Integer(2), Integer(1)/Integer(10)).sech() #_
↪# needs sage.symbolic
[0.397174529918189 +/- ...e-16] + [-0.0365488656274242 +/- ...e-17]*I
```

sin()

Return the sine of this ball.

EXAMPLES:

```
sage: CBF(i*pi).sin() #_
↪needs sage.symbolic
[11.54873935725775 +/- ...e-15]*I
```

```
>>> from sage.all import *
>>> CBF(i*pi).sin() #_
↪needs sage.symbolic
[11.54873935725775 +/- ...e-15]*I
```

sin_integral()

Return the sine integral with argument `self`.

EXAMPLES:

```
sage: CBF(1, 1).Si() # abs tol 3e-15
[1.10422265823558 +/- 2.48e-15] + [0.88245380500792 +/- 3.36e-15]*I
sage: CBF(0).Si()
0
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).Si() # abs tol 3e-15
[1.10422265823558 +/- 2.48e-15] + [0.88245380500792 +/- 3.36e-15]*I
>>> CBF(Integer(0)).Si()
0
```

sinh()

Return the hyperbolic sine of this ball.

EXAMPLES:

```
sage: CBF(1, 1).sinh()
[0.634963914784736 +/- ...e-16] + [1.298457581415977 +/- ...e-16]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).sinh()
[0.634963914784736 +/- ...e-16] + [1.298457581415977 +/- ...e-16]*I
```

sinh_integral()

Return the hyperbolic sine integral with argument `self`.

EXAMPLES:

```
sage: CBF(1, 1).Shi() # abs tol 3e-15
[0.88245380500792 +/- 3.36e-15] + [1.10422265823558 +/- 2.48e-15]*I
sage: CBF(0).Shi()
0
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).Shi() # abs tol 3e-15
[0.88245380500792 +/- 3.36e-15] + [1.10422265823558 +/- 2.48e-15]*I
>>> CBF(Integer(0)).Shi()
0
```

spherical_harmonic(*phi*, *n*, *m*)

Return the spherical harmonic $Y_n^m(\theta, \phi)$ evaluated at θ given by `self`. In the current implementation, `n` and `m` must be small integers.

EXAMPLES:

```
sage: CBF(1+I).spherical_harmonic(1/2, -3, -2)
[0.80370071745224 +/- ...e-15] + [-0.07282031864711 +/- ...e-15]*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1)+I).spherical_harmonic(Integer(1)/Integer(2), -Integer(3), -
↳ Integer(2))
[0.80370071745224 +/- ...e-15] + [-0.07282031864711 +/- ...e-15]*I
```

sqrt(*analytic=False*)

Return the square root of this ball.

If either the real or imaginary part is exactly zero, only a single real square root is needed.

INPUT:

- `analytic` – boolean (default: `False`); if `True`, return an indeterminate (not-a-number) value when the input ball touches the branch cut

EXAMPLES:

```
sage: CBF(-2).sqrt()
[1.414213562373095 +/- ...e-16]*I
sage: CBF(-2).sqrt(analytic=True)
nan + nan*I
```

```
>>> from sage.all import *
>>> CBF(-Integer(2)).sqrt()
[1.414213562373095 +/- ...e-16]*I
>>> CBF(-Integer(2)).sqrt(analytic=True)
nan + nan*I
```

squash()

Return an exact ball with the same midpoint as this ball.

OUTPUT: a *ComplexBall*

EXAMPLES:

```
sage: mid = CBF(1/3, 1/10).squash()
sage: mid
[0.3333333333333333 +/- ...e-17] + [0.0999999999999999 +/- ...e-18]*I
sage: mid.parent()
Complex ball field with 53 bits of precision
sage: mid.is_exact()
True
```

```
>>> from sage.all import *
>>> mid = CBF(Integer(1)/Integer(3), Integer(1)/Integer(10)).squash()
>>> mid
[0.3333333333333333 +/- ...e-17] + [0.0999999999999999 +/- ...e-18]*I
>>> mid.parent()
Complex ball field with 53 bits of precision
>>> mid.is_exact()
True
```

See also

mid()

tan()

Return the tangent of this ball.

EXAMPLES:

```
sage: CBF(pi/2, 1/10).tan() #_
↪ needs sage.symbolic
[+/- ...e-14] + [10.03331113225399 +/- ...e-15]*I
sage: CBF(pi/2).tan() #_
↪ needs sage.symbolic
nan
```

```
>>> from sage.all import *
>>> CBF(pi/Integer(2), Integer(1)/Integer(10)).tan() _
↪ # needs sage.symbolic
[+/- ...e-14] + [10.03331113225399 +/- ...e-15]*I
>>> CBF(pi/Integer(2)).tan() _
↪ # needs sage.symbolic
nan
```

tanh()

Return the hyperbolic tangent of this ball.

EXAMPLES:

```
sage: CBF(1, 1).tanh()
[1.083923327338694 +/- ...e-16] + [0.2717525853195117 +/- ...e-17]*I
sage: CBF(0, pi/2).tanh()
↪needs sage.symbolic
nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).tanh()
[1.083923327338694 +/- ...e-16] + [0.2717525853195117 +/- ...e-17]*I
>>> CBF(Integer(0), pi/Integer(2)).tanh()
↪ # needs sage.symbolic
nan*I
```

trim()

Return a trimmed copy of this ball.

Return a copy of this ball with both the real and imaginary parts trimmed (see *trim()*).

EXAMPLES:

```
sage: b = CBF(1/3, RBF(1/3, rad=.01))
sage: b.mid()
0.333333333333333 + 0.333333333333333*I
sage: b.trim().mid()
0.333333333333333 + 0.333333015441895*I
```

```
>>> from sage.all import *
>>> b = CBF(Integer(1)/Integer(3), RBF(Integer(1)/Integer(3), rad=RealNumber(
↪'.01'))
>>> b.mid()
0.333333333333333 + 0.333333333333333*I
>>> b.trim().mid()
0.333333333333333 + 0.333333015441895*I
```

See also

round()

union(*other*)

Return a ball containing the convex hull of *self* and *other*.

EXAMPLES:

```
sage: b = CBF(1 + i).union(0)
sage: b.real().endpoints()
(-9.31322574615479e-10, 1.00000000093133)
```



```
>>> from sage.all import *
>>> b = CBF(Integer(1) + i).union(Integer(0))
>>> b.real().endpoints()
(-9.31322574615479e-10, 1.00000000093133)
```

zeta ($a=None$)

Return the image of this ball by the Hurwitz zeta function.

For $a = None$, this computes the Riemann zeta function.

EXAMPLES:

```
sage: CBF(1, 1).zeta()
[0.5821580597520036 +/- ...e-17] + [-0.9268485643308071 +/- ...e-17]*I
sage: CBF(1, 1).zeta(1)
[0.5821580597520036 +/- ...e-17] + [-0.9268485643308071 +/- ...e-17]*I
sage: CBF(1, 1).zeta(1/2)
[1.497919876084167 +/- ...e-16] + [0.2448655353684164 +/- ...e-17]*I
sage: CBF(1, 1).zeta(CBF(1, 1))
[-0.3593983122202835 +/- ...e-17] + [-2.875283329756940 +/- ...e-16]*I
sage: CBF(1, 1).zeta(-1)
nan + nan*I
```

```
>>> from sage.all import *
>>> CBF(Integer(1), Integer(1)).zeta()
[0.5821580597520036 +/- ...e-17] + [-0.9268485643308071 +/- ...e-17]*I
>>> CBF(Integer(1), Integer(1)).zeta(Integer(1))
[0.5821580597520036 +/- ...e-17] + [-0.9268485643308071 +/- ...e-17]*I
>>> CBF(Integer(1), Integer(1)).zeta(Integer(1)/Integer(2))
[1.497919876084167 +/- ...e-16] + [0.2448655353684164 +/- ...e-17]*I
>>> CBF(Integer(1), Integer(1)).zeta(CBF(Integer(1), Integer(1)))
[-0.3593983122202835 +/- ...e-17] + [-2.875283329756940 +/- ...e-16]*I
>>> CBF(Integer(1), Integer(1)).zeta(-Integer(1))
nan + nan*I
```

zetaderiv (k)

Return the image of this ball by the k -th derivative of the Riemann zeta function.

For a more flexible interface, see the low-level method `_zeta_series` of polynomials with complex ball coefficients.

EXAMPLES:

```
sage: CBF(1/2, 3).zetaderiv(1)
[0.191759884092721...] + [-0.073135728865928...]*I
sage: CBF(2).zetaderiv(3)
[-6.0001458028430...]
```

```
>>> from sage.all import *
>>> CBF(Integer(1)/Integer(2), Integer(3)).zetaderiv(Integer(1))
[0.191759884092721...] + [-0.073135728865928...]*I
>>> CBF(Integer(2)).zetaderiv(Integer(3))
[-6.0001458028430...]
```

```
class sage.rings.complex_arb.ComplexBallField(precision=53)
```

Bases: `UniqueRepresentation`, `ComplexBallField`

An approximation of the field of complex numbers using pairs of mid-rad intervals.

INPUT:

- `precision` – integer ≥ 2

EXAMPLES:

```
sage: CBF(1)
1.0000000000000000
```

```
>>> from sage.all import *
>>> CBF(Integer(1))
1.0000000000000000
```

See also

- `sage.rings.complex_arb`
- `sage.rings.complex_mpfr`
- `sage.rings.complex_mphi`
- `sage.rings.real_arb`

Element

alias of `ComplexBall`

`characteristic()`

Complex ball fields have characteristic zero.

EXAMPLES:

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

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

`complex_field()`

Return the complex ball field with the same precision, i.e. `self`.

EXAMPLES:

```
sage: CBF.complex_field() is CBF
True
```

```
>>> from sage.all import *
>>> CBF.complex_field() is CBF
True
```

construction()

Return the construction of a complex ball field as the algebraic closure of the real ball field with the same precision.

EXAMPLES:

```
sage: functor, base = CBF.construction()
sage: functor, base
(AlgebraicClosureFunctor, Real ball field with 53 bits of precision)
sage: functor(base) is CBF
True
```

```
>>> from sage.all import *
>>> functor, base = CBF.construction()
>>> functor, base
(AlgebraicClosureFunctor, Real ball field with 53 bits of precision)
>>> functor(base) is CBF
True
```

gen(i)

For $i = 0$, return the imaginary unit in this complex ball field.

EXAMPLES:

```
sage: CBF.0
1.0000000000000000*I
sage: CBF.gen(1)
Traceback (most recent call last):
...
ValueError: only one generator
```

```
>>> from sage.all import *
>>> CBF.gen(0)
1.0000000000000000*I
>>> CBF.gen(Integer(1))
Traceback (most recent call last):
...
ValueError: only one generator
```

gens()

Return the tuple of generators of this complex ball field, i.e. $(i,)$.

EXAMPLES:

```
sage: CBF.gens()
(1.0000000000000000*I,)
sage: CBF.gens_dict()
{'1.0000000000000000*I': 1.0000000000000000*I}
```

```
>>> from sage.all import *
>>> CBF.gens()
(1.0000000000000000*I,)
>>> CBF.gens_dict()
{'1.0000000000000000*I': 1.0000000000000000*I}
```

integral (*func*, *a*, *b*, *params=None*, *rel_tol=None*, *abs_tol=None*, *deg_limit=None*, *eval_limit=None*, *depth_limit=None*, *use_heap=None*, *verbose=None*)

Compute a rigorous enclosure of the integral of *func* on the interval $[a, b]$.

INPUT:

- *func* – a callable object accepting two parameters, a complex ball *x* and a boolean flag *analytic*, and returning an element of this ball field (or some value that coerces into this ball field), such that:
 - *func*(*x*, *False*) evaluates the integrand *f* on the ball *x*. There are no restrictions on the behavior of *f* on *x*; in particular, it can be discontinuous.
 - *func*(*x*, *True*) evaluates *f*(*x*) if *f* is analytic on the whole *x*, and returns some non-finite ball (e.g., *self*(NaN)) otherwise.

(The *analytic* flag only needs to be checked for integrands that are non-analytic but bounded in some regions, typically complex functions with branch cuts, like $\sqrt{z}$. In particular, it can be ignored for meromorphic functions.)

- *a*, *b* – integration bounds. The bounds can be real or complex balls, or elements of any parent that coerces into this ball field, e.g. rational or algebraic numbers.
- *rel_tol* – relative accuracy goal (default: 2^{-p} where *p* is the precision of the ball field)
- *abs_tol* – absolute accuracy goal (default: 2^{-p} where *p* is the precision of the ball field)

Additionally, the following optional parameters can be used to control the integration algorithm. See the [FLINT documentation](#) for more information.

- *deg_limit* – maximum quadrature degree for each subinterval
- *eval_limit* – maximum number of function evaluations
- *depth_limit* – maximum search depth for adaptive subdivision
- *use_heap* – boolean (default: *False*); if *True*, use a priority queue instead of a stack to manage subintervals. This sometimes gives better results for integrals with slow convergence but may require more memory and increasing *depth_limit*.
- *verbose* – integer (default: 0); if set to 1, some information about the overall integration process is printed to standard output. If set to 2, information about each subinterval is printed.

EXAMPLES:

Some analytic integrands:

```
sage: CBF.integral(lambda x, _: x, 0, 1)
[0.5000000000000000 +/- ...e-16]

sage: CBF.integral(lambda x, _: x.gamma(), 1 - CBF(i), 1 + CBF(i)) # abs tol
↪ 1e-13
[+/- 1.39e-15] + [1.57239266949806 +/- 8.33e-15]*I

sage: C = ComplexBallField(100)
sage: C.integral(lambda x, _: x.cos() * x.sin(), 0, 1)
[0.35403670913678559674939205737 +/- ...e-30]

sage: CBF.integral(lambda x, _: (x + x.exp()).sin(), 0, 8)
[0.34740017266 +/- ...e-12]

sage: C = ComplexBallField(2000)
```

(continues on next page)

(continued from previous page)

```
sage: C.integral(lambda x, _: (x + x.exp()).sin(), 0, 8) # long time
[0.34740017...55347713 +/- ...e-598]
```

```
>>> from sage.all import *
>>> CBF.integral(lambda x, _: x, Integer(0), Integer(1))
[0.5000000000000000 +/- ...e-16]

>>> CBF.integral(lambda x, _: x.gamma(), Integer(1) - CBF(i), Integer(1) +
↳ CBF(i)) # abs tol 1e-13
[+/- 1.39e-15] + [1.57239266949806 +/- 8.33e-15]*I

>>> C = ComplexBallField(Integer(100))
>>> C.integral(lambda x, _: x.cos() * x.sin(), Integer(0), Integer(1))
[0.35403670913678559674939205737 +/- ...e-30]

>>> CBF.integral(lambda x, _: (x + x.exp()).sin(), Integer(0), Integer(8))
[0.34740017266 +/- ...e-12]

>>> C = ComplexBallField(Integer(2000))
>>> C.integral(lambda x, _: (x + x.exp()).sin(), Integer(0), Integer(8)) #
↳ long time
[0.34740017...55347713 +/- ...e-598]
```

Here the integration path crosses the branch cut of the square root:

```
sage: def my_sqrt(z, analytic):
....:     if (analytic and not z.real() > 0
....:         and z.imag().contains_zero()):
....:         return CBF(NaN)
....:     else:
....:         return z.sqrt()
sage: CBF.integral(my_sqrt, -1 + CBF(i), -1 - CBF(i)) #
↳ needs sage.symbolic
[+/- ...e-14] + [-0.4752076627926 +/- 5...e-14]*I
```

```
>>> from sage.all import *
>>> def my_sqrt(z, analytic):
...     if (analytic and not z.real() > Integer(0)
...         and z.imag().contains_zero()):
...         return CBF(NaN)
...     else:
...         return z.sqrt()
>>> CBF.integral(my_sqrt, -Integer(1) + CBF(i), -Integer(1) - CBF(i))
↳ # needs sage.symbolic
[+/- ...e-14] + [-0.4752076627926 +/- 5...e-14]*I
```

Note, though, that proper handling of the `analytic` flag is required even when the path does not touch the branch cut:

```
sage: correct = CBF.integral(my_sqrt, 1, 2); correct
[1.21895141649746 +/- ...e-15]
sage: RBF(integral(sqrt(x), x, 1, 2)) # long time #
```

(continues on next page)

(continued from previous page)

```

↪needs sage.symbolic
[1.21895141649746 +/- ...e-15]
sage: wrong = CBF.integral(lambda z, _: z.sqrt(), 1, 2) # WRONG!
sage: correct - wrong
[-5.640636259e-5 +/- ...e-15]

```

```

>>> from sage.all import *
>>> correct = CBF.integral(my_sqrt, Integer(1), Integer(2)); correct
[1.21895141649746 +/- ...e-15]
>>> RBF(integral(sqrt(x), x, Integer(1), Integer(2))) # long time ↵
↪ # needs sage.symbolic
[1.21895141649746 +/- ...e-15]
>>> wrong = CBF.integral(lambda z, _: z.sqrt(), Integer(1), Integer(2)) # ↵
↪ WRONG!
>>> correct - wrong
[-5.640636259e-5 +/- ...e-15]

```

We can integrate the real absolute value function by defining a piecewise holomorphic extension:

```

sage: def real_abs(z, analytic):
....:     if z.real().contains_zero():
....:         if analytic:
....:             return z.parent() (NaN)
....:         else:
....:             return z.union(-z)
....:     elif z.real() > 0:
....:         return z
....:     else:
....:         return -z
sage: CBF.integral(real_abs, -1, 1)
[1.000000000000...]
sage: CBF.integral(lambda z, analytic: real_abs(z.sin(), analytic), 0, 2*CBF.
↪pi())
[4.000000000000...]

```

```

>>> from sage.all import *
>>> def real_abs(z, analytic):
...     if z.real().contains_zero():
...         if analytic:
...             return z.parent() (NaN)
...         else:
...             return z.union(-z)
...     elif z.real() > Integer(0):
...         return z
...     else:
...         return -z
>>> CBF.integral(real_abs, -Integer(1), Integer(1))
[1.000000000000...]
>>> CBF.integral(lambda z, analytic: real_abs(z.sin(), analytic), Integer(0), ↵
↪Integer(2)*CBF.pi())
[4.000000000000...]

```

Some methods of complex balls natively support the `analytic` flag:

```
sage: CBF.integral(lambda z, analytic: z.log(analytic=analytic),
....:             -1-CBF(i), -1+CBF(i))
[+/- ...e-14] + [0.26394350735484 +/- ...e-15]*I
sage: from sage.rings.complex_arb import ComplexBall
sage: CBF.integral(ComplexBall.sqrt, -1+CBF(i), -1-CBF(i))
[+/- ...e-14] + [-0.4752076627926 +/- 5...e-14]*I
```

```
>>> from sage.all import *
>>> CBF.integral(lambda z, analytic: z.log(analytic=analytic),
...             -Integer(1)-CBF(i), -Integer(1)+CBF(i))
[+/- ...e-14] + [0.26394350735484 +/- ...e-15]*I
>>> from sage.rings.complex_arb import ComplexBall
>>> CBF.integral(ComplexBall.sqrt, -Integer(1)+CBF(i), -Integer(1)-CBF(i))
[+/- ...e-14] + [-0.4752076627926 +/- 5...e-14]*I
```

Here the integrand has a pole on or very close to the integration path, but there is no need to explicitly handle the `analytic` flag since the integrand is unbounded:

```
sage: CBF.integral(lambda x, _: 1/x, -1, 1)
nan + nan*I
sage: CBF.integral(lambda x, _: 1/x, 10^-1000, 1)
nan + nan*I
sage: CBF.integral(lambda x, _: 1/x, 10^-1000, 1, abs_tol=1e-10)
[2302.5850930 +/- ...e-8]
```

```
>>> from sage.all import *
>>> CBF.integral(lambda x, _: Integer(1)/x, -Integer(1), Integer(1))
nan + nan*I
>>> CBF.integral(lambda x, _: Integer(1)/x, Integer(10)**-Integer(1000),
↪Integer(1))
nan + nan*I
>>> CBF.integral(lambda x, _: Integer(1)/x, Integer(10)**-Integer(1000),
↪Integer(1), abs_tol=RealNumber('1e-10'))
[2302.5850930 +/- ...e-8]
```

Tolerances:

```
sage: CBF.integral(lambda x, _: x.exp(), -1020, -1010)
[+/- ...e-438]
sage: CBF.integral(lambda x, _: x.exp(), -1020, -1010, abs_tol=1e-450)
[2.304377150950e-439 +/- ...e-452]
sage: CBF.integral(lambda x, _: x.exp(), -1020, -1010, abs_tol=0)
[2.304377150950e-439 +/- 7...e-452]
sage: CBF.integral(lambda x, _: x.exp(), -1020, -1010, rel_tol=1e-2, abs_
↪tol=0)
[2.3044e-439 +/- ...e-444]

sage: epsi = CBF(1e-10)
sage: CBF.integral(lambda x, _: x*(1/x).sin(), epsi, 1)
[0.38 +/- ...e-3]
sage: CBF.integral(lambda x, _: x*(1/x).sin(), epsi, 1, use_heap=True)
[0.37853002 +/- ...e-9]
```

```

>>> from sage.all import *
>>> CBF.integral(lambda x, _: x.exp(), -Integer(1020), -Integer(1010))
[+/- ...e-438]
>>> CBF.integral(lambda x, _: x.exp(), -Integer(1020), -Integer(1010), abs_
↳tol=RealNumber('1e-450'))
[2.304377150950e-439 +/- ...e-452]
>>> CBF.integral(lambda x, _: x.exp(), -Integer(1020), -Integer(1010), abs_
↳tol=Integer(0))
[2.304377150950e-439 +/- 7...e-452]
>>> CBF.integral(lambda x, _: x.exp(), -Integer(1020), -Integer(1010), rel_
↳tol=RealNumber('1e-2'), abs_tol=Integer(0))
[2.3044e-439 +/- ...e-444]

>>> epsi = CBF(RealNumber('1e-10'))
>>> CBF.integral(lambda x, _: x*(Integer(1)/x).sin(), epsi, Integer(1))
[0.38 +/- ...e-3]
>>> CBF.integral(lambda x, _: x*(Integer(1)/x).sin(), epsi, Integer(1), use_
↳heap=True)
[0.37853002 +/- ...e-9]

```

ALGORITHM:

Uses the `acb_calc` module of the FLINT library.

is_exact()

Complex ball fields are not exact.

EXAMPLES:

```

sage: ComplexBallField().is_exact()
False

```

```

>>> from sage.all import *
>>> ComplexBallField().is_exact()
False

```

ngens()

Return 1 as the only generator is the imaginary unit.

EXAMPLES:

```

sage: CBF.ngens()
1

```

```

>>> from sage.all import *
>>> CBF.ngens()
1

```

pi()

Return a ball enclosing π .

EXAMPLES:


```
sage: CBF.pi()
[3.141592653589793 +/- ...e-16]
sage: ComplexBallField(128).pi()
[3.1415926535897932384626433832795028842 +/- ...e-38]

sage: CBF.pi().parent()
Complex ball field with 53 bits of precision
```

```
>>> from sage.all import *
>>> CBF.pi()
[3.141592653589793 +/- ...e-16]
>>> ComplexBallField(Integer(128)).pi()
[3.1415926535897932384626433832795028842 +/- ...e-38]

>>> CBF.pi().parent()
Complex ball field with 53 bits of precision
```

prec()

Return the bit precision used for operations on elements of this field.

EXAMPLES:

```
sage: ComplexBallField().precision()
53
```

```
>>> from sage.all import *
>>> ComplexBallField().precision()
53
```

precision()

Return the bit precision used for operations on elements of this field.

EXAMPLES:

```
sage: ComplexBallField().precision()
53
```

```
>>> from sage.all import *
>>> ComplexBallField().precision()
53
```

some_elements()

Complex ball fields contain elements with exact, inexact, infinite, or undefined real and imaginary parts.

EXAMPLES:

```
sage: CBF.some_elements()
[1.000000000000000,
 -0.500000000000000*I,
 1.000000000000000 + [0.333333333333333 +/- ...e-17]*I,
 [-0.333333333333333 +/- ...e-17] + 0.250000000000000*I,
 [-2.175556475109056e+181961467118333366510562 +/- ...
 ↪e+181961467118333366510545],
```

(continues on next page)

(continued from previous page)

```
[+/- inf],
[0.3333333333333333 +/- ...e-17] + [+/- inf]*I,
[+/- inf] + [+/- inf]*I,
nan,
nan + nan*I,
[+/- inf] + nan*I]
```

```
>>> from sage.all import *
>>> CBF.some_elements()
[1.0000000000000000,
 -0.5000000000000000*I,
 1.0000000000000000 + [0.3333333333333333 +/- ...e-17]*I,
 [-0.3333333333333333 +/- ...e-17] + 0.2500000000000000*I,
 [-2.175556475109056e+181961467118333366510562 +/- ...
  e+181961467118333366510545],
 [+/- inf],
 [0.3333333333333333 +/- ...e-17] + [+/- inf]*I,
 [+/- inf] + [+/- inf]*I,
 nan,
 nan + nan*I,
 [+/- inf] + nan*I]
```

zeta_zeros (*count*, *start=1*)

Compute consecutive zeros of the Riemann zeta function.

INPUT:

- *count* – positive integer; number of zeros to be computed, must fit in a machine integer
- *start* – positive integer (default: 1); index of the first zero to be computed

OUTPUT:

A list of *count* consecutive zeros of the Riemann zeta function, starting from the *start*-th zero. Indexing starts at one, following usual mathematical notations.

EXAMPLES:

```
sage: CBF.zeta_zeros(10)
[0.5000000000000000 + [14.134725141734...]*I,
 0.5000000000000000 + [21.0220396387715...]*I,
 0.5000000000000000 + [25.010857580145...]*I,
 0.5000000000000000 + [30.4248761258595...]*I,
 0.5000000000000000 + [32.935061587739...]*I,
 0.5000000000000000 + [37.586178158825...]*I,
 0.5000000000000000 + [40.918719012147...]*I,
 0.5000000000000000 + [43.32707328091...]*I,
 0.5000000000000000 + [48.005150881167...]*I,
 0.5000000000000000 + [49.773832477672...]*I]

sage: CBF.zeta_zeros(6, start=5)
[0.5000000000000000 + [32.935061587739...]*I,
 0.5000000000000000 + [37.586178158825...]*I,
 0.5000000000000000 + [40.918719012147...]*I,
 0.5000000000000000 + [43.32707328091...]*I,
```

(continues on next page)

(continued from previous page)

```
0.5000000000000000 + [48.005150881167...]*I,
0.5000000000000000 + [49.773832477672...]*I]
```

```
>>> from sage.all import *
>>> CBF.zeta_zeros(Integer(10))
[0.5000000000000000 + [14.134725141734...]*I,
0.5000000000000000 + [21.0220396387715...]*I,
0.5000000000000000 + [25.010857580145...]*I,
0.5000000000000000 + [30.4248761258595...]*I,
0.5000000000000000 + [32.935061587739...]*I,
0.5000000000000000 + [37.586178158825...]*I,
0.5000000000000000 + [40.918719012147...]*I,
0.5000000000000000 + [43.32707328091...]*I,
0.5000000000000000 + [48.005150881167...]*I,
0.5000000000000000 + [49.773832477672...]*I]

>>> CBF.zeta_zeros(Integer(6), start=Integer(5))
[0.5000000000000000 + [32.935061587739...]*I,
0.5000000000000000 + [37.586178158825...]*I,
0.5000000000000000 + [40.918719012147...]*I,
0.5000000000000000 + [43.32707328091...]*I,
0.5000000000000000 + [48.005150881167...]*I,
0.5000000000000000 + [49.773832477672...]*I]
```

```
class sage.rings.complex_arb.IntegrationContext
```

Bases: object

Used to wrap the integrand and hold some context information during numerical integration.

EXACT REAL ARITHMETIC

3.1 Lazy real and complex numbers

These classes are very lazy, in the sense that it doesn't really do anything but simply sits between exact rings of characteristic 0 and the real numbers. The values are actually computed when they are cast into a field of fixed precision.

The main purpose of these classes is to provide a place for exact rings (e.g. number fields) to embed for the coercion model (as only one embedding can be specified in the forward direction).

```
sage.rings.real_lazy.ComplexLazyField()
```

Return the lazy complex field.

EXAMPLES:

There is only one lazy complex field:

```
sage: ComplexLazyField() is ComplexLazyField()
True
```

```
>>> from sage.all import *
>>> ComplexLazyField() is ComplexLazyField()
True
```

```
class sage.rings.real_lazy.ComplexLazyField_class
```

Bases: *LazyField*

This class represents the set of complex numbers to unspecified precision. For the most part it simply wraps exact elements and defers evaluation until a specified precision is requested.

For more information, see the documentation of the [RLF](#).

EXAMPLES:

[illegible]

```
>>> from sage.all import *
>>> a = CLF(-Integer(1)).sqrt()
>>> a
```

(continues on next page)

(continued from previous page)

[illegible]

```
construction()
```

Return the functorial construction of `self`, namely, algebraic closure of the real lazy field.

EXAMPLES:

```
sage: c, S = CLF.construction(); S
Real Lazy Field
sage: CLF == c(S)
True
```

```
>>> from sage.all import *
>>> c, S = CLF.construction(); S
Real Lazy Field
>>> CLF == c(S)
True
```

qen ($i=0$)

Return the i -th generator of `self`.

EXAMPLES:

[illegible][illegible]

```
interval_field (prec=None)
```

Return the interval field that represents the same mathematical field as `self`.

EXAMPLES:

```
sage: CLF.interval_field()
Complex Interval Field with 53 bits of precision
sage: CLF.interval_field(333)
↳needs sage.rings.complex_interval_field
Complex Interval Field with 333 bits of precision
sage: CLF.interval_field() is CIF
True
```

```

>>> from sage.all import *
>>> CLF.interval_field()
Complex Interval Field with 53 bits of precision
>>> CLF.interval_field(Integer(333))
↳ # needs sage.rings.complex_interval_field
Complex Interval Field with 333 bits of precision
>>> CLF.interval_field() is CIF
True

```

class sage.rings.real_lazy.LazyAlgebraic

Bases: *LazyFieldElement*

This represents an algebraic number, specified by a polynomial over $\mathbf{Q}$ and a real or complex approximation.

EXAMPLES:

```

sage: x = polygen(QQ)
sage: from sage.rings.real_lazy import LazyAlgebraic
sage: a = LazyAlgebraic(RLF, x^2-2, 1.5)
sage: a
1.414213562373095?

```

```

>>> from sage.all import *
>>> x = polygen(QQ)
>>> from sage.rings.real_lazy import LazyAlgebraic
>>> a = LazyAlgebraic(RLF, x**Integer(2)-Integer(2), RealNumber('1.5'))
>>> a
1.414213562373095?

```

eval(R)

Convert self into an element of R .

EXAMPLES:

```

sage: from sage.rings.real_lazy import LazyAlgebraic
sage: a = LazyAlgebraic(CLF, QQ['x'].cyclotomic_polynomial(7), 0.6+0.8*CC.0)
sage: a
0.6234898018587335? + 0.7818314824680299?I
sage: ComplexField(150)(a) # indirect doctest
↳ needs sage.rings.number_field
0.6234898018587335052500488400423981063227473 + 0.
↳ 78183148246802980870844452667405775023233452*I

sage: a = LazyAlgebraic(CLF, QQ['x'].0^2-7, -2.0)
sage: RR(a)
↳ needs sage.rings.number_field
-2.64575131106459
sage: RR(a)^2
↳ needs sage.rings.number_field
7.000000000000000

```

```

>>> from sage.all import *
>>> from sage.rings.real_lazy import LazyAlgebraic
>>> a = LazyAlgebraic(CLF, QQ['x'].cyclotomic_polynomial(Integer(7)),

```

(continues on next page)

(continued from previous page)

```

↪RealNumber('0.6')+RealNumber('0.8')*CC.gen(0))
>>> a
0.6234898018587335? + 0.7818314824680299*I
>>> ComplexField(Integer(150))(a) # indirect doctest
↪ # needs sage.rings.number_field
0.62348980185873353052500488400423981063227473 + 0.
↪78183148246802980870844452667405775023233452*I

>>> a = LazyAlgebraic(CLF, QQ['x'].gen(0)**Integer(2)-Integer(7), -RealNumber(
↪'2.0'))
>>> RR(a) #
↪needs sage.rings.number_field
-2.64575131106459
>>> RR(a)**Integer(2)
↪ # needs sage.rings.number_field
7.000000000000000

```

```
class sage.rings.real_lazy.LazyBinop
```

Bases: *LazyFieldElement*

A lazy element representing a binary (usually arithmetic) operation between two other lazy elements.

EXAMPLES:

[illegible][illegible]

depth ()

Return the depth of `self` as an arithmetic expression.

This is the maximum number of dependent intermediate expressions when evaluating `self`, and is used to determine the precision needed to get the final result to the desired number of bits.

It is equal to the maximum of the right and left depths, plus one.

EXAMPLES:

```
sage: from sage.rings.real_lazy import LazyBinop
sage: a = LazyBinop(RLF, 6, 8, operator.mul)
sage: a.depth()
1
sage: b = LazyBinop(RLF, 2, a, operator.sub)
```

(continues on next page)

(continued from previous page)

```
sage: b.depth()
2
```

```
>>> from sage.all import *
>>> from sage.rings.real_lazy import LazyBinop
>>> a = LazyBinop(RLF, Integer(6), Integer(8), operator.mul)
>>> a.depth()
1
>>> b = LazyBinop(RLF, Integer(2), a, operator.sub)
>>> b.depth()
2
```

eval(R)

Convert the operands to elements of R , then perform the operation on them.

EXAMPLES:

```
sage: from sage.rings.real_lazy import LazyBinop
sage: a = LazyBinop(RLF, 6, 8, operator.add)
sage: a.eval(RR)
14.000000000000000
```

```
>>> from sage.all import *
>>> from sage.rings.real_lazy import LazyBinop
>>> a = LazyBinop(RLF, Integer(6), Integer(8), operator.add)
>>> a.eval(RR)
14.000000000000000
```

A bit absurd:

```
sage: a.eval(str)
'68'
```

```
>>> from sage.all import *
>>> a.eval(str)
'68'
```

class `sage.rings.real_lazy.LazyConstant`

Bases: `LazyFieldElement`

This class represents a real or complex constant (such as π or i).

eval(R)

Convert `self` into an element of R .

EXAMPLES:

```
sage: from sage.rings.real_lazy import LazyConstant
sage: a = LazyConstant(RLF, 'e')
sage: RDF(a) # indirect doctest
2.718281828459045
sage: a = LazyConstant(CLF, 'I')
sage: CC(a)
1.000000000000000*I
```

```

>>> from sage.all import *
>>> from sage.rings.real_lazy import LazyConstant
>>> a = LazyConstant(RLF, 'e')
>>> RDF(a) # indirect doctest
2.718281828459045
>>> a = LazyConstant(CLF, 'I')
>>> CC(a)
1.000000000000000*I

```

class sage.rings.real_lazy.LazyField

Bases: [Field](#)

The base class for lazy real fields.

Warning

LazyField uses `__getattr__()`, to implement:

```

sage: CLF.pi
3.141592653589794?

```

```

>>> from sage.all import *
>>> CLF.pi
3.141592653589794?

```

I (NT, 20/04/2012) did not manage to have `__getattr__` call `Parent.__getattr__()` in case of failure; hence we can't use this `__getattr__` trick for extension types to recover the methods from categories. Therefore, at this point, no concrete subclass of this class should be an extension type (which is probably just fine):

```

sage: RLF.__class__
<class 'sage.rings.real_lazy.RealLazyField_class_with_category'>
sage: CLF.__class__
<class 'sage.rings.real_lazy.ComplexLazyField_class_with_category'>

```

```

>>> from sage.all import *
>>> RLF.__class__
<class 'sage.rings.real_lazy.RealLazyField_class_with_category'>
>>> CLF.__class__
<class 'sage.rings.real_lazy.ComplexLazyField_class_with_category'>

```

Element

alias of [LazyWrapper](#)

algebraic_closure()

Return the algebraic closure of `self`, i.e., the complex lazy field.

EXAMPLES:

```

sage: RLF.algebraic_closure()
Complex Lazy Field

sage: CLF.algebraic_closure()
Complex Lazy Field

```

```
>>> from sage.all import *
>>> RLF.algebraic_closure()
Complex Lazy Field

>>> CLF.algebraic_closure()
Complex Lazy Field
```

interval_field(*prec=None*)

Abstract method to create the corresponding interval field.

class sage.rings.real_lazy.**LazyFieldElement**

Bases: `FieldElement`

approx()

Return self as an element of an interval field.

EXAMPLES:

```
sage: CLF(1/6).approx()
0.16666666666666667?
sage: CLF(1/6).approx().parent()
Complex Interval Field with 53 bits of precision
```

```
>>> from sage.all import *
>>> CLF(Integer(1)/Integer(6)).approx()
0.16666666666666667?
>>> CLF(Integer(1)/Integer(6)).approx().parent()
Complex Interval Field with 53 bits of precision
```

When the absolute value is involved, the result might be real:

```
sage: # needs sage.symbolic
sage: z = exp(CLF(1 + I/2)); z
2.38551673095914? + 1.303213729686996?*I
sage: r = z.abs(); r
2.71828182845905?
sage: parent(z.approx())
Complex Interval Field with 53 bits of precision
sage: parent(r.approx())
Real Interval Field with 53 bits of precision
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> z = exp(CLF(Integer(1) + I/Integer(2))); z
2.38551673095914? + 1.303213729686996?*I
>>> r = z.abs(); r
2.71828182845905?
>>> parent(z.approx())
Complex Interval Field with 53 bits of precision
>>> parent(r.approx())
Real Interval Field with 53 bits of precision
```

continued_fraction()

Return the continued fraction of self.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: a = RLF(sqrt(2)) + RLF(sqrt(3))
sage: cf = a.continued_fraction()
sage: cf
[3; 6, 1, 5, 7, 1, 1, 4, 1, 38, 43, 1, 3, 2, 1, 1, 1, 1, 2, 4, ...]
sage: cf.convergent(100)
444927297812646558239761867973501208151173610180916865469/
↪141414466649174973335183571854340329919207428365474086063
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> a = RLF(sqrt(Integer(2))) + RLF(sqrt(Integer(3)))
>>> cf = a.continued_fraction()
>>> cf
[3; 6, 1, 5, 7, 1, 1, 4, 1, 38, 43, 1, 3, 2, 1, 1, 1, 1, 2, 4, ...]
>>> cf.convergent(Integer(100))
444927297812646558239761867973501208151173610180916865469/
↪141414466649174973335183571854340329919207428365474086063
```

depth()

Abstract method for returning the depth of `self` as an arithmetic expression.

This is the maximum number of dependent intermediate expressions when evaluating `self`, and is used to determine the precision needed to get the final result to the desired number of bits.

It is equal to the maximum of the right and left depths, plus one.

EXAMPLES:

```
sage: from sage.rings.real_lazy import LazyBinop
sage: a = LazyBinop(RLF, 6, 8, operator.mul)
sage: a.depth()
1
```

```
>>> from sage.all import *
>>> from sage.rings.real_lazy import LazyBinop
>>> a = LazyBinop(RLF, Integer(6), Integer(8), operator.mul)
>>> a.depth()
1
```

eval(R)

Abstract method for converting `self` into an element of `R`.

EXAMPLES:

```
sage: a = RLF(12)
sage: a.eval(ZZ)
12
```

```
>>> from sage.all import *
>>> a = RLF(Integer(12))
>>> a.eval(ZZ)
12
```

class sage.rings.real_lazy.LazyNamedUnop

Bases: *LazyUnop*

This class is used to represent the many named methods attached to real numbers, and is instantiated by the `__getattr__` method of *LazyElements*.

EXAMPLES:

```
sage: from sage.rings.real_lazy import LazyNamedUnop
sage: a = LazyNamedUnop(RLF, 1, 'arcsin')
sage: RR(a)
1.57079632679490
sage: a = LazyNamedUnop(RLF, 9, 'log', extra_args=(3,))
sage: RR(a)
2.000000000000000
```

```
>>> from sage.all import *
>>> from sage.rings.real_lazy import LazyNamedUnop
>>> a = LazyNamedUnop(RLF, Integer(1), 'arcsin')
>>> RR(a)
1.57079632679490
>>> a = LazyNamedUnop(RLF, Integer(9), 'log', extra_args=(Integer(3),))
>>> RR(a)
2.000000000000000
```

approx()

Do something reasonable with functions that are not defined on the interval fields.

eval(*R*)

Convert *self* into an element of *R*.

class sage.rings.real_lazy.LazyUnop

Bases: *LazyFieldElement*

Represent a unevaluated single function of one variable.

EXAMPLES:

```
sage: from sage.rings.real_lazy import LazyUnop
sage: a = LazyUnop(RLF, 3, sqrt); a
1.732050807568878?
sage: a._arg
3
sage: a._op
<function sqrt at ...>
sage: Reals(100)(a)
1.7320508075688772935274463415
sage: Reals(100)(a)^2
3.0000000000000000000000000000000
```

```
>>> from sage.all import *
>>> from sage.rings.real_lazy import LazyUnop
>>> a = LazyUnop(RLF, Integer(3), sqrt); a
1.732050807568878?
>>> a._arg
```

(continues on next page)

(continued from previous page)

```

3
>>> a._op
<function sqrt at ...>
>>> Reals(Integer(100))(a)
1.7320508075688772935274463415
>>> Reals(Integer(100))(a)**Integer(2)
3.00000000000000000000000000000000

```

depth()

Return the depth of `self` as an arithmetic expression.

This is the maximum number of dependent intermediate expressions when evaluating `self`, and is used to determine the precision needed to get the final result to the desired number of bits.

It is equal to one more than the depth of its operand.

EXAMPLES:

```

sage: from sage.rings.real_lazy import LazyUnop
sage: a = LazyUnop(RLF, 3, sqrt)
sage: a.depth()
1
sage: b = LazyUnop(RLF, a, sin)
sage: b.depth()
2

```

```

>>> from sage.all import *
>>> from sage.rings.real_lazy import LazyUnop
>>> a = LazyUnop(RLF, Integer(3), sqrt)
>>> a.depth()
1
>>> b = LazyUnop(RLF, a, sin)
>>> b.depth()
2

```

eval(R)

Convert `self` into an element of `R`.

EXAMPLES:

```

sage: from sage.rings.real_lazy import LazyUnop
sage: a = LazyUnop(RLF, 3, sqrt)
sage: a.eval(ZZ)                                     #_
↪needs sage.symbolic
sqrt(3)

```

```

>>> from sage.all import *
>>> from sage.rings.real_lazy import LazyUnop
>>> a = LazyUnop(RLF, Integer(3), sqrt)
>>> a.eval(ZZ)                                     #_
↪needs sage.symbolic
sqrt(3)

```

class sage.rings.real_lazy.LazyWrapper

Bases: *LazyFieldElement*

A lazy element that simply wraps an element of another ring.

EXAMPLES:

```
sage: from sage.rings.real_lazy import LazyWrapper
sage: a = LazyWrapper(RLF, 3)
sage: a._value
3
```

```
>>> from sage.all import *
>>> from sage.rings.real_lazy import LazyWrapper
>>> a = LazyWrapper(RLF, Integer(3))
>>> a._value
3
```

continued_fraction()

Return the continued fraction of *self*.

EXAMPLES:

```
sage: a = RLF(sqrt(2)) #_
↳needs sage.symbolic
sage: a.continued_fraction() #_
↳needs sage.symbolic
[1; 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]
```

```
>>> from sage.all import *
>>> a = RLF(sqrt(Integer(2))) #_
↳ # needs sage.symbolic
>>> a.continued_fraction() #_
↳needs sage.symbolic
[1; 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]
```

depth()

Return the depth of *self* as an expression, which is always 0.

EXAMPLES:

```
sage: RLF(4).depth()
0
```

```
>>> from sage.all import *
>>> RLF(Integer(4)).depth()
0
```

eval(*R*)

Convert *self* into an element of *R*.

EXAMPLES:

```
sage: a = RLF(12)
sage: a.eval(ZZ)
```

(continues on next page)

(continued from previous page)

```
12
sage: a.eval(ZZ).parent()
Integer Ring
```

```
>>> from sage.all import *
>>> a = RLF(Integer(12))
>>> a.eval(ZZ)
12
>>> a.eval(ZZ).parent()
Integer Ring
```

class sage.rings.real_lazy.LazyWrapperMorphism

Bases: Morphism

This morphism coerces elements from anywhere into lazy rings by creating a wrapper element (as fast as possible).

EXAMPLES:

```
sage: from sage.rings.real_lazy import LazyWrapperMorphism
sage: f = LazyWrapperMorphism(QQ, RLF)
sage: a = f(3); a
3
sage: type(a)
<class 'sage.rings.real_lazy.LazyWrapper'>
sage: a._value
3
sage: a._value.parent()
Rational Field
```

```
>>> from sage.all import *
>>> from sage.rings.real_lazy import LazyWrapperMorphism
>>> f = LazyWrapperMorphism(QQ, RLF)
>>> a = f(Integer(3)); a
3
>>> type(a)
<class 'sage.rings.real_lazy.LazyWrapper'>
>>> a._value
3
>>> a._value.parent()
Rational Field
```

sage.rings.real_lazy.RealLazyField()

Return the lazy real field.

EXAMPLES:

There is only one lazy real field:

```
sage: RealLazyField() is RealLazyField()
True
```

```
>>> from sage.all import *
>>> RealLazyField() is RealLazyField()
True
```


EXAMPLES:

```
sage: c, S = RLF.construction(); S
Rational Field
sage: RLF == c(S)
True
```

```
>>> from sage.all import *
>>> c, S = RLF.construction(); S
Rational Field
>>> RLF == c(S)
True
```

gen(*i=0*)Return the *i*-th generator of self.

EXAMPLES:

```
sage: RLF.gen()
1
```

```
>>> from sage.all import *
>>> RLF.gen()
1
```

interval_field(*prec=None*)

Return the interval field that represents the same mathematical field as self.

EXAMPLES:

```
sage: RLF.interval_field()
Real Interval Field with 53 bits of precision
sage: RLF.interval_field(200)
Real Interval Field with 200 bits of precision
```

```
>>> from sage.all import *
>>> RLF.interval_field()
Real Interval Field with 53 bits of precision
>>> RLF.interval_field(Integer(200))
Real Interval Field with 200 bits of precision
```

sage.rings.real_lazy.make_element(*parent, *args*)

Create an element of parent.

EXAMPLES:

```
sage: a = RLF(pi) + RLF(sqrt(1/2)) # indirect doctest #_
↪needs sage.symbolic
sage: bool(loads(dumps(a)) == a) #_
↪needs sage.symbolic
True
```

```
>>> from sage.all import *
>>> a = RLF(pi) + RLF(sqrt(Integer(1)/Integer(2))) # indirect doctest _
```

(continues on next page)

(continued from previous page)

```
↪          # needs sage.symbolic
>>> bool(loads(dumps(a)) == a)                                     #↪
↪needs sage.symbolic
True
```


INDICES AND TABLES

- [Index](#)
- [Module Index](#)
- [Search Page](#)

PYTHON MODULE INDEX

r

- `sage.rings.complex_arb`, 365
- `sage.rings.complex_double`, 163
- `sage.rings.complex_interval`, 292
- `sage.rings.complex_interval_field`, 283
- `sage.rings.complex_mpc`, 115
- `sage.rings.complex_mpfr`, 77
- `sage.rings.real_arb`, 314
- `sage.rings.real_double`, 138
- `sage.rings.real_interval_absolute`, 273
- `sage.rings.real_lazy`, 433
- `sage.rings.real_mpfi`, 197
- `sage.rings.real_mpfr`, 1

A

- `above_abs()` (*sage.rings.complex_arb.ComplexBall method*), 370
- `above_abs()` (*sage.rings.real_arb.RealBall method*), 321
- `abs()` (*sage.rings.complex_double.ComplexDoubleElement method*), 165
- `abs()` (*sage.rings.real_double.RealDoubleElement method*), 139
- `abs()` (*sage.rings.real_interval_absolute.RealIntervalAbsoluteElement method*), 274
- `abs2()` (*sage.rings.complex_double.ComplexDoubleElement method*), 165
- `absolute_diameter()` (*sage.rings.real_interval_absolute.RealIntervalAbsoluteElement method*), 275
- `absolute_diameter()` (*sage.rings.real_mpfir.RealIntervalFieldElement method*), 204
- `absprec()` (*sage.rings.real_interval_absolute.RealIntervalAbsoluteField_class method*), 281
- `accuracy()` (*sage.rings.complex_arb.ComplexBall method*), 371
- `accuracy()` (*sage.rings.real_arb.RealBall method*), 322
- `add_error()` (*sage.rings.complex_arb.ComplexBall method*), 371
- `add_error()` (*sage.rings.real_arb.RealBall method*), 322
- `additive_order()` (*sage.rings.complex_mpfir.ComplexNumber method*), 85
- `agm()` (*sage.rings.complex_double.ComplexDoubleElement method*), 165
- `agm()` (*sage.rings.complex_mpc.MPCComplexNumber method*), 120
- `agm()` (*sage.rings.complex_mpfir.ComplexNumber method*), 86
- `agm()` (*sage.rings.real_arb.RealBall method*), 323
- `agm()` (*sage.rings.real_double.RealDoubleElement method*), 139
- `agm()` (*sage.rings.real_mpfir.RealNumber method*), 13
- `agm1()` (*sage.rings.complex_arb.ComplexBall method*), 371
- `airy()` (*sage.rings.complex_arb.ComplexBall method*), 372
- `airy_ai()` (*sage.rings.complex_arb.ComplexBall method*), 372
- `airy_ai_prime()` (*sage.rings.complex_arb.ComplexBall method*), 372
- `airy_bi()` (*sage.rings.complex_arb.ComplexBall method*), 373
- `airy_bi_prime()` (*sage.rings.complex_arb.ComplexBall method*), 373
- `alea()` (*sage.rings.real_mpfir.RealIntervalFieldElement method*), 205
- `algdep()` (*sage.rings.complex_double.ComplexDoubleElement method*), 167
- `algdep()` (*sage.rings.complex_mpc.MPCComplexNumber method*), 120
- `algdep()` (*sage.rings.complex_mpfir.ComplexNumber method*), 87
- `algdep()` (*sage.rings.real_double.RealDoubleElement method*), 140
- `algdep()` (*sage.rings.real_mpfir.RealIntervalFieldElement method*), 205
- `algdep()` (*sage.rings.real_mpfir.RealNumber method*), 15
- `algebraic_closure()` (*sage.rings.complex_double.ComplexDoubleField_class method*), 189
- `algebraic_closure()` (*sage.rings.complex_mpfir.ComplexField_class method*), 80
- `algebraic_closure()` (*sage.rings.real_arb.RealBallField method*), 356
- `algebraic_closure()` (*sage.rings.real_double.RealDoubleField_class method*), 157
- `algebraic_closure()` (*sage.rings.real_lazy.LazyField method*), 438
- `algebraic_closure()` (*sage.rings.real_mpfir.RealIntervalField_class method*), 265
- `algebraic_closure()` (*sage.rings.real_mpfir.RealField_class method*), 4
- `algebraic_dependency()` (*sage.rings.complex_double.ComplexDoubleElement method*), 167
- `algebraic_dependency()` (*sage.rings.complex_mpc.MPCComplexNumber method*), 121
- `algebraic_dependency()` (*sage.rings.complex_mpfir.ComplexNumber method*), 88
- `algebraic_dependency()` (*sage.rings.real_double.RealDoubleElement method*), 140
- `algebraic_dependency()` (*sage.rings.real_mpfir.Real-*

- IntervalFieldElement* method), 206
- `algebraic_dependency()` (*sage.rings.real_mpfr.RealNumber* method), 15
- `approx()` (*sage.rings.real_lazy.LazyFieldElement* method), 439
- `approx()` (*sage.rings.real_lazy.LazyNamedUnop* method), 441
- `arccos()` (*sage.rings.complex_arb.ComplexBall* method), 373
- `arccos()` (*sage.rings.complex_double.ComplexDoubleElement* method), 168
- `arccos()` (*sage.rings.complex_mpc.MPComplexNumber* method), 122
- `arccos()` (*sage.rings.complex_mpfr.ComplexNumber* method), 88
- `arccos()` (*sage.rings.real_arb.RealBall* method), 323
- `arccos()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 207
- `arccos()` (*sage.rings.real_mpfr.RealNumber* method), 16
- `arccosh()` (*sage.rings.complex_arb.ComplexBall* method), 374
- `arccosh()` (*sage.rings.complex_double.ComplexDoubleElement* method), 169
- `arccosh()` (*sage.rings.complex_mpc.MPComplexNumber* method), 122
- `arccosh()` (*sage.rings.complex_mpfr.ComplexNumber* method), 89
- `arccosh()` (*sage.rings.real_arb.RealBall* method), 323
- `arccosh()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 208
- `arccosh()` (*sage.rings.real_mpfr.RealNumber* method), 16
- `arccot()` (*sage.rings.complex_double.ComplexDoubleElement* method), 169
- `arccoth()` (*sage.rings.complex_double.ComplexDoubleElement* method), 169
- `arccoth()` (*sage.rings.complex_mpc.MPComplexNumber* method), 122
- `arccoth()` (*sage.rings.complex_mpfr.ComplexNumber* method), 89
- `arccoth()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 208
- `arccoth()` (*sage.rings.real_mpfr.RealNumber* method), 17
- `arccsc()` (*sage.rings.complex_double.ComplexDoubleElement* method), 169
- `arccsch()` (*sage.rings.complex_double.ComplexDoubleElement* method), 169
- `arccsch()` (*sage.rings.complex_mpc.MPComplexNumber* method), 122
- `arccsch()` (*sage.rings.complex_mpfr.ComplexNumber* method), 89
- `arccsch()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 209
- `arccsch()` (*sage.rings.real_mpfr.RealNumber* method), 17
- `arcsec()` (*sage.rings.complex_double.ComplexDoubleElement* method), 170
- `arcsech()` (*sage.rings.complex_double.ComplexDoubleElement* method), 170
- `arcsech()` (*sage.rings.complex_mpc.MPComplexNumber* method), 123
- `arcsech()` (*sage.rings.complex_mpfr.ComplexNumber* method), 89
- `arcsech()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 209
- `arcsech()` (*sage.rings.real_mpfr.RealNumber* method), 17
- `arcsin()` (*sage.rings.complex_arb.ComplexBall* method), 374
- `arcsin()` (*sage.rings.complex_double.ComplexDoubleElement* method), 170
- `arcsin()` (*sage.rings.complex_mpc.MPComplexNumber* method), 123
- `arcsin()` (*sage.rings.complex_mpfr.ComplexNumber* method), 90
- `arcsin()` (*sage.rings.real_arb.RealBall* method), 324
- `arcsin()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 209
- `arcsin()` (*sage.rings.real_mpfr.RealNumber* method), 17
- `arcsinh()` (*sage.rings.complex_arb.ComplexBall* method), 375
- `arcsinh()` (*sage.rings.complex_double.ComplexDoubleElement* method), 170
- `arcsinh()` (*sage.rings.complex_mpc.MPComplexNumber* method), 123
- `arcsinh()` (*sage.rings.complex_mpfr.ComplexNumber* method), 90
- `arcsinh()` (*sage.rings.real_arb.RealBall* method), 324
- `arcsinh()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 210
- `arcsinh()` (*sage.rings.real_mpfr.RealNumber* method), 18
- `arctan()` (*sage.rings.complex_arb.ComplexBall* method), 375
- `arctan()` (*sage.rings.complex_double.ComplexDoubleElement* method), 171
- `arctan()` (*sage.rings.complex_mpc.MPComplexNumber* method), 123
- `arctan()` (*sage.rings.complex_mpfr.ComplexNumber* method), 90
- `arctan()` (*sage.rings.real_arb.RealBall* method), 324
- `arctan()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 210
- `arctan()` (*sage.rings.real_mpfr.RealNumber* method), 18
- `arctanh()` (*sage.rings.complex_arb.ComplexBall* method), 376
- `arctanh()` (*sage.rings.complex_double.ComplexDou-*

bleElement method), 171
`arctanh()` (*sage.rings.complex_mpc.MPCComplexNumber method*), 124
`arctanh()` (*sage.rings.complex_mpfr.ComplexNumber method*), 90
`arctanh()` (*sage.rings.real_arb.RealBall method*), 324
`arctanh()` (*sage.rings.real_mpfi.RealIntervalFieldElement method*), 211
`arctanh()` (*sage.rings.real_mpfr.RealNumber method*), 18
`arg()` (*sage.rings.complex_arb.ComplexBall method*), 376
`arg()` (*sage.rings.complex_double.ComplexDoubleElement method*), 171
`arg()` (*sage.rings.complex_interval.ComplexIntervalFieldElement method*), 293
`arg()` (*sage.rings.complex_mpfr.ComplexNumber method*), 91
`argument()` (*sage.rings.complex_double.ComplexDoubleElement method*), 172
`argument()` (*sage.rings.complex_interval.ComplexIntervalFieldElement method*), 293
`argument()` (*sage.rings.complex_mpc.MPCComplexNumber method*), 124
`argument()` (*sage.rings.complex_mpfr.ComplexNumber method*), 91
`argument()` (*sage.rings.real_mpfi.RealIntervalFieldElement method*), 211
`as_integer_ratio()` (*sage.rings.real_double.RealDoubleElement method*), 141
`as_integer_ratio()` (*sage.rings.real_mpfr.RealNumber method*), 19

B

`barnes_g()` (*sage.rings.complex_arb.ComplexBall method*), 376
`base` (*sage.rings.real_mpfr.RealLiteral attribute*), 12
`bell_number()` (*sage.rings.real_arb.RealBallField method*), 356
`below_abs()` (*sage.rings.complex_arb.ComplexBall method*), 377
`below_abs()` (*sage.rings.real_arb.RealBall method*), 325
`bernoulli()` (*sage.rings.real_arb.RealBallField method*), 357
`bessel_I()` (*sage.rings.complex_arb.ComplexBall method*), 378
`bessel_J()` (*sage.rings.complex_arb.ComplexBall method*), 378
`bessel_J_Y()` (*sage.rings.complex_arb.ComplexBall method*), 378
`bessel_K()` (*sage.rings.complex_arb.ComplexBall method*), 379
`bessel_Y()` (*sage.rings.complex_arb.ComplexBall method*), 379
`beta()` (*sage.rings.real_arb.RealBall method*), 326

`bisection()` (*sage.rings.complex_interval.ComplexIntervalFieldElement method*), 294
`bisection()` (*sage.rings.real_mpfi.RealIntervalFieldElement method*), 212

C

`catalan_constant()` (*sage.rings.real_arb.RealBallField method*), 357
`catalan_constant()` (*sage.rings.real_mpfr.RealField_class method*), 5
`CCToMPC` (*class in sage.rings.complex_mpc*), 116
`ceil()` (*sage.rings.real_arb.RealBall method*), 326
`ceil()` (*sage.rings.real_double.RealDoubleElement method*), 141
`ceil()` (*sage.rings.real_mpfi.RealIntervalFieldElement method*), 213
`ceil()` (*sage.rings.real_mpfr.RealNumber method*), 20
`ceiling()` (*sage.rings.real_double.RealDoubleElement method*), 142
`ceiling()` (*sage.rings.real_mpfi.RealIntervalFieldElement method*), 214
`ceiling()` (*sage.rings.real_mpfr.RealNumber method*), 20
`center()` (*sage.rings.complex_interval.ComplexIntervalFieldElement method*), 295
`center()` (*sage.rings.real_arb.RealBall method*), 327
`center()` (*sage.rings.real_mpfi.RealIntervalFieldElement method*), 215
`characteristic()` (*sage.rings.complex_arb.ComplexBallField method*), 422
`characteristic()` (*sage.rings.complex_double.ComplexDoubleField_class method*), 189
`characteristic()` (*sage.rings.complex_interval_field.ComplexIntervalField_class method*), 286
`characteristic()` (*sage.rings.complex_mpc.MPCComplexField_class method*), 117
`characteristic()` (*sage.rings.complex_mpfr.ComplexField_class method*), 81
`characteristic()` (*sage.rings.real_arb.RealBallField method*), 358
`characteristic()` (*sage.rings.real_double.RealDoubleField_class method*), 157
`characteristic()` (*sage.rings.real_mpfi.RealIntervalField_class method*), 265
`characteristic()` (*sage.rings.real_mpfr.RealField_class method*), 5
`chebyshev_T()` (*sage.rings.complex_arb.ComplexBall method*), 379
`chebyshev_T()` (*sage.rings.real_arb.RealBall method*), 327
`chebyshev_U()` (*sage.rings.complex_arb.ComplexBall method*), 380

- `chebyshev_U()` (*sage.rings.real_arb.RealBall* method), 328
- `Chi()` (*sage.rings.complex_arb.ComplexBall* method), 368
- `Chi()` (*sage.rings.real_arb.RealBall* method), 320
- `Ci()` (*sage.rings.complex_arb.ComplexBall* method), 368
- `Ci()` (*sage.rings.real_arb.RealBall* method), 320
- `cmp_abs()` (in module *sage.rings.complex_mpfr*), 111
- `complex_field()` (*sage.rings.complex_arb.ComplexBallField* method), 422
- `complex_field()` (*sage.rings.real_arb.RealBallField* method), 358
- `complex_field()` (*sage.rings.real_double.RealDoubleField_class* method), 157
- `complex_field()` (*sage.rings.real_mpfi.RealIntervalField_class* method), 265
- `complex_field()` (*sage.rings.real_mpfr.RealField_class* method), 5
- `ComplexBall` (class in *sage.rings.complex_arb*), 368
- `ComplexBallField` (class in *sage.rings.complex_arb*), 421
- `ComplexDoubleElement` (class in *sage.rings.complex_double*), 164
- `ComplexDoubleField()` (in module *sage.rings.complex_double*), 188
- `ComplexDoubleField_class` (class in *sage.rings.complex_double*), 189
- `ComplexField()` (in module *sage.rings.complex_mpfr*), 77
- `ComplexField_class` (class in *sage.rings.complex_mpfr*), 78
- `ComplexIntervalField()` (in module *sage.rings.complex_interval_field*), 283
- `ComplexIntervalField_class` (class in *sage.rings.complex_interval_field*), 284
- `ComplexIntervalFieldElement` (class in *sage.rings.complex_interval*), 292
- `ComplexLazyField()` (in module *sage.rings.real_lazy*), 433
- `ComplexLazyField_class` (class in *sage.rings.real_lazy*), 433
- `ComplexNumber` (class in *sage.rings.complex_mpfr*), 85
- `ComplexToCDF` (class in *sage.rings.complex_double*), 193
- `conj()` (*sage.rings.complex_double.ComplexDoubleElement* method), 172
- `conjugate()` (*sage.rings.complex_arb.ComplexBall* method), 380
- `conjugate()` (*sage.rings.complex_double.ComplexDoubleElement* method), 172
- `conjugate()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 295
- `conjugate()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 125
- `conjugate()` (*sage.rings.complex_mpfr.ComplexNumber* method), 92
- `conjugate()` (*sage.rings.real_double.RealDoubleElement* method), 142
- `conjugate()` (*sage.rings.real_mpfr.RealNumber* method), 21
- `construction()` (*sage.rings.complex_arb.ComplexBallField* method), 422
- `construction()` (*sage.rings.complex_double.ComplexDoubleField_class* method), 189
- `construction()` (*sage.rings.complex_interval_field.ComplexIntervalField_class* method), 287
- `construction()` (*sage.rings.complex_mpfr.ComplexField_class* method), 81
- `construction()` (*sage.rings.real_arb.RealBallField* method), 358
- `construction()` (*sage.rings.real_double.RealDoubleField_class* method), 157
- `construction()` (*sage.rings.real_lazy.ComplexLazyField_class* method), 434
- `construction()` (*sage.rings.real_lazy.RealLazyField_class* method), 445
- `construction()` (*sage.rings.real_mpfi.RealIntervalField_class* method), 266
- `construction()` (*sage.rings.real_mpfr.RealField_class* method), 6
- `contains_exact()` (*sage.rings.complex_arb.ComplexBall* method), 380
- `contains_exact()` (*sage.rings.real_arb.RealBall* method), 329
- `contains_integer()` (*sage.rings.complex_arb.ComplexBall* method), 381
- `contains_integer()` (*sage.rings.real_arb.RealBall* method), 330
- `contains_zero()` (*sage.rings.complex_arb.ComplexBall* method), 381
- `contains_zero()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 296
- `contains_zero()` (*sage.rings.real_arb.RealBall* method), 330
- `contains_zero()` (*sage.rings.real_interval_absolute.RealIntervalAbsoluteElement* method), 275
- `contains_zero()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 215
- `continued_fraction()` (*sage.rings.real_lazy.LazyFieldElement* method), 439
- `continued_fraction()` (*sage.rings.real_lazy.LazyWrapper* method), 443
- `cos()` (*sage.rings.complex_arb.ComplexBall* method), 382
- `cos()` (*sage.rings.complex_double.ComplexDoubleElement* method), 173
- `cos()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 296
- `cos()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 125

- `cos()` (*sage.rings.complex_mpfr.ComplexNumber* method), 92
`cos()` (*sage.rings.real_arb.RealBall* method), 330
`cos()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 216
`cos()` (*sage.rings.real_mpfr.RealNumber* method), 21
`cos_integral()` (*sage.rings.complex_arb.ComplexBall* method), 382
`cos_integral()` (*sage.rings.real_arb.RealBall* method), 331
`cosh()` (*sage.rings.complex_arb.ComplexBall* method), 382
`cosh()` (*sage.rings.complex_double.ComplexDoubleElement* method), 173
`cosh()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 297
`cosh()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 125
`cosh()` (*sage.rings.complex_mpfr.ComplexNumber* method), 92
`cosh()` (*sage.rings.real_arb.RealBall* method), 331
`cosh()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 216
`cosh()` (*sage.rings.real_mpfr.RealNumber* method), 22
`cosh_integral()` (*sage.rings.complex_arb.ComplexBall* method), 382
`cosh_integral()` (*sage.rings.real_arb.RealBall* method), 331
`cospi()` (*sage.rings.real_arb.RealBallField* method), 359
`cot()` (*sage.rings.complex_arb.ComplexBall* method), 383
`cot()` (*sage.rings.complex_double.ComplexDoubleElement* method), 173
`cot()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 126
`cot()` (*sage.rings.complex_mpfr.ComplexNumber* method), 92
`cot()` (*sage.rings.real_arb.RealBall* method), 331
`cot()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 216
`cot()` (*sage.rings.real_mpfr.RealNumber* method), 22
`coth()` (*sage.rings.complex_arb.ComplexBall* method), 383
`coth()` (*sage.rings.complex_double.ComplexDoubleElement* method), 173
`coth()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 126
`coth()` (*sage.rings.complex_mpfr.ComplexNumber* method), 93
`coth()` (*sage.rings.real_arb.RealBall* method), 332
`coth()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 216
`coth()` (*sage.rings.real_mpfr.RealNumber* method), 22
`create_ComplexIntervalFieldElement()` (in module *sage.rings.complex_interval*), 313
`create_ComplexNumber()` (in module *sage.rings.complex_mpfr*), 112
`create_key()` (*sage.rings.real_interval_absolute.Factory* method), 273
`create_object()` (*sage.rings.real_interval_absolute.Factory* method), 273
`create_RealBall()` (in module *sage.rings.real_arb*), 365
`create_RealNumber()` (in module *sage.rings.real_mpfr*), 69
`crosses_log_branch_cut()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 297
`csc()` (*sage.rings.complex_arb.ComplexBall* method), 383
`csc()` (*sage.rings.complex_double.ComplexDoubleElement* method), 174
`csc()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 127
`csc()` (*sage.rings.complex_mpfr.ComplexNumber* method), 93
`csc()` (*sage.rings.real_arb.RealBall* method), 332
`csc()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 217
`csc()` (*sage.rings.real_mpfr.RealNumber* method), 22
`csch()` (*sage.rings.complex_arb.ComplexBall* method), 384
`csch()` (*sage.rings.complex_double.ComplexDoubleElement* method), 174
`csch()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 127
`csch()` (*sage.rings.complex_mpfr.ComplexNumber* method), 93
`csch()` (*sage.rings.real_arb.RealBall* method), 332
`csch()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 217
`csch()` (*sage.rings.real_mpfr.RealNumber* method), 23
`cube()` (*sage.rings.complex_arb.ComplexBall* method), 384
`cube_root()` (*sage.rings.real_double.RealDoubleElement* method), 142
`cube_root()` (*sage.rings.real_mpfr.RealNumber* method), 23
- ## D
- `depth()` (*sage.rings.real_lazy.LazyBinop* method), 436
`depth()` (*sage.rings.real_lazy.LazyFieldElement* method), 440
`depth()` (*sage.rings.real_lazy.LazyUnop* method), 442
`depth()` (*sage.rings.real_lazy.LazyWrapper* method), 443
`diameter()` (*sage.rings.complex_arb.ComplexBall* method), 384
`diameter()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 298
`diameter()` (*sage.rings.real_arb.RealBall* method), 332

- diameter() (*sage.rings.real_interval_absolute.RealIntervalAbsoluteElement* method), 276
- diameter() (*sage.rings.real_mpf. RealIntervalFieldElement* method), 217
- dilog() (*sage.rings.complex_double.ComplexDoubleElement* method), 174
- dilog() (*sage.rings.complex_mpc.MPCComplexNumber* method), 127
- dilog() (*sage.rings.complex_mpf. ComplexNumber* method), 94
- double_factorial() (*sage.rings.real_arb.RealBallField* method), 359
- double_toRR (class in *sage.rings.real_mpf*), 71
- ## E
- edges() (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 298
- edges() (*sage.rings.real_mpf. RealIntervalFieldElement* method), 218
- Ei() (*sage.rings.complex_arb.ComplexBall* method), 369
- Ei() (*sage.rings.real_arb.RealBall* method), 320
- eint() (*sage.rings.real_mpf. RealNumber* method), 23
- eisenstein() (*sage.rings.complex_arb.ComplexBall* method), 385
- Element (*sage.rings.complex_arb.ComplexBallField* attribute), 422
- Element (*sage.rings.complex_interval_field.ComplexIntervalField_class* attribute), 286
- Element (*sage.rings.real_arb.RealBallField* attribute), 356
- Element (*sage.rings.real_lazy.LazyField* attribute), 438
- Element (*sage.rings.real_mpf. RealIntervalField_class* attribute), 265
- elliptic_e() (*sage.rings.complex_arb.ComplexBall* method), 385
- elliptic_e_inc() (*sage.rings.complex_arb.ComplexBall* method), 386
- elliptic_f() (*sage.rings.complex_arb.ComplexBall* method), 386
- elliptic_invariants() (*sage.rings.complex_arb.ComplexBall* method), 387
- elliptic_k() (*sage.rings.complex_arb.ComplexBall* method), 388
- elliptic_p() (*sage.rings.complex_arb.ComplexBall* method), 388
- elliptic_pi() (*sage.rings.complex_arb.ComplexBall* method), 389
- elliptic_pi_inc() (*sage.rings.complex_arb.ComplexBall* method), 389
- elliptic_rf() (*sage.rings.complex_arb.ComplexBall* method), 390
- elliptic_rg() (*sage.rings.complex_arb.ComplexBall* method), 390
- elliptic_rj() (*sage.rings.complex_arb.ComplexBall* method), 391
- elliptic_roots() (*sage.rings.complex_arb.ComplexBall* method), 391
- elliptic_sigma() (*sage.rings.complex_arb.ComplexBall* method), 392
- elliptic_zeta() (*sage.rings.complex_arb.ComplexBall* method), 392
- endpoints() (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 299
- endpoints() (*sage.rings.real_arb.RealBall* method), 333
- endpoints() (*sage.rings.real_interval_absolute.RealIntervalAbsoluteElement* method), 277
- endpoints() (*sage.rings.real_mpf. RealIntervalFieldElement* method), 219
- epsilon() (*sage.rings.real_mpf. RealNumber* method), 24
- erf() (*sage.rings.complex_arb.ComplexBall* method), 392
- erf() (*sage.rings.real_arb.RealBall* method), 333
- erf() (*sage.rings.real_mpf. RealNumber* method), 25
- erfc() (*sage.rings.complex_arb.ComplexBall* method), 392
- erfc() (*sage.rings.real_mpf. RealNumber* method), 26
- erfi() (*sage.rings.real_arb.RealBall* method), 334
- eta() (*sage.rings.complex_double.ComplexDoubleElement* method), 175
- eta() (*sage.rings.complex_mpc.MPCComplexNumber* method), 128
- eta() (*sage.rings.complex_mpf. ComplexNumber* method), 95
- euler_constant() (*sage.rings.real_arb.RealBallField* method), 360
- euler_constant() (*sage.rings.real_double.RealDoubleField_class* method), 158
- euler_constant() (*sage.rings.real_mpf. RealIntervalField_class* method), 266
- euler_constant() (*sage.rings.real_mpf. RealField_class* method), 6
- eval() (*sage.rings.real_lazy.LazyAlgebraic* method), 435
- eval() (*sage.rings.real_lazy.LazyBinop* method), 437
- eval() (*sage.rings.real_lazy.LazyConstant* method), 437
- eval() (*sage.rings.real_lazy.LazyFieldElement* method), 440
- eval() (*sage.rings.real_lazy.LazyNamedUnop* method), 441
- eval() (*sage.rings.real_lazy.LazyUnop* method), 442
- eval() (*sage.rings.real_lazy.LazyWrapper* method), 443
- exact_rational() (*sage.rings.real_mpf. RealNumber* method), 26
- exp() (*sage.rings.complex_arb.ComplexBall* method), 393
- exp() (*sage.rings.complex_double.ComplexDoubleElement* method), 177
- exp() (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 299
- exp() (*sage.rings.complex_mpc.MPCComplexNumber* method), 128

`exp()` (*sage.rings.complex_mpfr.ComplexNumber* method), 96
`exp()` (*sage.rings.real_arb.RealBall* method), 334
`exp()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 219
`exp()` (*sage.rings.real_mpfr.RealNumber* method), 27
`exp2()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 220
`exp2()` (*sage.rings.real_mpfr.RealNumber* method), 28
`exp10()` (*sage.rings.real_mpfr.RealNumber* method), 27
`exp_integral_e()` (*sage.rings.complex_arb.ComplexBall* method), 393
`expm1()` (*sage.rings.real_arb.RealBall* method), 334
`expm1()` (*sage.rings.real_mpfr.RealNumber* method), 29
`exppii()` (*sage.rings.complex_arb.ComplexBall* method), 393

F

`factorial()` (*sage.rings.real_double.RealDoubleField_class* method), 158
`factorial()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 221
`factorial()` (*sage.rings.real_mpfr.RealField_class* method), 6
`Factory` (class in *sage.rings.real_interval_absolute*), 273
`fibonacci()` (*sage.rings.real_arb.RealBallField* method), 360
`FloatToCDF` (class in *sage.rings.complex_double*), 194
`floor()` (*sage.rings.real_arb.RealBall* method), 334
`floor()` (*sage.rings.real_double.RealDoubleElement* method), 143
`floor()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 221
`floor()` (*sage.rings.real_mpfr.RealNumber* method), 29
`fp_rank()` (*sage.rings.real_mpfr.RealNumber* method), 30
`fp_rank_delta()` (*sage.rings.real_mpfr.RealNumber* method), 31
`fp_rank_diameter()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 222
`frac()` (*sage.rings.real_double.RealDoubleElement* method), 143
`frac()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 224
`frac()` (*sage.rings.real_mpfr.RealNumber* method), 32

G

`gamma()` (*sage.rings.complex_arb.ComplexBall* method), 394
`gamma()` (*sage.rings.complex_double.ComplexDoubleElement* method), 178
`gamma()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 129

`gamma()` (*sage.rings.complex_mpfr.ComplexNumber* method), 97
`gamma()` (*sage.rings.real_arb.RealBall* method), 335
`gamma()` (*sage.rings.real_arb.RealBallField* method), 361
`gamma()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 225
`gamma()` (*sage.rings.real_mpfr.RealNumber* method), 32
`gamma_inc()` (*sage.rings.complex_arb.ComplexBall* method), 394
`gamma_inc()` (*sage.rings.complex_double.ComplexDoubleElement* method), 178
`gamma_inc()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 129
`gamma_inc()` (*sage.rings.complex_mpfr.ComplexNumber* method), 97
`gamma_inc()` (*sage.rings.real_arb.RealBall* method), 335
`gamma_inc_lower()` (*sage.rings.real_arb.RealBall* method), 336
`gegenbauer_C()` (*sage.rings.complex_arb.ComplexBall* method), 395
`gen()` (*sage.rings.complex_arb.ComplexBallField* method), 423
`gen()` (*sage.rings.complex_double.ComplexDoubleField_class* method), 190
`gen()` (*sage.rings.complex_interval_field.ComplexIntervalField_class* method), 287
`gen()` (*sage.rings.complex_mpc.MPCComplexField_class* method), 118
`gen()` (*sage.rings.complex_mpfr.ComplexField_class* method), 81
`gen()` (*sage.rings.real_double.RealDoubleField_class* method), 158
`gen()` (*sage.rings.real_lazy.ComplexLazyField_class* method), 434
`gen()` (*sage.rings.real_lazy.RealLazyField_class* method), 446
`gen()` (*sage.rings.real_mpfi.RealIntervalField_class* method), 266
`gen()` (*sage.rings.real_mpfr.RealField_class* method), 7
`gens()` (*sage.rings.complex_arb.ComplexBallField* method), 423
`gens()` (*sage.rings.real_arb.RealBallField* method), 362
`gens()` (*sage.rings.real_mpfi.RealIntervalField_class* method), 267
`gens()` (*sage.rings.real_mpfr.RealField_class* method), 7

H

`hermite_H()` (*sage.rings.complex_arb.ComplexBall* method), 395
`hex()` (*sage.rings.real_mpfr.RealNumber* method), 33
`hypergeometric()` (*sage.rings.complex_arb.ComplexBall* method), 395
`hypergeometric_U()` (*sage.rings.complex_arb.ComplexBall* method), 398

- I**
- `identical()` (*sage.rings.complex_arb.ComplexBall method*), 398
 - `identical()` (*sage.rings.real_arb.RealBall method*), 336
 - `imag()` (*sage.rings.complex_arb.ComplexBall method*), 398
 - `imag()` (*sage.rings.complex_double.ComplexDoubleElement method*), 179
 - `imag()` (*sage.rings.complex_interval.ComplexIntervalFieldElement method*), 300
 - `imag()` (*sage.rings.complex_mpc.MPCComplexNumber method*), 129
 - `imag()` (*sage.rings.complex_mpfr.ComplexNumber method*), 98
 - `imag()` (*sage.rings.real_arb.RealBall method*), 336
 - `imag()` (*sage.rings.real_double.RealDoubleElement method*), 143
 - `imag()` (*sage.rings.real_mpfi.RealIntervalFieldElement method*), 227
 - `imag()` (*sage.rings.real_mpfr.RealNumber method*), 33
 - `imag_part()` (*sage.rings.complex_double.ComplexDoubleElement method*), 179
 - `imag_part()` (*sage.rings.complex_mpfr.ComplexNumber method*), 98
 - `int_toRR` (class in *sage.rings.real_mpfr*), 71
 - `integer_part()` (*sage.rings.real_double.RealDoubleElement method*), 144
 - `integer_part()` (*sage.rings.real_mpfr.RealNumber method*), 34
 - `INTEGERToMPC` (class in *sage.rings.complex_mpc*), 116
 - `integral()` (*sage.rings.complex_arb.ComplexBallField method*), 423
 - `IntegrationContext` (class in *sage.rings.complex_arb*), 431
 - `intersection()` (*sage.rings.complex_interval.ComplexIntervalFieldElement method*), 300
 - `intersection()` (*sage.rings.real_mpfi.RealIntervalFieldElement method*), 227
 - `interval_field()` (*sage.rings.real_lazy.ComplexLazyField_class method*), 434
 - `interval_field()` (*sage.rings.real_lazy.LazyField method*), 439
 - `interval_field()` (*sage.rings.real_lazy.RealLazyField_class method*), 446
 - `is_ComplexDoubleElement()` (in module *sage.rings.complex_double*), 195
 - `is_ComplexIntervalFieldElement()` (in module *sage.rings.complex_interval*), 314
 - `is_ComplexNumber()` (in module *sage.rings.complex_mpfr*), 113
 - `is_exact()` (*sage.rings.complex_arb.ComplexBall method*), 399
 - `is_exact()` (*sage.rings.complex_arb.ComplexBallField method*), 428
 - `is_exact()` (*sage.rings.complex_double.ComplexDoubleField_class method*), 190
 - `is_exact()` (*sage.rings.complex_interval_field.ComplexIntervalField_class method*), 287
 - `is_exact()` (*sage.rings.complex_interval.ComplexIntervalFieldElement method*), 301
 - `is_exact()` (*sage.rings.complex_mpc.MPCComplexField_class method*), 118
 - `is_exact()` (*sage.rings.complex_mpfr.ComplexField_class method*), 81
 - `is_exact()` (*sage.rings.real_arb.RealBall method*), 337
 - `is_exact()` (*sage.rings.real_arb.RealBallField method*), 362
 - `is_exact()` (*sage.rings.real_double.RealDoubleField_class method*), 158
 - `is_exact()` (*sage.rings.real_mpfi.RealIntervalField_class method*), 267
 - `is_exact()` (*sage.rings.real_mpfi.RealIntervalFieldElement method*), 228
 - `is_exact()` (*sage.rings.real_mpfr.RealField_class method*), 7
 - `is_field()` (*sage.rings.complex_interval_field.ComplexIntervalField_class method*), 287
 - `is_finite()` (*sage.rings.real_arb.RealBall method*), 338
 - `is_imaginary()` (*sage.rings.complex_mpc.MPCComplexNumber method*), 130
 - `is_imaginary()` (*sage.rings.complex_mpfr.ComplexNumber method*), 99
 - `is_infinity()` (*sage.rings.complex_double.ComplexDoubleElement method*), 180
 - `is_infinity()` (*sage.rings.complex_mpfr.ComplexNumber method*), 99
 - `is_infinity()` (*sage.rings.real_arb.RealBall method*), 338
 - `is_infinity()` (*sage.rings.real_double.RealDoubleElement method*), 144
 - `is_infinity()` (*sage.rings.real_mpfr.RealNumber method*), 35
 - `is_int()` (*sage.rings.real_mpfi.RealIntervalFieldElement method*), 229
 - `is_integer()` (*sage.rings.complex_double.ComplexDoubleElement method*), 180
 - `is_integer()` (*sage.rings.complex_mpfr.ComplexNumber method*), 99
 - `is_integer()` (*sage.rings.real_double.RealDoubleElement method*), 145
 - `is_integer()` (*sage.rings.real_mpfr.RealNumber method*), 35
 - `is_NaN()` (*sage.rings.complex_arb.ComplexBall method*), 399
 - `is_NaN()` (*sage.rings.complex_double.ComplexDoubleElement method*), 180
 - `is_NaN()` (*sage.rings.complex_interval.ComplexIntervalFieldElement method*), 300

- `is_NaN()` (*sage.rings.complex_mpfr.ComplexNumber* method), 98
`is_NaN()` (*sage.rings.real_arb.RealBall* method), 337
`is_NaN()` (*sage.rings.real_double.RealDoubleElement* method), 144
`is_NaN()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 228
`is_NaN()` (*sage.rings.real_mpfr.RealNumber* method), 34
`is_negative()` (*sage.rings.real_interval_absolute.RealIntervalAbsoluteElement* method), 277
`is_negative_infinity()` (*sage.rings.complex_double.ComplexDoubleElement* method), 181
`is_negative_infinity()` (*sage.rings.complex_mpfr.ComplexNumber* method), 100
`is_negative_infinity()` (*sage.rings.real_arb.RealBall* method), 339
`is_negative_infinity()` (*sage.rings.real_double.RealDoubleElement* method), 145
`is_negative_infinity()` (*sage.rings.real_mpfr.RealNumber* method), 36
`is_nonzero()` (*sage.rings.complex_arb.ComplexBall* method), 400
`is_nonzero()` (*sage.rings.real_arb.RealBall* method), 339
`is_positive()` (*sage.rings.real_interval_absolute.RealIntervalAbsoluteElement* method), 278
`is_positive_infinity()` (*sage.rings.complex_double.ComplexDoubleElement* method), 181
`is_positive_infinity()` (*sage.rings.complex_mpfr.ComplexNumber* method), 100
`is_positive_infinity()` (*sage.rings.real_arb.RealBall* method), 339
`is_positive_infinity()` (*sage.rings.real_double.RealDoubleElement* method), 145
`is_positive_infinity()` (*sage.rings.real_mpfr.RealNumber* method), 36
`is_real()` (*sage.rings.complex_arb.ComplexBall* method), 400
`is_real()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 130
`is_real()` (*sage.rings.complex_mpfr.ComplexNumber* method), 100
`is_real()` (*sage.rings.real_mpfr.RealNumber* method), 37
`is_RealDoubleElement()` (in module *sage.rings.real_double*), 163
`is_RealIntervalField()` (in module *sage.rings.real_mpfi*), 272
`is_RealIntervalFieldElement()` (in module *sage.rings.real_mpfi*), 272
`is_RealNumber()` (in module *sage.rings.real_mpfr*), 72
`is_square()` (*sage.rings.complex_double.ComplexDoubleElement* method), 181
`is_square()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 302
`is_square()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 131
`is_square()` (*sage.rings.complex_mpfr.ComplexNumber* method), 101
`is_square()` (*sage.rings.real_double.RealDoubleElement* method), 146
`is_square()` (*sage.rings.real_mpfr.RealNumber* method), 37
`is_unit()` (*sage.rings.real_mpfr.RealNumber* method), 38
`is_zero()` (*sage.rings.complex_arb.ComplexBall* method), 401
`is_zero()` (*sage.rings.real_arb.RealBall* method), 340
- ## J
- `j0()` (*sage.rings.real_mpfr.RealNumber* method), 38
`j1()` (*sage.rings.real_mpfr.RealNumber* method), 38
`jacobi_P()` (*sage.rings.complex_arb.ComplexBall* method), 401
`jacobi_theta()` (*sage.rings.complex_arb.ComplexBall* method), 401
`jn()` (*sage.rings.real_mpfr.RealNumber* method), 39
- ## L
- `laguerre_L()` (*sage.rings.complex_arb.ComplexBall* method), 402
`lambert_w()` (*sage.rings.complex_arb.ComplexBall* method), 403
`lambert_w()` (*sage.rings.real_arb.RealBall* method), 340
`late_import()` (in module *sage.rings.complex_mpc*), 137
`late_import()` (in module *sage.rings.complex_mpfr*), 114
`LazyAlgebraic` (class in *sage.rings.real_lazy*), 435
`LazyBinop` (class in *sage.rings.real_lazy*), 436
`LazyConstant` (class in *sage.rings.real_lazy*), 437
`LazyField` (class in *sage.rings.real_lazy*), 438
`LazyFieldElement` (class in *sage.rings.real_lazy*), 439
`LazyNamedUnop` (class in *sage.rings.real_lazy*), 440
`LazyUnop` (class in *sage.rings.real_lazy*), 441
`LazyWrapper` (class in *sage.rings.real_lazy*), 442
`LazyWrapperMorphism` (class in *sage.rings.real_lazy*), 444
`legendre_P()` (*sage.rings.complex_arb.ComplexBall* method), 403
`legendre_Q()` (*sage.rings.complex_arb.ComplexBall* method), 404
`lexico_cmp()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 302
`lexico_cmp()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 229
`Li()` (*sage.rings.complex_arb.ComplexBall* method), 369
`li()` (*sage.rings.complex_arb.ComplexBall* method), 404

`Li()` (*sage.rings.real_arb.RealBall* method), 320
`li()` (*sage.rings.real_arb.RealBall* method), 340
`literal` (*sage.rings.real_mpfr.RealLiteral* attribute), 12
`log()` (*sage.rings.complex_arb.ComplexBall* method), 405
`log()` (*sage.rings.complex_double.ComplexDoubleElement* method), 182
`log()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 303
`log()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 131
`log()` (*sage.rings.complex_mpfr.ComplexNumber* method), 101
`log()` (*sage.rings.real_arb.RealBall* method), 341
`log()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 230
`log()` (*sage.rings.real_mpfr.RealNumber* method), 39
`log1p()` (*sage.rings.complex_arb.ComplexBall* method), 406
`log1p()` (*sage.rings.real_arb.RealBall* method), 341
`log1p()` (*sage.rings.real_mpfr.RealNumber* method), 41
`log2()` (*sage.rings.real_arb.RealBallField* method), 362
`log2()` (*sage.rings.real_double.RealDoubleField_class* method), 159
`log2()` (*sage.rings.real_mpfi.RealIntervalField_class* method), 267
`log2()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 231
`log2()` (*sage.rings.real_mpfr.RealField_class* method), 8
`log2()` (*sage.rings.real_mpfr.RealNumber* method), 42
`log10()` (*sage.rings.complex_double.ComplexDoubleElement* method), 182
`log10()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 231
`log10()` (*sage.rings.real_mpfr.RealNumber* method), 40
`log_b()` (*sage.rings.complex_double.ComplexDoubleElement* method), 182
`log_barnes_g()` (*sage.rings.complex_arb.ComplexBall* method), 406
`log_gamma()` (*sage.rings.complex_arb.ComplexBall* method), 407
`log_gamma()` (*sage.rings.real_arb.RealBall* method), 342
`log_gamma()` (*sage.rings.real_mpfr.RealNumber* method), 43
`log_integral()` (*sage.rings.complex_arb.ComplexBall* method), 407
`log_integral()` (*sage.rings.real_arb.RealBall* method), 342
`log_integral_offset()` (*sage.rings.complex_arb.ComplexBall* method), 408
`log_integral_offset()` (*sage.rings.real_arb.RealBall* method), 342
`logabs()` (*sage.rings.complex_double.ComplexDoubleElement* method), 183
`lower()` (*sage.rings.real_arb.RealBall* method), 342

`lower()` (*sage.rings.real_interval_absolute.RealIntervalAbsoluteElement* method), 278
`lower()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 232
`lower_field()` (*sage.rings.real_mpfi.RealIntervalField_class* method), 267

M

`magnitude()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 304
`magnitude()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 233
`make_ComplexIntervalFieldElement0()` (in module *sage.rings.complex_interval*), 314
`make_ComplexNumber0()` (in module *sage.rings.complex_mpfr*), 114
`make_element()` (in module *sage.rings.real_lazy*), 446
`max()` (*sage.rings.real_arb.RealBall* method), 343
`max()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 234
`maximal_accuracy()` (*sage.rings.real_arb.RealBallField* method), 362
`mid()` (*sage.rings.complex_arb.ComplexBall* method), 408
`mid()` (*sage.rings.real_arb.RealBall* method), 343
`middle_field()` (*sage.rings.complex_interval_field.ComplexIntervalField_class* method), 288
`middle_field()` (*sage.rings.real_mpfi.RealIntervalField_class* method), 268
`midpoint()` (*sage.rings.real_interval_absolute.RealIntervalAbsoluteElement* method), 279
`mignitude()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 305
`mignitude()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 236
`min()` (*sage.rings.real_arb.RealBall* method), 344
`min()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 236
`modular_delta()` (*sage.rings.complex_arb.ComplexBall* method), 409
`modular_eta()` (*sage.rings.complex_arb.ComplexBall* method), 410
`modular_j()` (*sage.rings.complex_arb.ComplexBall* method), 410
`modular_lambda()` (*sage.rings.complex_arb.ComplexBall* method), 410
`module`
 sage.rings.complex_arb, 365
 sage.rings.complex_double, 163
 sage.rings.complex_interval, 292
 sage.rings.complex_interval_field, 283
 sage.rings.complex_mpc, 115
 sage.rings.complex_mpfr, 77
 sage.rings.real_arb, 314

- sage.rings.real_double, 138
 sage.rings.real_interval_absolute, 273
 sage.rings.real_lazy, 433
 sage.rings.real_mphi, 197
 sage.rings.real_mphi, 1
 MPComplexField() (in module sage.rings.complex_mpc), 116
 MPComplexField_class (class in sage.rings.complex_mpc), 117
 MPComplexNumber (class in sage.rings.complex_mpc), 120
 MPctoMPC (class in sage.rings.complex_mpc), 137
 mphi_prec() (sage.rings.real_interval_absolute.RealIntervalAbsoluteElement method), 279
 mphi_get_exp_max() (in module sage.rings.real_mphi), 72
 mphi_get_exp_max_max() (in module sage.rings.real_mphi), 73
 mphi_get_exp_min() (in module sage.rings.real_mphi), 73
 mphi_get_exp_min_min() (in module sage.rings.real_mphi), 74
 mphi_prec_max() (in module sage.rings.real_mphi), 74
 mphi_prec_min() (in module sage.rings.real_mphi), 75
 mphi_set_exp_max() (in module sage.rings.real_mphi), 75
 mphi_set_exp_min() (in module sage.rings.real_mphi), 76
 MphiOp (class in sage.rings.real_interval_absolute), 273
 MPFRtoMPC (class in sage.rings.complex_mpc), 137
 multiplicative_order() (sage.rings.complex_interval.ComplexIntervalFieldElement method), 305
 multiplicative_order() (sage.rings.complex_mphi.ComplexNumber method), 103
 multiplicative_order() (sage.rings.real_double.RealDoubleElement method), 146
 multiplicative_order() (sage.rings.real_mphi.RealIntervalFieldElement method), 238
 multiplicative_order() (sage.rings.real_mphi.RealNumber method), 43
- ## N
- name() (sage.rings.complex_mpc.MPComplexField_class method), 118
 name() (sage.rings.real_double.RealDoubleField_class method), 159
 name() (sage.rings.real_mphi.RealIntervalField_class method), 268
 name() (sage.rings.real_mphi.RealField_class method), 8
 NaN() (sage.rings.real_double.RealDoubleElement method), 139
 nan() (sage.rings.real_double.RealDoubleElement method), 147
 NaN() (sage.rings.real_double.RealDoubleField_class method), 156
 nan() (sage.rings.real_double.RealDoubleField_class method), 159
 nbits() (sage.rings.complex_arb.ComplexBall method), 411
 nbits() (sage.rings.real_arb.RealBall method), 345
 nearby_rational() (sage.rings.real_mphi.RealNumber method), 44
 nextabove() (sage.rings.real_mphi.RealNumber method), 45
 nextbelow() (sage.rings.real_mphi.RealNumber method), 46
 nexttoward() (sage.rings.real_mphi.RealNumber method), 47
 ngens() (sage.rings.complex_arb.ComplexBallField method), 428
 ngens() (sage.rings.complex_double.ComplexDoubleField_class method), 190
 ngens() (sage.rings.complex_interval_field.ComplexIntervalField_class method), 288
 ngens() (sage.rings.complex_mpc.MPComplexField_class method), 118
 ngens() (sage.rings.complex_mphi.ComplexField_class method), 82
 ngens() (sage.rings.real_double.RealDoubleField_class method), 159
 ngens() (sage.rings.real_mphi.RealIntervalField_class method), 268
 ngens() (sage.rings.real_mphi.RealField_class method), 8
 norm() (sage.rings.complex_double.ComplexDoubleElement method), 183
 norm() (sage.rings.complex_interval.ComplexIntervalFieldElement method), 306
 norm() (sage.rings.complex_mpc.MPComplexNumber method), 131
 norm() (sage.rings.complex_mphi.ComplexNumber method), 104
 nth_root() (sage.rings.complex_double.ComplexDoubleElement method), 184
 nth_root() (sage.rings.complex_mpc.MPComplexNumber method), 132
 nth_root() (sage.rings.complex_mphi.ComplexNumber method), 104
 nth_root() (sage.rings.real_mphi.RealNumber method), 47
 numerical_approx() (sage.rings.real_mphi.RealLiteral method), 12
- ## O
- overlaps() (sage.rings.complex_arb.ComplexBall method), 411
 overlaps() (sage.rings.complex_interval.ComplexIntervalFieldElement method), 306

`overlaps()` (*sage.rings.real_arb.RealBall* method), 345
`overlaps()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 239

P

`pi()` (*sage.rings.complex_arb.ComplexBallField* method), 428
`pi()` (*sage.rings.complex_double.ComplexDoubleField_class* method), 190
`pi()` (*sage.rings.complex_interval_field.ComplexIntervalField_class* method), 288
`pi()` (*sage.rings.complex_mpfr.ComplexField_class* method), 82
`pi()` (*sage.rings.real_arb.RealBallField* method), 363
`pi()` (*sage.rings.real_double.RealDoubleField_class* method), 160
`pi()` (*sage.rings.real_mpfi.RealIntervalField_class* method), 268
`pi()` (*sage.rings.real_mpfr.RealField_class* method), 8
`plot()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 307
`plot()` (*sage.rings.complex_mpfr.ComplexNumber* method), 105
`polylog()` (*sage.rings.complex_arb.ComplexBall* method), 412
`polylog()` (*sage.rings.real_arb.RealBall* method), 346
`pow()` (*sage.rings.complex_arb.ComplexBall* method), 412
`prec()` (*sage.rings.complex_arb.ComplexBallField* method), 429
`prec()` (*sage.rings.complex_double.ComplexDoubleElement* method), 184
`prec()` (*sage.rings.complex_double.ComplexDoubleField_class* method), 191
`prec()` (*sage.rings.complex_interval_field.ComplexIntervalField_class* method), 288
`prec()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 308
`prec()` (*sage.rings.complex_mpc.MPCComplexField_class* method), 118
`prec()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 132
`prec()` (*sage.rings.complex_mpfr.ComplexField_class* method), 82
`prec()` (*sage.rings.complex_mpfr.ComplexNumber* method), 106
`prec()` (*sage.rings.real_arb.RealBallField* method), 363
`prec()` (*sage.rings.real_double.RealDoubleElement* method), 147
`prec()` (*sage.rings.real_double.RealDoubleField_class* method), 160
`prec()` (*sage.rings.real_mpfi.RealIntervalField_class* method), 269
`prec()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 239

`prec()` (*sage.rings.real_mpfr.RealField_class* method), 9
`prec()` (*sage.rings.real_mpfr.RealNumber* method), 49
`precision()` (*sage.rings.complex_arb.ComplexBallField* method), 429
`precision()` (*sage.rings.complex_double.ComplexDoubleField_class* method), 191
`precision()` (*sage.rings.complex_interval_field.ComplexIntervalField_class* method), 289
`precision()` (*sage.rings.complex_mpfr.ComplexField_class* method), 82
`precision()` (*sage.rings.real_arb.RealBallField* method), 363
`precision()` (*sage.rings.real_double.RealDoubleField_class* method), 160
`precision()` (*sage.rings.real_mpfi.RealIntervalField_class* method), 269
`precision()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 240
`precision()` (*sage.rings.real_mpfr.RealField_class* method), 9
`precision()` (*sage.rings.real_mpfr.RealNumber* method), 50
`psi()` (*sage.rings.complex_arb.ComplexBall* method), 413
`psi()` (*sage.rings.real_arb.RealBall* method), 347
`psi()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 240

Q

`QQtoRR` (class in *sage.rings.real_mpfr*), 2

R

`rad()` (*sage.rings.complex_arb.ComplexBall* method), 413
`rad()` (*sage.rings.real_arb.RealBall* method), 347
`rad_as_ball()` (*sage.rings.real_arb.RealBall* method), 347
`random_element()` (*sage.rings.complex_double.ComplexDoubleField_class* method), 191
`random_element()` (*sage.rings.complex_interval_field.ComplexIntervalField_class* method), 289
`random_element()` (*sage.rings.complex_mpc.MPCComplexField_class* method), 119
`random_element()` (*sage.rings.complex_mpfr.ComplexField_class* method), 83
`random_element()` (*sage.rings.real_double.RealDoubleField_class* method), 160
`random_element()` (*sage.rings.real_mpfi.RealIntervalField_class* method), 269
`random_element()` (*sage.rings.real_mpfr.RealField_class* method), 9
`real()` (*sage.rings.complex_arb.ComplexBall* method), 414
`real()` (*sage.rings.complex_double.ComplexDoubleElement* method), 185

- `real()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 308
`real()` (*sage.rings.complex_mpc.MPComplexNumber* method), 133
`real()` (*sage.rings.complex_mpfr.ComplexNumber* method), 106
`real()` (*sage.rings.real_arb.RealBall* method), 348
`real()` (*sage.rings.real_double.RealDoubleElement* method), 147
`real()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 240
`real()` (*sage.rings.real_mpfr.RealNumber* method), 50
`real_double_field()` (*sage.rings.complex_double.ComplexDoubleField_class* method), 192
`real_field()` (*sage.rings.complex_interval_field.ComplexIntervalField_class* method), 290
`real_part()` (*sage.rings.complex_double.ComplexDoubleElement* method), 185
`real_part()` (*sage.rings.complex_mpfr.ComplexNumber* method), 106
`RealBall` (class in *sage.rings.real_arb*), 319
`RealBallField` (class in *sage.rings.real_arb*), 355
`RealDoubleElement` (class in *sage.rings.real_double*), 139
`RealDoubleField()` (in module *sage.rings.real_double*), 155
`RealDoubleField_class` (class in *sage.rings.real_double*), 155
`RealField()` (in module *sage.rings.real_mpfr*), 3
`RealField_class` (class in *sage.rings.real_mpfr*), 4
`RealInterval()` (in module *sage.rings.real_mpfi*), 203
`RealIntervalAbsoluteElement` (class in *sage.rings.real_interval_absolute*), 273
`RealIntervalAbsoluteField()` (in module *sage.rings.real_interval_absolute*), 280
`RealIntervalAbsoluteField_class` (class in *sage.rings.real_interval_absolute*), 281
`RealIntervalField()` (in module *sage.rings.real_mpfi*), 204
`RealIntervalField_class` (class in *sage.rings.real_mpfi*), 260
`RealIntervalFieldElement` (class in *sage.rings.real_mpfi*), 204
`RealLazyField()` (in module *sage.rings.real_lazy*), 444
`RealLazyField_class` (class in *sage.rings.real_lazy*), 444
`RealLiteral` (class in *sage.rings.real_mpfr*), 12
`RealNumber` (class in *sage.rings.real_mpfr*), 13
`relative_diameter()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 241
`rgamma()` (*sage.rings.complex_arb.ComplexBall* method), 414
`rgamma()` (*sage.rings.real_arb.RealBall* method), 348
`rising_factorial()` (*sage.rings.complex_arb.ComplexBall* method), 414
`rising_factorial()` (*sage.rings.real_arb.RealBall* method), 348
`round()` (*sage.rings.complex_arb.ComplexBall* method), 415
`round()` (*sage.rings.real_arb.RealBall* method), 349
`round()` (*sage.rings.real_double.RealDoubleElement* method), 147
`round()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 241
`round()` (*sage.rings.real_mpfr.RealNumber* method), 50
`rounding_mode()` (*sage.rings.complex_mpc.MPComplexField_class* method), 119
`rounding_mode()` (*sage.rings.real_mpfr.RealField_class* method), 10
`rounding_mode_imag()` (*sage.rings.complex_mpc.MPComplexField_class* method), 119
`rounding_mode_real()` (*sage.rings.complex_mpc.MPComplexField_class* method), 120
`RRtoCC` (class in *sage.rings.complex_mpfr*), 111
`RRtoRR` (class in *sage.rings.real_mpfr*), 2
`rsqrt()` (*sage.rings.complex_arb.ComplexBall* method), 416
`rsqrt()` (*sage.rings.real_arb.RealBall* method), 349
- ## S
- `sage.rings.complex_arb` module, 365
`sage.rings.complex_double` module, 163
`sage.rings.complex_interval` module, 292
`sage.rings.complex_interval_field` module, 283
`sage.rings.complex_mpc` module, 115
`sage.rings.complex_mpfr` module, 77
`sage.rings.real_arb` module, 314
`sage.rings.real_double` module, 138
`sage.rings.real_interval_absolute` module, 273
`sage.rings.real_lazy` module, 433
`sage.rings.real_mpfi` module, 197
`sage.rings.real_mpfr` module, 1
`scientific_notation()` (*sage.rings.complex_interval_field.ComplexIntervalField_class* method), 290

`scientific_notation()` (*sage.rings.complex_mpfr.ComplexField_class* method), 84
`scientific_notation()` (*sage.rings.real_mpfi.RealIntervalField_class* method), 270
`scientific_notation()` (*sage.rings.real_mpfr.RealField_class* method), 10
`sec()` (*sage.rings.complex_arb.ComplexBall* method), 416
`sec()` (*sage.rings.complex_double.ComplexDoubleElement* method), 185
`sec()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 133
`sec()` (*sage.rings.complex_mpfr.ComplexNumber* method), 107
`sec()` (*sage.rings.real_arb.RealBall* method), 350
`sec()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 242
`sec()` (*sage.rings.real_mpfr.RealNumber* method), 51
`sech()` (*sage.rings.complex_arb.ComplexBall* method), 417
`sech()` (*sage.rings.complex_double.ComplexDoubleElement* method), 185
`sech()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 133
`sech()` (*sage.rings.complex_mpfr.ComplexNumber* method), 107
`sech()` (*sage.rings.real_arb.RealBall* method), 350
`sech()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 242
`sech()` (*sage.rings.real_mpfr.RealNumber* method), 51
`section()` (*sage.rings.complex_mpc.MPCtoMPC* method), 137
`section()` (*sage.rings.real_mpfr.RRtoRR* method), 2
`set_global_complex_round_mode()` (in module *sage.rings.complex_mpfr*), 115
`Shi()` (*sage.rings.complex_arb.ComplexBall* method), 369
`Shi()` (*sage.rings.real_arb.RealBall* method), 321
`shift_ceil()` (in module *sage.rings.real_interval_absolute*), 282
`shift_floor()` (in module *sage.rings.real_interval_absolute*), 282
`Si()` (*sage.rings.complex_arb.ComplexBall* method), 370
`Si()` (*sage.rings.real_arb.RealBall* method), 321
`sign()` (*sage.rings.real_double.RealDoubleElement* method), 148
`sign()` (*sage.rings.real_mpfr.RealNumber* method), 51
`sign_mantissa_exponent()` (*sage.rings.real_double.RealDoubleElement* method), 148
`sign_mantissa_exponent()` (*sage.rings.real_mpfr.RealNumber* method), 52
`simplest_rational()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 242
`simplest_rational()` (*sage.rings.real_mpfr.RealNumber* method), 53
`sin()` (*sage.rings.complex_arb.ComplexBall* method), 417
`sin()` (*sage.rings.complex_double.ComplexDoubleElement* method), 186
`sin()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 308
`sin()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 134
`sin()` (*sage.rings.complex_mpfr.ComplexNumber* method), 107
`sin()` (*sage.rings.real_arb.RealBall* method), 350
`sin()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 244
`sin()` (*sage.rings.real_mpfr.RealNumber* method), 57
`sin_integral()` (*sage.rings.complex_arb.ComplexBall* method), 417
`sin_integral()` (*sage.rings.real_arb.RealBall* method), 350
`sincos()` (*sage.rings.real_mpfr.RealNumber* method), 57
`sinh()` (*sage.rings.complex_arb.ComplexBall* method), 417
`sinh()` (*sage.rings.complex_double.ComplexDoubleElement* method), 186
`sinh()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 309
`sinh()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 134
`sinh()` (*sage.rings.complex_mpfr.ComplexNumber* method), 108
`sinh()` (*sage.rings.real_arb.RealBall* method), 351
`sinh()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 244
`sinh()` (*sage.rings.real_mpfr.RealNumber* method), 58
`sinh_integral()` (*sage.rings.complex_arb.ComplexBall* method), 418
`sinh_integral()` (*sage.rings.real_arb.RealBall* method), 351
`sinpi()` (*sage.rings.real_arb.RealBallField* method), 363
`some_elements()` (*sage.rings.complex_arb.ComplexBallField* method), 429
`some_elements()` (*sage.rings.real_arb.RealBallField* method), 364
`spherical_harmonic()` (*sage.rings.complex_arb.ComplexBall* method), 418
`split_complex_string()` (in module *sage.rings.complex_mpc*), 137
`sqr()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 134
`sqrt()` (*sage.rings.complex_arb.ComplexBall* method), 418
`sqrt()` (*sage.rings.complex_double.ComplexDoubleElement* method), 186
`sqrt()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 310
`sqrt()` (*sage.rings.complex_mpc.MPCComplexNumber* method), 135

- `sqrt()` (*sage.rings.complex_mpfr.ComplexNumber* method), 108
`sqrt()` (*sage.rings.real_arb.RealBall* method), 351
`sqrt()` (*sage.rings.real_double.RealDoubleElement* method), 149
`sqrt()` (*sage.rings.real_interval_absolute.RealIntervalAbsoluteElement* method), 280
`sqrt()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 244
`sqrt()` (*sage.rings.real_mpfr.RealNumber* method), 58
`sqrt1pm1()` (*sage.rings.real_arb.RealBall* method), 351
`sqrtpos()` (*sage.rings.real_arb.RealBall* method), 352
`square()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 246
`square_root()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 246
`squash()` (*sage.rings.complex_arb.ComplexBall* method), 419
`squash()` (*sage.rings.real_arb.RealBall* method), 352
`str()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 310
`str()` (*sage.rings.complex_mpc.MPComplexNumber* method), 135
`str()` (*sage.rings.complex_mpfr.ComplexNumber* method), 108
`str()` (*sage.rings.real_double.RealDoubleElement* method), 150
`str()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 247
`str()` (*sage.rings.real_mpfr.RealNumber* method), 59
- ## T
- `tan()` (*sage.rings.complex_arb.ComplexBall* method), 419
`tan()` (*sage.rings.complex_double.ComplexDoubleElement* method), 187
`tan()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 311
`tan()` (*sage.rings.complex_mpc.MPComplexNumber* method), 136
`tan()` (*sage.rings.complex_mpfr.ComplexNumber* method), 110
`tan()` (*sage.rings.real_arb.RealBall* method), 353
`tan()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 251
`tan()` (*sage.rings.real_mpfr.RealNumber* method), 63
`tanh()` (*sage.rings.complex_arb.ComplexBall* method), 419
`tanh()` (*sage.rings.complex_double.ComplexDoubleElement* method), 188
`tanh()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 312
`tanh()` (*sage.rings.complex_mpc.MPComplexNumber* method), 136
`tanh()` (*sage.rings.complex_mpfr.ComplexNumber* method), 110
`tanh()` (*sage.rings.real_arb.RealBall* method), 353
`tanh()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 251
`to_prec()` (*sage.rings.complex_double.ComplexDoubleField_class* method), 192
`to_prec()` (*sage.rings.complex_interval_field.ComplexIntervalField_class* method), 291
`to_prec()` (*sage.rings.complex_mpfr.ComplexField_class* method), 84
`to_prec()` (*sage.rings.real_double.RealDoubleField_class* method), 161
`to_prec()` (*sage.rings.real_mpfi.RealIntervalField_class* method), 271
`to_prec()` (*sage.rings.real_mpfr.RealField_class* method), 11
`ToRDF` (class in *sage.rings.real_double*), 162
`trim()` (*sage.rings.complex_arb.ComplexBall* method), 420
`trim()` (*sage.rings.real_arb.RealBall* method), 353
`trunc()` (*sage.rings.real_double.RealDoubleElement* method), 151
`trunc()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 252
`trunc()` (*sage.rings.real_mpfr.RealNumber* method), 64
- ## U
- `ulp()` (*sage.rings.real_double.RealDoubleElement* method), 152
`ulp()` (*sage.rings.real_mpfr.RealNumber* method), 64
`union()` (*sage.rings.complex_arb.ComplexBall* method), 420
`union()` (*sage.rings.complex_interval.ComplexIntervalFieldElement* method), 312
`union()` (*sage.rings.real_arb.RealBall* method), 354
`union()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 253
`unique_ceil()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 253
`unique_floor()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 254
`unique_integer()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 255
`unique_round()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 255
`unique_sign()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 256
`unique_trunc()` (*sage.rings.real_mpfi.RealIntervalFieldElement* method), 257
`upper()` (*sage.rings.real_arb.RealBall* method), 354
`upper()` (*sage.rings.real_interval_absolute.RealIntervalAbsoluteElement* method), 280

`upper()` (*sage.rings.real_mpfi.RealIntervalFieldElement*
method), 258

`upper_field()` (*sage.rings.real_mpfi.RealInterval-*
Field_class method), 271

Y

`y0()` (*sage.rings.real_mpfr.RealNumber* method), 67

`y1()` (*sage.rings.real_mpfr.RealNumber* method), 67

`yn()` (*sage.rings.real_mpfr.RealNumber* method), 68

Z

`zeta()` (*sage.rings.complex_arb.ComplexBall* method),
421

`zeta()` (*sage.rings.complex_double.ComplexDoubleEle-*
ment method), 188

`zeta()` (*sage.rings.complex_double.ComplexDouble-*
Field_class method), 192

`zeta()` (*sage.rings.complex_interval_field.ComplexInter-*
valField_class method), 291

`zeta()` (*sage.rings.complex_interval.ComplexInterval-*
FieldElement method), 312

`zeta()` (*sage.rings.complex_mpc.MPCComplexNumber*
method), 136

`zeta()` (*sage.rings.complex_mpfr.ComplexField_class*
method), 85

`zeta()` (*sage.rings.complex_mpfr.ComplexNumber*
method), 110

`zeta()` (*sage.rings.real_arb.RealBall* method), 354

`zeta()` (*sage.rings.real_arb.RealBallField* method), 364

`zeta()` (*sage.rings.real_double.RealDoubleField_class*
method), 161

`zeta()` (*sage.rings.real_mpfi.RealIntervalField_class*
method), 271

`zeta()` (*sage.rings.real_mpfi.RealIntervalFieldElement*
method), 259

`zeta()` (*sage.rings.real_mpfr.RealField_class* method), 12

`zeta()` (*sage.rings.real_mpfr.RealNumber* method), 68

`zeta_zeros()` (*sage.rings.complex_arb.ComplexBall-*
Field method), 430

`zetaderiv()` (*sage.rings.complex_arb.ComplexBall*
method), 421

`zetaderiv()` (*sage.rings.real_arb.RealBall* method), 355

`ZZtoRR` (class in *sage.rings.real_mpfr*), 69