Sage Reference Manual: Fixed and Arbitrary Precision Numerical Fields

Release 7.0

The Sage Development Team

January 20, 2016

CHAPTER

ONE

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 Real Numbers

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 (trac ticket #13033)
- Travis Scrimshaw (2012-11-02): Added doctests for full coverage

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
```

TESTS:

```
sage: -1e30
-1.00000000000000e30
sage: hex(-1. + 2^-52)
'-0xf.ffffffffffffp-4'
```

Make sure we don't have a new field for every new literal:

```
sage: parent(2.0) is parent(2.0)
True
sage: RealField(100, rnd='RNDZ') is RealField(100, rnd='RNDD')
False
sage: RealField(100, rnd='RNDZ') is RealField(100, rnd='RNDZ')
True
sage: RealField(100, rnd='RNDZ') is RealField(100, rnd=1)
True
```

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

Bases: `sage.categories.map.Map`

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

Bases: `sage.categories.map.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
```

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

RealField(prec, sci_not, rnd):

INPUT:

- `prec` – (integer) precision; default = 53 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` – (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
  - 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
```

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
```

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.'

---

class `sage.rings.real_mpfr.`**`RealField_class`**

Bases: `sage.rings.ring.Field`

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 the documentation for the module `sage.rings.real_mpfr` for more details.

`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
```

catalan_constant()

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

EXAMPLES:

```
sage: RealField(100).catalan_constant()
0.91596559417721901505460351493
```

characteristic()

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

EXAMPLES:

```
sage: RealField(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
```

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
```

euler_constant()

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

EXAMPLES:

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

factorial (*n*)

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

EXAMPLES:

```
sage: RR.factorial(0)
1.00000000000000
sage: RR.factorial(1000000)
8.26393168833124e5565708
sage: RR.factorial(-1)
Traceback (most recent call last):
...
ArithmeticError: n must be nonnegative
```

gen (*i=0*)

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

EXAMPLES:

```
sage: R=RealField(100)
sage: R.gen(0)
1.0000000000000000000000000000
sage: R.gen(1)
Traceback (most recent call last):
...
IndexError: self has only one generator
```

gens ()

Return a list of generators.

EXAMPLE:

```
sage: RR.gens()
[1.00000000000000]
```

is_exact ()

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

EXAMPLE:

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

is_finite ()

Return `False`, since the field of real numbers is not finite.

EXAMPLES:

```
sage: RealField(10).is_finite()
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
```

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'
```

ngens()

Return the number of generators.

EXAMPLES:

```
sage: 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
```

prec()

Return the precision of `self`.

EXAMPLES:

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

precision()

Return the precision of `self`.

EXAMPLES:

```
sage: RR.precision()
53
sage: RealField(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: RealField(100).random_element(-5, 10)
-1.7093633198207765227646362966
sage: RealField(10).random_element()
-0.11
```

TESTS:

```
sage: RealField(31).random_element()
-0.676162510
sage: RealField(32).random_element()
0.689774422
sage: RealField(33).random_element()
0.396496861
sage: RealField(63).random_element()
-0.339980711116375371
sage: RealField(64).random_element()
-0.0453049884016705260
sage: RealField(65).random_element()
-0.592671470958970813 7
sage: RealField(10).random_element()
0.23
sage: RealField(10).random_element()
-0.41
sage: RR.random_element()
-0.0420335212948924
sage: RR.random_element()
-0.616678906367394
```

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'
```

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
0.251200000000000
sage: RR.scientific_notation(True)
sage: elt
2.51200000000000e-1
sage: RR.scientific_notation()
True
```

```
sage: RR.scientific_notation(False)
sage: elt
0.251200000000000
sage: R = RealField(20, sci_not=1)
sage: R.scientific_notation()
True
sage: R(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
```

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()
-1.00000000000000
sage: R.zeta(1)
1.00000000000000
sage: R.zeta(5)
Traceback (most recent call last):
...
ValueError: No 5th root of unity in self
```

class `sage.rings.real_mpfr.`**RealLiteral**

Bases: `sage.rings.real_mpfr.RealNumber`

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

base

literal

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

Bases: `sage.structure.element.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
```

The AGM always lies between the geometric and arithmetic mean:

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

It is, of course, symmetric:

```
sage: 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
```

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]
sage: RR.pi()/4 * (a+b)/E
1.96811775182478
```

TESTS:

```
sage: 1.5.agm(0)
0.000000000000000
```

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 `algdep` command.

EXAMPLE:

```
sage: r = sqrt(2.0); r
1.41421356237310
sage: r.algebraic_dependency(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 `algdep` command.

EXAMPLE:

```
sage: r = sqrt(2.0); r
1.41421356237310
sage: r.algebraic_dependency(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
```

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
```

arccoth()

Return the inverse hyperbolic cotangent of `self`.

EXAMPLES:

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

arcsech()

Return the inverse hyperbolic secant of `self`.

EXAMPLES:

```
sage: i = RR.pi()/3
sage: q = i.sech()
sage: 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
0.000000000000000
```

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
0.000000000000000
```

arctan()

Return the inverse tangent of `self`.

EXAMPLES:

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

ceil()

Return the ceiling of `self`.

EXAMPLES:

```
sage: (2.99).ceil()
3
```

```
sage: (2.00).ceil()
2
sage: (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
```

ceiling()

Return the ceiling of `self`.

EXAMPLES:

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

conjugate()

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

EXAMPLES:

```
sage: x = RealField(100)(1.238)
sage: x.conjugate()
1.2380000000000000000000000000
```

cos()

Returnn the cosine of `self`.

EXAMPLES:

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

cosh()

Return the hyperbolic cosine of `self`.

EXAMPLES:

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

cot()

Return the cotangent of `self`.

EXAMPLES:

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

coth()

Return the hyperbolic cotangent of `self`.

EXAMPLES:

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

csc()

Return the cosecant of `self`.

EXAMPLES:

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

csch()

Return the hyperbolic cosecant of `self`.

EXAMPLES:

```
sage: RealField(100)(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()
5.00000000000000
sage: r = -119.0
sage: r.cube_root()^3 - r        # illustrates precision loss
-1.42108547152020e-14
```

eint()

Returns the exponential integral of this number.

EXAMPLES:

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

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

epsilon(*field=None*)

Returns `abs(self)` divided by 2^b where b is the precision in bits of `self`. Equivalently, return `abs(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 `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()
1.00000000000000
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
```

We use a different field:

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

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
```

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()
1.00000000000000
```

erfc()

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

EXAMPLES:

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

exact_rational()

Returns 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()
-25/8
```

TESTS:

```
sage: RR('nan').exact_rational()
Traceback (most recent call last):
...
ValueError: Cannot convert NaN or infinity to rational number
sage: RR('-infinity').exact_rational()
Traceback (most recent call last):
...
ValueError: Cannot convert NaN or infinity to rational number
```

exp()

Return $e^{\texttt{self}}$.

EXAMPLES:

```
sage: r = 0.0
sage: r.exp()
1.00000000000000

sage: r = 32.3
sage: a = r.exp(); a
1.06588847274864e14
sage: a.log()
32.3000000000000

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

exp10()

Return $10^{\texttt{self}}$.

EXAMPLES:

```
sage: r = 0.0
sage: r.exp10()
1.00000000000000

sage: r = 32.0
sage: r.exp10()
1.00000000000000e32

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

exp2()

Return $2^{\texttt{self}}$.

EXAMPLES:

```
sage: r = 0.0
sage: r.exp2()
1.00000000000000

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

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

expm1()

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

EXAMPLES:

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

sage: r = 1e-16
sage: exp(r)-1
0.000000000000000
sage: r.expm1()
1.00000000000000e-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
```

fp_rank()

Returns 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
```

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
...       (RR(-infinity), -1.0, 0.0, 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
```

There are lots of floating-point numbers around 0:

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

frac()

Return a real number such that `self = self.trunc() + self.frac()`. The return value will also satisfy `-1 < self.frac() < 1`.

EXAMPLES:

```
sage: (2.99).frac()
0.990000000000000
sage: (2.50).frac()
0.500000000000000
sage: (-2.79).frac()
-0.790000000000000
sage: (-2.79).trunc() + (-2.79).frac()
-2.79000000000000
```

gamma()

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

EXAMPLES:

```
sage: R = RealField()
sage: R(6).gamma()
120.000000000000
sage: R(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'

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')]
```

Special values:

```
sage: [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
```

integer_part()

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

OUTPUT: a Sage Integer

EXAMPLE:

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

A big number with no decimal point:

```
sage: a = RR(10^17); a
1.00000000000000e17
sage: 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
```

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
```

is_integer()

Return `True` if this number is a integer.

EXAMPLES:

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

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
```

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
```

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 non-negative.

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
```

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
```

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
```

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
```

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
```

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

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

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

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

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

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

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

log1p()

Return log base e of `1 + self`.

EXAMPLES:

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

For small values, this is more accurate than computing `log(1 + self)` directly, as it avoids cancellation issues:

```
sage: r = 3e-10
sage: r.log1p()
2.99999999955000e-10
sage: (1+r).log()
3.00000024777111e-10
sage: r100 = RealField(100)(r)
sage: (1+r100).log()
2.9999999995500000000978021372e-10
```

For small values, this is more accurate than computing `log(1 + self)` directly, as it avoid cancelation issues:

```
sage: r = 3e-10
sage: r.log1p()
2.99999999955000e-10
sage: (1+r).log()
3.00000024777111e-10
sage: r100 = RealField(100)(r)
sage: (1+r100).log()
2.9999999995500000000978021372e-10

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

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

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

log2()

Return log to the base 2 of `self`.

EXAMPLES:

```
sage: r = 16.0
sage: r.log2()
4.00000000000000

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

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

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

log_gamma()

Return the logarithm of gamma of `self`.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(6).log_gamma()
4.78749174278205
sage: R(1e10).log_gamma()
2.20258509288811e11
```

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
```

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 `[self-max_error .. self+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
```

```
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: RR(pi).nearby_rational(max_denominator=120)
355/113
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
```

TESTS:

```
sage: RR('nan').nearby_rational(max_denominator=1000)
Traceback (most recent call last):
...
ValueError: Cannot convert NaN or infinity to rational number
sage: RR('nan').nearby_rational(max_error=0.01)
Traceback (most recent call last):
...
ValueError: Cannot convert NaN or infinity to rational number
sage: RR(oo).nearby_rational(max_denominator=1000)
Traceback (most recent call last):
...
ValueError: Cannot convert NaN or infinity to rational number
sage: RR(oo).nearby_rational(max_error=0.01)
Traceback (most recent call last):
...
ValueError: Cannot convert NaN or infinity to rational number
```

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(truncate=False)
'-1.4142135623730951'
sage: RR(-sqrt(2)).nextabove().str(truncate=False)
'-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(truncate=False)
'-1.4142135623730951'
sage: RR(-sqrt(2)).nextbelow().str(truncate=False)
'-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(truncate=False)
'1.0000000000000002'
sage: (1.0).nexttoward(RR('-infinity')).str(truncate=False)
'0.99999999999999989'
sage: RR(infinity).nexttoward(0)
2.09857871646739e323228496             # 32-bit
5.87565378911159e1388255822130839282   # 64-bit
sage: RR(pi).str(truncate=False)
'3.1415926535897931'
sage: RR(pi).nexttoward(22/7).str(truncate=False)
'3.1415926535897936'
sage: RR(pi).nexttoward(21/7).str(truncate=False)
'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)
2.00000000000000
sage: R(8).nth_root(3.7)     # illustrate rounding down
2.00000000000000
sage: R(-8).nth_root(3)
-2.00000000000000
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)
32.0000000000000
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(4000000)
1.00000034657365
sage: R(-27).nth_root(3)
-3.00000000000000
sage: R(-4).nth_root(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
```

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
```

TESTS:

The old and new algorithms should give exactly the same results in all cases:

```
sage: def check(x, n):
...       answers = []
...       for sign in (1, -1):
...           if is_even(n) and sign == -1:
...               continue
...           for rounding in ('RNDN', 'RNDD', 'RNDU', 'RNDZ'):
...               fld = RealField(x.prec(), rnd=rounding)
...               fx = fld(sign * x)
...               alg_mpfr = fx.nth_root(n, algorithm=1)
...               alg_mpfi = fx.nth_root(n, algorithm=2)
...               assert(alg_mpfr == alg_mpfi)
...               if sign == 1: answers.append(alg_mpfr)
...       return answers
```

Check some perfect powers (and nearby numbers):

```
sage: check(16.0, 4)
[2.00000000000000, 2.00000000000000, 2.00000000000000, 2.00000000000000]
sage: check((16.0).nextabove(), 4)
[2.00000000000000, 2.00000000000000, 2.00000000000001, 2.00000000000000]
sage: check((16.0).nextbelow(), 4)
```

```
[2.00000000000000, 1.99999999999999, 2.00000000000000, 1.99999999999999]
sage: check(((9.0 * 256)^7), 7)
[2304.00000000000, 2304.00000000000, 2304.00000000000, 2304.00000000000]
sage: check(((9.0 * 256)^7).nextabove(), 7)
[2304.00000000000, 2304.00000000000, 2304.00000000001, 2304.00000000000]
sage: check(((9.0 * 256)^7).nextbelow(), 7)
[2304.00000000000, 2303.99999999999, 2304.00000000000, 2303.99999999999]
sage: check(((5.0 / 512)^17), 17)
[0.00976562500000000, 0.00976562500000000, 0.00976562500000000, 0.00976562500000000]
sage: check(((5.0 / 512)^17).nextabove(), 17)
[0.00976562500000000, 0.00976562500000000, 0.00976562500000001, 0.00976562500000000]
sage: check(((5.0 / 512)^17).nextbelow(), 17)
[0.00976562500000000, 0.00976562499999999, 0.00976562500000000, 0.00976562499999999]
```

And check some non-perfect powers:

```
sage: check(2.0, 3)
[1.25992104989487, 1.25992104989487, 1.25992104989488, 1.25992104989487]
sage: check(2.0, 4)
[1.18920711500272, 1.18920711500272, 1.18920711500273, 1.18920711500272]
sage: check(2.0, 5)
[1.14869835499704, 1.14869835499703, 1.14869835499704, 1.14869835499703]
```

And some different precisions:

```
sage: check(RealField(20)(22/7), 19)
[1.0621, 1.0621, 1.0622, 1.0621]
sage: check(RealField(200)(e), 4)
[1.2840254166877414840734205680624364583362808652814630892175, 1.2840254166877414840734205680
```

Check that trac ticket #12105 is fixed:

```
sage: RealField(53)(0.05).nth_root(7 * 10^8)
0.999999995720382
```

prec()

Return the precision of `self`.

EXAMPLES:

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

precision()

Return the precision of `self`.

EXAMPLES:

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

real()

Return the real part of `self`.

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

EXAMPLES:

```
sage: RR(2).real()
2.00000000000000
sage: RealField(200)(-4.5).real()
-4.5000000000000000000000000000000000000000000000000000000000
```

round()

Rounds `self` to the nearest integer. The rounding mode of the parent field has no effect on this function.

EXAMPLES:

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

sec()

Returns the secant of this number

EXAMPLES:

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

sech()

Return the hyperbolic secant of `self`.

EXAMPLES:

```
sage: RealField(100)(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(2.1).sign()
1
sage: R(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`, `+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)` (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 `+infinity`, `-infinity`, 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
```

The mantissa is always a nonnegative number (see trac ticket #14448):

```
sage: RR(-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)
```

TESTS:

```
sage: R('+0').sign_mantissa_exponent()
(1, 0, 0)
sage: R('-0').sign_mantissa_exponent()
(-1, 0, 0)
```

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))
-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))
355/113
sage: check(RR(pi))
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')
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
```

TESTS:

```
sage: RR('nan').simplest_rational()
Traceback (most recent call last):
...
ValueError: Cannot convert NaN or infinity to rational number
sage: RR('-infinity').simplest_rational()
Traceback (most recent call last):
...
ValueError: Cannot convert NaN or infinity to rational number
```

sin()

Return the sine of `self`.

EXAMPLES:

```
sage: R = RealField(100)
sage: R(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()
(0.500000000000000, 0.866025403784439)
```

sinh()

Return the hyperbolic sine of `self`.

EXAMPLES:

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

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

The square root function.

INPUT:

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

EXAMPLES:

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

sage: r = 4.0
sage: r.sqrt()
2.00000000000000
sage: r.sqrt()^2 == r
True

sage: r = 4344
sage: r.sqrt()
2*sqrt(1086)

sage: r = 4344.0
sage: r.sqrt()^2 == r
True
sage: r.sqrt()^2 - r
0.000000000000000

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

str(*base=10*, *no_sci=None*, *e=None*, *truncate=1*, *skip_zeroes=0*)

Return a string representation of `self`.

INPUT:

- `base` – base for output
- `no_sci` – if 2, never print using scientific notation; if 1 or `True`, print using scientific notation only for very large or very small numbers; if 0 or `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` – if `True`, round off the last digits in printing to lessen confusing base-2 roundoff issues.
- `skip_zeroes` – if `True`, skip trailing zeroes in mantissa

EXAMPLES:

```
sage: a = 61/3.0; a
20.3333333333333
sage: a.str(truncate=False)
'20.333333333333332'
sage: a.str(2)
'10100.010101010101010101010101010101010101010101010101'
sage: a.str(no_sci=False)
'2.03333333333333e1'
sage: a.str(16, no_sci=False)
'1.4555555555555@1'
sage: b = 2.0^99
sage: b.str()
'6.33825300114115e29'
sage: b.str(no_sci=False)
'6.33825300114115e29'
sage: b.str(no_sci=True)
'6.33825300114115e29'
sage: c = 2.0^100
sage: c.str()
'1.26765060022823e30'
sage: c.str(no_sci=False)
'1.26765060022823e30'
sage: c.str(no_sci=True)
'1.26765060022823e30'
sage: c.str(no_sci=2)
'1267650600228230000000000000000.'
sage: 0.5^53
1.11022302462516e-16
sage: 0.5^54
5.55111512312578e-17
sage: (0.01).str()
'0.0100000000000000'
sage: (0.01).str(skip_zeroes=True)
'0.01'
sage: (-10.042).str()
'-10.0420000000000'
sage: (-10.042).str(skip_zeroes=True)
'-10.042'
sage: (389.0).str(skip_zeroes=True)
'389.'
```

Test various bases:

```
sage: print (65536.0).str(base=2)
1.000000000000000000000000000000000000000000000000000e16
sage: print (65536.0).str(base=36)
1ekg.00000000
sage: print (65536.0).str(base=62)
H32.0000000
sage: print (65536.0).str(base=63)
Traceback (most recent call last):
...
ValueError: base (=63) must be an integer between 2 and 62
```

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
```

tanh()

Return the hyperbolic tangent of `self`.

EXAMPLES:

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

trunc()

Truncate `self`.

EXAMPLES:

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

ulp(*field=None*)

Returns 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
```

The ulp of zero is the smallest non-zero number:

```
sage: a = RR(0).ulp()
sage: a
2.38256490488795e-323228497            # 32-bit
8.50969131174084e-1388255822130839284  # 64-bit
sage: a.fp_rank()
1
```

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

```
sage: 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
```

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
```

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
```

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
```

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
```

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
```

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
1.64493406684823
sage: pari(2).zeta(precision=128).python().prec()
128
sage: pari(2).zeta(precision=65).python().prec()
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)
<type 'sage.libs.pari.gen.gen'>
sage: R(z)
1.64493406684823
```

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

Bases: `sage.categories.map.Map`

`sage.rings.real_mpfr.`**create_RealField**(*prec=53*, *type='MPFR'*, *rnd='RNDN'*, *sci_not=0*)

Create a real field with given precision, type, rounding mode and scientific notation.

Some options are ignored for certain types (RDF for example).

INPUT:

- `prec` – a positive integer

- `type` – type of real field:
  - `'RDF'` – the Sage real field corresponding to native doubles
  - `'Interval'` – real fields implementing interval arithmetic
  - `'RLF'` – the real lazy field
  - `'MPFR'` – floating point real numbers implemented using the MPFR library
- `rnd` – rounding mode:
  - `'RNDN'` – round to nearest
  - `'RNDZ'` – round toward zero
  - `'RNDD'` – round down
  - `'RNDU'` – round up
- `sci_not` – boolean, whether to use scientific notation for printing

OUTPUT:

the appropriate real field

EXAMPLES:

```
sage: from sage.rings.real_mpfr import create_RealField
sage: create_RealField(30)
Real Field with 30 bits of precision
sage: create_RealField(20, 'RDF') # ignores precision
Real Double Field
sage: create_RealField(60, 'Interval')
Real Interval Field with 60 bits of precision
sage: create_RealField(40, 'RLF') # ignores precision
Real Lazy Field
```

`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 slightly more (controlled by pad) bits than given by `s`.

INPUT:

- `s` – a string that defines a real number (or something whose string representation defines a number)
- `base` – an integer between 2 and 62
- `pad` – an integer = 0.
- `rnd` – rounding mode:
  - `'RNDN'` – round to nearest
  - `'RNDZ'` – round toward zero
  - `'RNDD'` – round down
  - `'RNDU'` – round up
- `min_prec` – number will have at least this many bits of precision, no matter what.

EXAMPLES:

```
sage: RealNumber('2.3') # indirect doctest
2.30000000000000
sage: RealNumber(10)
10.0000000000000
```

```
sage: RealNumber('1.0000000000000000000000000000000000')
1.000000000000000000000000000000000
sage: RealField(200)(1.2)
1.200000000000000000000000000000000000000000000000000000000
sage: (1.2).parent() is RR
True
```

We can use various bases:

```
sage: RealNumber("10101e2",base=2)
84.0000000000000
sage: RealNumber("deadbeef", base=16)
3.73592855900000e9
sage: RealNumber("deadbeefxxx", base=16)
Traceback (most recent call last):
...
TypeError: Unable to convert x (='deadbeefxxx') to real number.
sage: RealNumber("z", base=36)
35.0000000000000
sage: RealNumber("AAA", base=37)
14070.0000000000
sage: RealNumber("aaa", base=37)
50652.0000000000
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
```

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("1.50000000000000000000000000000000000000", rnd="RNDU").parent()
Real Field with 130 bits of precision and rounding RNDU
```

TESTS:

```
sage: RealNumber('.00000000000000000000000000000000001').prec()
53
sage: RealNumber('-.00000000000000000000000000000000001').prec()
53
```

Make sure we've rounded up `log(10,2)` enough to guarantee sufficient precision (trac ticket #10164):

```
sage: ks = 5*10**5, 10**6
sage: all(RealNumber("1." + "0"*k +"1")-1 > 0 for k in ks)
True
```

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

Bases: `sage.categories.map.Map`

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

Bases: `sage.categories.map.Map`

`sage.rings.real_mpfr.`**`is_RealField`**(*x*)

Returns `True` if `x` is technically of a Python real field type.

EXAMPLES:

```
sage: sage.rings.real_mpfr.is_RealField(RR)
True
sage: sage.rings.real_mpfr.is_RealField(CC)
False
```

`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)
True
sage: is_RealNumber(float(2.3))
False
sage: is_RealNumber(RDF(2))
False
sage: is_RealNumber(pi)
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
```

`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
```

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()
```

`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)
0.000000000000000
```

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
```

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()
```

sage.rings.real_mpfr.**mpfr_prec_max**()

TESTS:

```
sage: from sage.rings.real_mpfr import mpfr_prec_max
sage: mpfr_prec_max()
2147483391
sage: R = RealField(2^31-257)
sage: R
Real Field with 2147483391 bits of precision
sage: R = RealField(2^31-256)
Traceback (most recent call last):
...
ValueError: prec (=2147483392) must be >= 2 and <= 2147483391
```

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()
2
sage: R = RealField(2)
sage: R(2) + R(1)
3.0
sage: R(4) + R(1)
4.0
sage: R = RealField(1)
Traceback (most recent call last):
...
ValueError: prec (=1) must be >= 2 and <= 2147483391
```

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
```

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
0.000000000000000
sage: mpfr_set_exp_min(old)
sage: 0.5 >> 1001
2.33315904625805e-302
```

1.2 Field of Arbitrary Precision Complex Numbers

AUTHORS:

- William Stein (2006-01-26): complete rewrite
- Niles Johnson (2010-08): trac ticket #3893: `random_element()` should pass on `*args` and `**kwds`.
- Travis Scrimshaw (2012-10-18): Added documentation for full coverage.

sage.rings.complex_field.**ComplexField**(*prec=53*, *names=None*)

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

EXAMPLES:

```
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
-1.0000000000000000000000000000000000000000000000000000000000
```

class sage.rings.complex_field.**ComplexField_class**(*prec=53*)

Bases: `sage.rings.ring.Field`

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
```

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: unable to coerce to a ComplexNumber: <type 'sage.rings.polynomial.polynomial_rational
```

This illustrates precision:

```
sage: CC = ComplexField(10); CC(1/3, 2/3)
0.33 + 0.67*I
sage: CC
Complex Field with 10 bits of precision
sage: CC = ComplexField(100); CC
Complex Field with 100 bits of precision
sage: z = CC(1/3, 2/3); z
0.33333333333333333333333333333 + 0.66666666666666666666666666667*I
```

We can load and save complex numbers and the complex field:

```
sage: loads(z.dumps()) == z
True
sage: loads(CC.dumps()) == CC
True
sage: k = ComplexField(100)
sage: loads(dumps(k)) == k
True
```

This illustrates basic properties of a complex field:

```
sage: CC = ComplexField(200)
sage: CC.is_field()
True
sage: CC.characteristic()
0
sage: CC.precision()
200
sage: CC.variable_name()
'I'
sage: CC == ComplexField(200)
```

```
True
sage: CC == ComplexField(53)
False
sage: CC == 1.1
False
```

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
```

characteristic()

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

EXAMPLES:

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

construction()

Returns 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
```

gen(*n=0*)

Return the generator of the complex field.

EXAMPLES:

```
sage: ComplexField().gen(0)
1.00000000000000*I
```

is_exact()

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

EXAMPLES:

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

is_field(*proof=True*)

Return `True` since the complex numbers are a field.

EXAMPLES:

```
sage: CC.is_field()
True
```

is_finite()

Return False since there are infinite number of complex numbers.

EXAMPLES:

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

ngens()

The number of generators of this complex field as an $\mathbf{R}$-algebra.

There is one generator, namely sqrt(-1).

EXAMPLES:

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

pi()

Returns π as a complex number.

EXAMPLES:

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

prec()

Return the precision of this complex field.

EXAMPLES:

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

precision()

Return the precision of this complex field.

EXAMPLES:

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

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

Returns 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() for _ in range(5)]
[0.153636193785613 - 0.502987375247518*I,
 0.609589964322241 - 0.948854594338216*I,
 0.968393085385764 - 0.148483595843485*I,
 -0.908976099636549 + 0.126219184235123*I,
 0.461226845462901 - 0.0420335212948924*I]
sage: CC6 = ComplexField(6)
sage: [CC6.random_element(2^-20) for _ in range(5)]
```

```
[-5.4e-7 - 3.3e-7*I, 2.1e-7 + 8.0e-7*I, -4.8e-7 - 8.6e-7*I, -6.0e-8 + 2.7e-7*I, 6.0e-8 + 1.8
sage: [CC6.random_element(pi^20) for _ in range(5)]
[6.7e8 - 5.4e8*I, -9.4e8 + 5.0e9*I, 1.2e9 - 2.7e8*I, -2.3e9 - 4.0e9*I, 7.7e9 + 1.2e9*I]
```

Passes extra positional or keyword arguments through:

```
sage: [CC.random_element(distribution='1/n') for _ in range(5)]
[-0.900931453455899 - 0.932172283929307*I,
 0.327862582226912 + 0.828104487111727*I,
 0.246299162813240 + 0.588214960163442*I,
 0.892970599589521 - 0.266744694790704*I,
 0.878458776600692 - 0.905641181799996*I]
```

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))
0.0250000000000000 + 2.00000000000000*I
sage: C.scientific_notation(True)
sage: C((0.025, 2))
2.50000000000000e-2 + 2.00000000000000e0*I
sage: C.scientific_notation(False)
sage: C((0.025, 2))
0.0250000000000000 + 2.00000000000000*I
```

to_prec(*prec*)

Returns 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
```

zeta(*n=2*)

Return a primitive n-th root of unity.

INPUT:

- `n` - an integer (default: 2)

OUTPUT: a complex n-th root of unity.

EXAMPLES:

```
sage: C = ComplexField()
sage: C.zeta(2)
-1.00000000000000
sage: C.zeta(5)
0.309016994374947 + 0.951056516295154*I
```

`sage.rings.complex_field.`**is_ComplexField**(*x*)

Check if `x` is a `complex field`.

EXAMPLES:

```
sage: from sage.rings.complex_field import is_ComplexField as is_CF
sage: is_CF(ComplexField())
True
sage: is_CF(ComplexField(12))
True
sage: is_CF(CC)
True
```

sage.rings.complex_field.**late_import**()

Import the objects/modules after build (when needed).

TESTS:

```
sage: sage.rings.complex_field.late_import()
```

1.3 Arbitrary Precision Complex Numbers

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
- Travis Scrimshaw (2012-10-18): Added documentation for full coverage

class sage.rings.complex_number.**CCtoCDF**

Bases: sage.categories.map.Map

class sage.rings.complex_number.**ComplexNumber**

Bases: sage.structure.element.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
```

additive_order()

Return the additive order of self.

EXAMPLES:

```
sage: CC(0).additive_order()
1
sage: 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 sgm 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 $|a_1 - b_1| \le |a_1 + b_1|$, or equivalently $\Re(b_1/a_1) \ge 0$. 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 $\Re(b_1) \ge 0$ (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")
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")
-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()
0.490319232466314
```

TESTS:

An example which came up in testing:

```
sage: I = CC(I)
sage: a =  0.501648970493109 + 1.11877240294744*I
sage: b =  1.05946309435930 + 1.05946309435930*I
sage: a.agm(b)
0.774901870587681 + 1.10254945079875*I

sage: a = CC(-0.32599972608379413, 0.60395514542928641)
sage: b = CC( 0.6062314525690593,  0.1425693337776659)
```

```
sage: a.agm(b)
0.199246281325876 + 0.478401702759654*I
sage: a.agm(-a)
0.000000000000000
sage: a.agm(0)
0.000000000000000
sage: CC(0).agm(a)
0.000000000000000
```

Consistency:

```
sage: a = 1 + 0.5*I
sage: b = 2 - 0.25*I
sage: a.agm(b) - ComplexField(100)(a).agm(b)
0.000000000000000
sage: ComplexField(200)(a).agm(b) - ComplexField(500)(a).agm(b)
0.000000000000000000000000000000000000000000000000000000000000
sage: ComplexField(500)(a).agm(b) - ComplexField(1000)(a).agm(b)
0.0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

algdep(*n*, ***kwds*)

Returns 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 won't be if z is a good approximation to an algebraic number of degree less than n.

ALGORITHM: Uses the PARI C-library algdep command.

INPUT: Type algdep? at the top level prompt. All additional parameters are passed onto the top-level algdep command.

EXAMPLE:

```
sage: C = ComplexField()
sage: z = (1/2)*(1 + sqrt(3.0) *C.0); z
0.500000000000000 + 0.866025403784439*I
sage: p = z.algdep(5); p
x^3 + 1
sage: p.factor()
(x + 1) * (x^2 - x + 1)
sage: z^2 - z + 1
1.11022302462516e-16
```

algebraic_dependancy(*n*)

Returns 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 won't be if z is a good approximation to an algebraic number of degree less than n.

ALGORITHM: Uses the PARI C-library algdep command.

INPUT: Type algdep? at the top level prompt. All additional parameters are passed onto the top-level algdep command.

EXAMPLE:

```
sage: C = ComplexField()
sage: z = (1/2)*(1 + sqrt(3.0) *C.0); z
0.500000000000000 + 0.866025403784439*I
sage: p = z.algebraic_dependancy(5); p
x^3 + 1
sage: p.factor()
(x + 1) * (x^2 - x + 1)
```

```
sage: z^2 - z + 1
1.11022302462516e-16
```

arccos()

Return the arccosine of `self`.

EXAMPLES:

```
sage: (1+CC(I)).arccos()
0.904556894302381 - 1.06127506190504*I
```

arccosh()

Return the hyperbolic arccosine of `self`.

EXAMPLES:

```
sage: (1+CC(I)).arccosh()
1.06127506190504 + 0.904556894302381*I
```

arccoth()

Return the hyperbolic arccotangent of `self`.

EXAMPLES:

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

arccsch()

Return the hyperbolic arccosecant of `self`.

EXAMPLES:

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

arcsech()

Return the hyperbolic arcsecant of `self`.

EXAMPLES:

```
sage: ComplexField(100)(1,1).arcsech()
0.53063753095251782601650945811 - 1.1185178796437059371676632938*I
```

arcsin()

Return the arcsine of `self`.

EXAMPLES:

```
sage: (1+CC(I)).arcsin()
0.666239432492515 + 1.06127506190504*I
```

arcsinh()

Return the hyperbolic arcsine of `self`.

EXAMPLES:

```
sage: (1+CC(I)).arcsinh()
1.06127506190504 + 0.666239432492515*I
```

arctan()

Return the arctangent of `self`.

EXAMPLES:

```
sage: (1+CC(I)).arctan()
1.01722196789785 + 0.402359478108525*I
```

arctanh()

Return the hyperbolic arctangent of `self`.

EXAMPLES:

```
sage: (1+CC(I)).arctanh()
0.402359478108525 + 1.01722196789785*I
```

arg()

See `argument()`.

EXAMPLES:

```
sage: i = CC.0
sage: (i^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
```

conjugate()

Return the complex conjugate of this complex number.

EXAMPLES:

```
sage: i = CC.0
sage: (1+i).conjugate()
1.00000000000000 - 1.00000000000000*I
```

cos()

Return the cosine of `self`.

EXAMPLES:

```
sage: (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
```

cotan()

Return the cotangent of `self`.

EXAMPLES:

```
sage: (1+CC(I)).cotan()
0.217621561854403 - 0.868014142895925*I
sage: i = ComplexField(200).0
sage: (1+i).cotan()
0.21762156185440268136513424360523807352075436916785404091068 - 0.86801414289592494863584920
sage: i = ComplexField(220).0
sage: (1+i).cotan()
0.21762156185440268136513424360523807352075436916785404091068124239 - 0.8680141428959249486
```

coth()

Return the hyperbolic cotangent of `self`.

EXAMPLES:

```
sage: ComplexField(100)(1,1).coth()
0.86801414289592494863584920892 - 0.21762156185440268136513424361*I
```

csc()

Return the cosecant of `self`.

EXAMPLES:

```
sage: ComplexField(100)(1,1).csc()
0.62151801717042842123490780586 - 0.30393100162842645033448560451*I
```

csch()

Return the hyperbolic cosecant of `self`.

EXAMPLES:

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

dilog()

Returns 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()
1.64493406684823
sage: float(pi^2/6)
1.6449340668482262

sage: b = ComplexNumber(0,1)
sage: b.dilog()
-0.205616758356028 + 0.915965594177219*I

sage: c = ComplexNumber(0,0)
sage: c.dilog()
0.000000000000000
```

eta (*omit_frac=False*)

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

The η function is

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

INPUT:

- `self` – element of the upper half plane (if not, raises a `ValueError`).
- `omit_frac` – (bool, default: `False`), if `True`, omit the $e^{\pi i z/12}$ 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()
0.742048775836565 + 0.198831370229911*I
```

We compute eta to low precision directly from the definition:

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

The optional argument allows us to omit the fractional part:

```
sage: z = 1 + i
sage: z.eta(omit_frac=True)
0.998129069925959
sage: prod([1-exp(2*pi*i*n*z) for n in range(1,10)])
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()
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))
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()
1.46869393991588515713896759732660426132695673662900872279767567631093696585951213872272450
```

gamma()

Return the Gamma function evaluated at this complex number.

EXAMPLES:

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

TESTS:

```
sage: CC(0).gamma()
Infinity

sage: CC(-1).gamma()
Infinity
```

gamma_inc(*t*)

Return the incomplete Gamma function evaluated at this complex number.

EXAMPLES:

```
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
```

TESTS:

Check that trac ticket #7099 is fixed:

```
sage: C = ComplexField(400)
sage: C(2 + I).gamma_inc(C(3 + I))  # abs tol 1e-120
0.121515644664508695525971545977439666159749344176962379708992904126499444842886620664991650
```

imag()

Return imaginary part of `self`.

EXAMPLES:

```
sage: i = ComplexField(100).0
sage: z = 2 + 3*i
sage: x = z.imag(); x
3.0000000000000000000000000000
sage: x.parent()
Real Field with 100 bits of precision
sage: z.imag_part()
3.0000000000000000000000000000
```

imag_part()

Return imaginary part of `self`.

EXAMPLES:

```
sage: i = ComplexField(100).0
sage: z = 2 + 3*i
sage: x = z.imag(); x
3.0000000000000000000000000000
sage: x.parent()
Real Field with 100 bits of precision
sage: z.imag_part()
3.0000000000000000000000000000
```

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
```

is_infinity()

Check if `self` is ∞.

EXAMPLES:

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

is_integer()

Return `True` if `self` is a integer

EXAMPLES:

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

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
```

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
```

is_square()

This function always returns true as **C** is algebraically closed.

EXAMPLES:

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

C is algebraically closed, hence every element is a square:

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

log(*base=None*)

Complex logarithm of z with branch chosen as follows: Write $z = \rho e^{i\theta}$ with $-\pi < \theta <= pi$. 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
```

```
sage: b = ComplexNumber(float(exp(42)),0)
sage: b.log()
41.9999999999971
```

```
sage: c = ComplexNumber(-1,0)
sage: 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
```

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
```

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
0.500000000000000 + 0.866025403784439*I
sage: abs(w)
1.00000000000000
sage: w.multiplicative_order()
Traceback (most recent call last):
...
NotImplementedError: order of element not known
```

norm()

Returns 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:

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

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()
5.00000000000000
sage: b = ComplexNumber(4.2,0)
sage: b.norm()
17.6400000000000
```

```
sage: b^2
17.6400000000000
```

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

The n-th root function.

INPUT:

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

EXAMPLES:

```
sage: a = CC(27)
sage: a.nth_root(3)
3.00000000000000
sage: a.nth_root(3, all=True)
[3.00000000000000, -1.50000000000000 + 2.59807621135332*I, -1.50000000000000 - 2.5980762113
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
```

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)
Graphics object consisting of 1 graphics primitive
```

or the more direct:

```
sage: z = CC(0,1)
sage: z.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
```

real ()

Return real part of `self`.

EXAMPLES:

```
sage: i = ComplexField(100).0
sage: z = 2 + 3*i
sage: x = z.real(); x
2.0000000000000000000000000000
sage: x.parent()
Real Field with 100 bits of precision
```

```
sage: z.real_part()
2.0000000000000000000000000000
```

real_part()

Return real part of self.

EXAMPLES:

```
sage: i = ComplexField(100).0
sage: z = 2 + 3*i
sage: x = z.real(); x
2.0000000000000000000000000000
sage: x.parent()
Real Field with 100 bits of precision
sage: z.real_part()
2.0000000000000000000000000000
```

sec()

Return the secant of self.

EXAMPLES:

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

sech()

Return the hyperbolic secant of self.

EXAMPLES:

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

sin()

Return the sine of self.

EXAMPLES:

```
sage: (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
```

sqrt(*all=False*)

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

INPUT:

- all - bool (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 + 0.7071067811865475244008443
```

str(*base=10*, *truncate=True*, *istr='I'*)

Return a string representation of `self`.

INPUT:

- `base` – (Default: 10) The base to use for printing
- `truncate` – (Default: `True`) Whether to print fewer digits than are available, to mask errors in the last bits.
- `istr` – (Default: `I`) String representation of the complex unit

EXAMPLES:

```
sage: a = CC(pi + I*e)
sage: a.str()
'3.14159265358979 + 2.71828182845905*I'
sage: a.str(truncate=False)
'3.1415926535897931 + 2.7182818284590451*I'
sage: a.str(base=2)
'11.001001000011111101101010100010001000010110100011000 + 10.1011011111100001010100010110001
sage: CC(0.5 + 0.625*I).str(base=2)
'0.10000000000000000000000000000000000000000000000000000 + 0.101000000000000000000000000000
sage: a.str(base=16)
'3.243f6a8885a30 + 2.b7e151628aed2*I'
sage: a.str(base=36)
'3.53i5ab8p5fc + 2.puw5nggjf8f*I'
sage: CC(0)
0.000000000000000
sage: CC.0.str(istr='%i')
'1.00000000000000*%i'
```

tan()

Return the tangent of `self`.

EXAMPLES:

```
sage: (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
```

zeta()

Return the Riemann zeta function evaluated at this complex number.

EXAMPLES:

```
sage: i = ComplexField(30).gen()
sage: z = 1 + i
sage: z.zeta()
0.58215806 - 0.92684856*I
sage: zeta(z)
0.58215806 - 0.92684856*I

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

class `sage.rings.complex_number.`**`RRtoCC`**

Bases: `sage.categories.map.Map`

EXAMPLES:

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

`sage.rings.complex_number.`**`cmp_abs`**(*a*, *b*)

Returns -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_number 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))
-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
```

sage.rings.complex_number.**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 `ComplexField(prec=n)`, where n potentially has slightly more (controlled by pad) bits than given by s.

INPUT:

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

EXAMPLES:

```
sage: ComplexNumber('2.3')
2.30000000000000
sage: ComplexNumber('2.3','1.1')
2.30000000000000 + 1.10000000000000*I
sage: ComplexNumber(10)
10.0000000000000
sage: ComplexNumber(10,10)
10.0000000000000 + 10.0000000000000*I
sage: ComplexNumber(1.000000000000000000000000000,2)
1.000000000000000000000000000 + 2.000000000000000000000000000*I
sage: ComplexNumber(1,2.000000000000000000000)
1.000000000000000000000 + 2.000000000000000000000*I

sage: sage.rings.complex_number.create_ComplexNumber(s_real=2,s_imag=1)
2.00000000000000 + 1.00000000000000*I
```

TESTS:

Make sure we've rounded up `log(10,2)` enough to guarantee sufficient precision (trac ticket #10164):

```
sage: s = "1." + "0"*10**6 + "1"
sage: sage.rings.complex_number.create_ComplexNumber(s,0).real()-1 == 0
False
sage: sage.rings.complex_number.create_ComplexNumber(0,s).imag()-1 == 0
False
```

sage.rings.complex_number.**is_ComplexNumber**(*x*)

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

EXAMPLES:

```
sage: from sage.rings.complex_number import is_ComplexNumber
sage: a = ComplexNumber(1,2); a
1.00000000000000 + 2.00000000000000*I
sage: is_ComplexNumber(a)
True
sage: b = ComplexNumber(1); b
1.00000000000000
sage: is_ComplexNumber(b)
True
```

Note that the global element `I` is of type `SymbolicConstant`. However, elements of the class `ComplexField_class` are of type `ComplexNumber`:

```
sage: c = 1 + 2*I
sage: is_ComplexNumber(c)
False
sage: d = CC(1 + 2*I)
sage: is_ComplexNumber(d)
True
```

`sage.rings.complex_number.`**`make_ComplexNumber0`**(*fld*, *mult_order*, *re*, *im*)

Create a complex number for pickling.

EXAMPLES:

```
sage: a = CC(1 + I)
sage: loads(dumps(a)) == a # indirect doctest
True
```

`sage.rings.complex_number.`**`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_number.set_global_complex_round_mode(0)
```

1.4 Arbitrary Precision Complex Numbers using GNU MPC

This is a binding for the MPC arbitrary-precision floating point library. It is adaptated from `real_mpfr.pyx` and `complex_number.pyx`.

We define a class `MPComplexField`, where each instance of `MPComplexField` 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 `MPComplexNumber`.

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.
- 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.

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
```

class `sage.rings.complex_mpc.`**`CCtoMPC`**

Bases: `sage.categories.map.Map`

class `sage.rings.complex_mpc.`**`INTEGERtoMPC`**

Bases: `sage.categories.map.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:

```
sage: MPComplexField()
Complex Field with 53 bits of precision
sage: MPComplexField(100)
Complex Field with 100 bits of precision
sage: MPComplexField(100).base_ring()
Real Field with 100 bits of precision
sage: i = MPComplexField(200).gen()
sage: i^2
-1.00000000000000000000000000000000000000000000000000000000000
```

class `sage.rings.complex_mpc.`**`MPComplexField_class`**

Bases: `sage.rings.ring.Field`

Initialize `self`.

INPUT:

- `prec` – (integer) precision; default = 53

  prec is the number of bits used to represent the matissa of both the real and imaginary part of complex floating-point number.

- `rnd` – (string) the rounding mode; default = `'RNDNN'`

  Rounding mode is of the form `'RNDxy'` where `x` and `y` are the rounding mode for respectively the real and imaginary parts and are one of:

  - `'N'` for rounding to nearest
  - `'Z'` for rounding towards zero
  - `'U'` for rounding towards plus infinity
  - `'D'` for rounding towards minus infinity

  For example, `'RNDZU'` indicates to round the real part towards zero, and the imaginary part towards plus infinity.

EXAMPLES:

```
sage: MPComplexField(17)
Complex Field with 17 bits of precision
sage: MPComplexField()
Complex Field with 53 bits of precision
```

```
sage: MPComplexField(1042,'RNDDZ')
Complex Field with 1042 bits of precision and rounding RNDDZ
```

ALGORITHMS: Computations are done using the MPC library.

characteristic()

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

EXAMPLES:

```
sage: MPComplexField(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
```

is_exact()

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

EXAMPLES:

```
sage: MPComplexField(42).is_exact()
False
```

is_finite()

Return `False`, since the field of complex numbers is not finite.

EXAMPLES:

```
sage: MPComplexField(17).is_finite()
False
```

name()

Return the name of the complex field.

EXAMPLES:

```
sage: C = MPComplexField(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
```

prec()

Return the precision of this field of complex numbers.

EXAMPLES:

```
sage: MPComplexField().prec()
53
sage: MPComplexField(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
```

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'
```

rounding_mode_imag ()

Return rounding mode used for the imaginary part of complex number.

EXAMPLES:

```
sage: MPComplexField(rnd='RNDZU').rounding_mode_imag()
'RNDU'
```

rounding_mode_real ()

Return rounding mode used for the real part of complex number.

EXAMPLES:

```
sage: MPComplexField(rnd='RNDZU').rounding_mode_real()
'RNDZ'
```

class sage.rings.complex_mpc.**MPComplexNumber**

Bases: sage.structure.element.FieldElement

A floating point approximation to a complex number using any specified precision common to both real and imaginary part.

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

Returns the algebraic geometrc mean of self and right.

EXAMPLES:

```
sage: MPC = MPComplexField()
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
```

algebraic_dependancy (*n*, ***kwds*)

Returns 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 won't be if z is a good approximation to an algebraic number of degree less than n.

ALGORITHM: Uses the PARI C-library algdep command.

INPUT: Type `algdep?` at the top level prompt. All additional parameters are passed onto the top-level algdep command.

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: z = (1/2)*(1 + sqrt(3.0) * MPC.0); z
0.500000000000000 + 0.866025403784439*I
sage: p = z.algebraic_dependency(5)
sage: p.factor()
(x + 1) * (x^2 - x + 1)^2
sage: z^2 - z + 1
1.11022302462516e-16
```

algebraic_dependency(*n*, ***kwds*)

Returns 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 won't be if z is a good approximation to an algebraic number of degree less than n.

ALGORITHM: Uses the PARI C-library algdep command.

INPUT: Type `algdep?` at the top level prompt. All additional parameters are passed onto the top-level algdep command.

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: z = (1/2)*(1 + sqrt(3.0) * MPC.0); z
0.500000000000000 + 0.866025403784439*I
sage: p = z.algebraic_dependency(5)
sage: p.factor()
(x + 1) * (x^2 - x + 1)^2
sage: z^2 - z + 1
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
```

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
```

arccoth()

Return the hyperbolic arccotangent of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField(100)
sage: MPC(1,1).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
```

arcsech()

Return the hyperbolic arcsecant of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField(100)
sage: MPC(1,1).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
```

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
```

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
```

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
```

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
```

conjugate()

Return the complex conjugate of this complex number:

$$\mathrm{conjugate}(a + ib) = a - ib.$$

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: i = MPC(0, 1)
sage: (1+i).conjugate()
1.00000000000000 - 1.00000000000000*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
```

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
```

cotan()

Return the cotangent of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField(53)
sage: (1+MPC(I)).cotan()
0.217621561854403 - 0.868014142895925*I
```

```
sage: i = MPComplexField(200).0
sage: (1+i).cotan()
0.21762156185440268136513424360523807352075436916785404091068 - 0.86801414289592494863584920
sage: i = MPComplexField(220).0
sage: (1+i).cotan()
0.21762156185440268136513424360523807352075436916785404091068124239 - 0.868014142895924948636
```

coth()

Return the hyperbolic cotangent of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField(100)
sage: MPC(1,1).coth()
0.86801414289592494863584920892 - 0.21762156185440268136513424361*I
```

csc()

Return the cosecent of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField(100)
sage: MPC(1,1).csc()
0.62151801717042842123490780586 - 0.30393100162842645033448560451*I
```

csch()

Return the hyperbolic cosecent of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField(100)
sage: MPC(1,1).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 = MPComplexField()
sage: a = MPC(1,0)
sage: a.dilog()
1.64493406684823
sage: float(pi^2/6)
1.6449340668482262

sage: b = MPC(0,1)
sage: b.dilog()
-0.205616758356028 + 0.915965594177219*I

sage: c = MPC(0,0)
sage: c.dilog()
0
```

eta (*omit_frac=False*)

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

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
```

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
```

TESTS:

```
sage: MPC(0).gamma()
Infinity

sage: MPC(-1).gamma()
Infinity
```

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
```

imag()

Return imaginary part of `self`.

EXAMPLES:

```
sage: C = MPComplexField(100)
sage: z = C(2, 3)
sage: x = z.imag(); x
3.0000000000000000000000000000
sage: x.parent()
Real Field with 100 bits of precision
```

is_imaginary()

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

EXAMPLES:

```
sage: C200 = MPComplexField(200)
sage: C200(1.23*i).is_imaginary()
True
sage: C200(1+i).is_imaginary()
False
```

is_real()

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

EXAMPLES:

```
sage: C200 = MPComplexField(200)
sage: C200(1.23).is_real()
True
sage: C200(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
```

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

```
sage: b = C200(5)
sage: 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
```

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:

$$\mathrm{norm}(a + ib) = a^2 + b^2.$$

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()
5.00000000000000
sage: b = MPC(4.2,0)
sage: b.norm()
17.6400000000000
sage: b^2
17.6400000000000
```

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

The n-th root function.

INPUT:

- `all` - bool (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)
3.00000000000000
sage: a.nth_root(3, all=True)
[3.00000000000000, -1.50000000000000 + 2.59807621135332*I, -1.50000000000000 - 2.59807621135
```

prec()

Return precision of this complex number.

EXAMPLES:

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

real()

Return the real part of `self`.

EXAMPLES:

```
sage: C = MPComplexField(100)
sage: z = C(2, 3)
sage: x = z.real(); x
```

```
2.0000000000000000000000000000
sage: x.parent()
Real Field with 100 bits of precision
```

sec()
Return the secant of this complex number.

EXAMPLES:

```
sage: MPC = MPComplexField(100)
sage: MPC(1,1).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
```

sin()
Return the sine of this complex number:

$$\sin(a+ib) = \sin a \cosh b + i \cos x \sinh b.$$

EXAMPLES:

```
sage: MPC = MPComplexField()
sage: u = MPC(2, 4)
sage: 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
```

sqr()
Return the square of a complex number:

$$(a+ib)^2 = (a^2 - b^2) + 2iab.$$

EXAMPLES:

```
sage: C = MPComplexField()
sage: a = C(5, 1)
sage: a.sqr()
24.0000000000000 + 10.0000000000000*I
```

sqrt()
Return the square root, taking the branch cut to be the negative real axis:

$$\sqrt{z} = \sqrt{|z|}(\cos(\arg(z)/2) + i \sin(\arg(z)/2)).$$

EXAMPLES:

```
sage: C = MPComplexField()
sage: a = C(24, 10)
sage: a.sqrt()
5.00000000000000 + 1.00000000000000*I
```

str(*base=10*, *truncate=True*)

Return a string of `self`.

INPUT:

- `base` – base for output
- `truncate` – if `True`, round off the last digits in printing to lessen confusing base-2 roundoff issues.

EXAMPLES:

```
sage: MPC = MPComplexField(64)
sage: z = MPC(-4, 3)/7
sage: z.str()
'-0.571428571428571429 + 0.428571428571428571*I'
sage: z.str(16)
'-0.92492492492492490 + 0.6db6db6db6db6db70*I'
sage: z.str(truncate=True)
'-0.571428571428571429 + 0.428571428571428571*I'
sage: z.str(2, True)
'-0.1001001001001001001001001001001001001001001001001001001001001001 + 0.0110110110110110110
```

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
```

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
```

zeta()

Return the Riemann zeta function evaluated at this complex number.

EXAMPLES:

```
sage: i = MPComplexField(30).gen()
sage: z = 1 + i
sage: z.zeta()
0.58215806 - 0.92684856*I
```

class sage.rings.complex_mpc.**MPCtoMPC**
Bases: sage.categories.map.Map

section()
EXAMPLES:

```
sage: from sage.rings.complex_mpc import *
sage: C10 = MPComplexField(10)
sage: C100 = MPComplexField(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
```

class sage.rings.complex_mpc.**MPFRtoMPC**
Bases: sage.categories.map.Map

sage.rings.complex_mpc.**late_import**()
Import the objects/modules after build (when needed).

TESTS:

```
sage: sage.rings.complex_mpc.late_import()
```

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')
('123.', '+456e789')
sage: sage.rings.complex_mpc.split_complex_string('123.456e789', base=2)
(None, None)
```

1.5 Double Precision Real Numbers

EXAMPLES:

We create the real double vector space of dimension 3:

```
sage: V = RDF^3; V
Vector space of dimension 3 over Real Double Field
```

Notice that this space is unique:

```
sage: V is RDF^3
True
sage: V is FreeModule(RDF, 3)
True
sage: V is VectorSpace(RDF, 3)
True
```

Also, you can instantly create a space of large dimension:

```
sage: V = RDF^10000
```

TESTS:

Test NumPy conversions:

```
sage: RDF(1).__array_interface__
{'typestr': '=f8'}
sage: import numpy
sage: numpy.array([RDF.pi()]).dtype
dtype('float64')
```

class sage.rings.real_double.**RealDoubleElement**

Bases: sage.structure.element.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
```

abs()

Returns the absolute value of self.

EXAMPLES:

```
sage: RDF(1e10).abs()
10000000000.0
sage: RDF(-1e10).abs()
10000000000.0
```

acosh()

Return the hyperbolic inverse cosine of self.

EXAMPLES:

```
sage: q = RDF.pi()/2
sage: i = q.cosh(); i
2.5091784786580567
sage: abs(i.acosh()-q) < 1e-15
True
```

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
```

The arithmetic-geometric mean always lies between the geometric and arithmetic mean:

```
sage: sqrt(a*b) < a.agm(b) < (a+b)/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 `algdep` command.

EXAMPLE:

```
sage: r = sqrt(RDF(2)); r
1.4142135623730951
sage: r.algebraic_dependency(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 `algdep` command.

EXAMPLE:

```
sage: r = sqrt(RDF(2)); r
1.4142135623730951
sage: r.algebraic_dependency(5)
x^2 - 2
```

arccos()

Return the inverse cosine of `self`.

EXAMPLES:

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

arcsin()

Return the inverse sine of `self`.

EXAMPLES:

```
sage: q = RDF.pi()/5
sage: i = q.sin()
sage: i.arcsin() == q
True
```

arcsinh()

Return the hyperbolic inverse sine of `self`.

EXAMPLES:

```
sage: q = RDF.pi()/2
sage: i = q.sinh(); i
2.3012989023072947
sage: abs(i.arcsinh()-q) < 1e-15
True
```

arctan()

Return the inverse tangent of `self`.

EXAMPLES:

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

arctanh()

Return the hyperbolic inverse tangent of `self`.

EXAMPLES:

```
sage: q = RDF.pi()/2
sage: i = q.tanh(); i
0.9171523356672744
sage: i.arctanh() - q  # rel tol 1
4.440892098500626e-16
```

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
```

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
```

conjugate()

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

EXAMPLES:

```
sage: RDF(4).conjugate()
4.0
```

cos()

Return the cosine of `self`.

EXAMPLES:

```
sage: t=RDF.pi()/2
sage: t.cos()
6.123233995736757e-17
```

cosh()

Return the hyperbolic cosine of `self`.

EXAMPLES:

```
sage: q = RDF.pi()/12
sage: q.cosh()
1.0344656400955106
```

coth()

Return the hyperbolic cotangent of `self`.

EXAMPLES:

```
sage: RDF(pi).coth()
1.003741873197321
sage: CDF(pi).coth()
1.0037418731973213
```

csch()

Return the hyperbolic cosecant of `self`.

EXAMPLES:

```
sage: RDF(pi).csch()
0.08658953753004694
sage: CDF(pi).csch()  # rel tol 1e-15
0.08658953753004696
```

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
```

dilog()

Return the dilogarithm of `self`.

This is defined by the series $\sum_n x^n/n^2$ for $|x| \le 1$. When the absolute value of `self` is greater than 1, the returned value is the real part of (the analytic continuation to $\mathbf{C}$ of) the dilogarithm of `self`.

EXAMPLES:

```
sage: RDF(1).dilog()  # rel tol 1.0e-13
1.6449340668482264
sage: RDF(2).dilog()  # rel tol 1.0e-13
2.46740110027234
```

erf()

Return the value of the error function on `self`.

EXAMPLES:

```
sage: RDF(6).erf()
1.0
```

exp()

Return $e^{\texttt{self}}$.

EXAMPLES:

```
sage: r = RDF(0.0)
sage: r.exp()
1.0

sage: r = RDF('32.3')
sage: a = r.exp(); a
106588847274864.47
sage: a.log()
32.3

sage: r = RDF('-32.3')
sage: r.exp()
9.381844588498685e-15

sage: RDF(1000).exp()
+infinity
```

exp10()

Return $10^{\texttt{self}}$.

EXAMPLES:

```
sage: r = RDF(0.0)
sage: r.exp10()
1.0

sage: r = RDF(32.0)
sage: r.exp10()
1.0000000000000069e+32

sage: r = RDF(-32.3)
sage: r.exp10()
5.011872336272702e-33
```

exp2()

Return $2^{\texttt{self}}$.

EXAMPLES:

```
sage: r = RDF(0.0)
sage: r.exp2()
1.0

sage: r = RDF(32.0)
sage: r.exp2()
4294967295.9999967

sage: r = RDF(-32.3)
sage: r.exp2()
1.8911724825302065e-10
```

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
```

frac()

Return a real number in $(-1, 1)$. It satisfies the relation: `x = x.trunc() + x.frac()`

EXAMPLES:

```
sage: RDF(2.99).frac()
0.9900000000000002
sage: RDF(2.50).frac()
0.5
sage: RDF(-2.79).frac()
-0.79
```

gamma()

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

EXAMPLES:

```
sage: RDF(6).gamma()
120.0
sage: RDF(1.5).gamma()   # rel tol 1e-15
0.8862269254527584
```

hypot(*other*)

Computes the value $\sqrt{s^2 + o^2}$ where s is `self` and o is `other` in such a way as to avoid overflow.

EXAMPLES:

```
sage: x = RDF(4e300); y = RDF(3e300);
sage: x.hypot(y)
5e+300
sage: sqrt(x^2+y^2) # overflow
+infinity
```

imag()

Return the imaginary part of this number, which is zero.

EXAMPLES:

```
sage: a = RDF(3)
sage: a.imag()
0.0
```

integer_part()

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

EXAMPLES:

```
sage: r = RDF('-1.6')
sage: a = r.integer_part(); a
-1
sage: type(a)
<type '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
```

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
```

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
```

is_integer()

Return True if this number is a integer

EXAMPLES:

```
sage: RDF(3.5).is_integer()
False
sage: RDF(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
```

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
```

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 non-negative.

EXAMPLES:

```
sage: RDF(3.5).is_square()
True
sage: RDF(0).is_square()
True
sage: RDF(-4).is_square()
False
```

log(*base=None*)

Return the logarithm.

INPUT:

•`base` – integer or `None` (default). The base of the logarithm. If `None` is specified, the base is e (the so-called natural logarithm).

OUTPUT:

The logarithm of `self`. If `self` is positive, a double floating point number. Infinity if `self` is zero. A imaginary complex floating point number if `self` is negative.

EXAMPLES:

```
sage: RDF(2).log()
0.6931471805599453
sage: RDF(2).log(2)
1.0
sage: RDF(2).log(pi)
0.6055115613982801
sage: RDF(2).log(10)
0.30102999566398114
sage: RDF(2).log(1.5)
1.7095112913514547
sage: RDF(0).log()
-infinity
sage: RDF(-1).log()
3.141592653589793*I
sage: RDF(-1).log(2)  # rel tol 1e-15
4.532360141827194*I
```

TESTS:

Make sure that we can take the log of small numbers accurately and the fix doesn't break preexisting values (trac ticket #12557):

```
sage: R = RealField(128)
sage: def check_error(x):
....:   x = RDF(x)
....:   log_RDF = x.log()
....:   log_RR = R(x).log()
....:   diff = R(log_RDF) - log_RR
....:   if abs(diff) < log_RDF.ulp():
....:       return True
....:   print "logarithm check failed for %s (diff = %s ulp)"%           ....:        (x, d
....:   return False
sage: all( check_error(2^x) for x in range(-100,100) )
True
sage: all( check_error(x) for x in sxrange(0.01, 2.00, 0.01) )
```

```
True
sage: all( check_error(x) for x in sxrange(0.99, 1.01, 0.001) )
True
sage: RDF(1.000000001).log()
1.000000082240371e-09
sage: RDF(1e-17).log()
-39.14394658089878
sage: RDF(1e-50).log()
-115.12925464970229
```

log10()

Return log to the base 10 of `self`.

EXAMPLES:

```
sage: r = RDF('16.0'); r.log10()
1.2041199826559246
sage: r.log() / RDF(log(10))
1.2041199826559246
sage: r = RDF('39.9'); r.log10()
1.6009728956867482
```

log2()

Return log to the base 2 of `self`.

EXAMPLES:

```
sage: r = RDF(16.0)
sage: r.log2()
4.0

sage: r = RDF(31.9); r.log2()
4.995484518877507
```

logpi()

Return log to the base π of `self`.

EXAMPLES:

```
sage: r = RDF(16); r.logpi()
2.4220462455931204
sage: r.log() / RDF(log(pi))
2.4220462455931204
sage: r = RDF('39.9'); r.logpi()
3.2203023346075152
```

multiplicative_order()

Returns n such that `self^n == 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
```

nan()

Return Not-a-Number NaN.

EXAMPLES:

```
sage: RDF.NaN()
NaN
```

nth_root(*n*)

Return the n^{th} root of self.

INPUT:

- n – an integer

OUTPUT:

The output is a complex double if self is negative and n is even, otherwise it is a real double.

EXAMPLES:

```
sage: r = RDF(-125.0); r.nth_root(3)
-5.000000000000001
sage: r.nth_root(5)
-2.6265278044037674
sage: RDF(-2).nth_root(5)^5  # rel tol 1e-15
-2.000000000000001
sage: RDF(-1).nth_root(5)^5
-1.0
sage: RDF(3).nth_root(10)^10
2.9999999999999982
sage: RDF(-1).nth_root(2)
6.123233995736757e-17 + 1.0*I
sage: RDF(-1).nth_root(4)
0.7071067811865476 + 0.7071067811865475*I
```

prec()

Return the precision of this number in bits.

Always returns 53.

EXAMPLES:

```
sage: RDF(0).prec()
53
```

real()

Return self - we are already real.

EXAMPLES:

```
sage: a = RDF(3)
sage: a.real()
3.0
```

restrict_angle()

Return a number congruent to self mod 2π that lies in the interval $(-\pi, \pi]$.

Specifically, it is the unique $x \in (-\pi, \pi]$ such that `self $= x + 2\pi n$ for some $n \in \mathbf{Z}$.

EXAMPLES:

```
sage: RDF(pi).restrict_angle()
3.141592653589793
sage: RDF(pi + 1e-10).restrict_angle()
-3.1415926534897936
```

```
sage: RDF(1+10^10*pi).restrict_angle()
0.9999977606...
```

round()

Given real number x, rounds up if fractional part is greater than 0.5, rounds down if fractional part is less than 0.5.

EXAMPLES:

```
sage: RDF(0.49).round()
0
sage: a=RDF(0.51).round(); a
1
```

sech()

Return the hyperbolic secant of `self`.

EXAMPLES:

```
sage: RDF(pi).sech()
0.08626673833405443
sage: CDF(pi).sech()
0.08626673833405443
```

sign()

Returns -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
```

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: 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
```

The mantissa is always a nonnegative number:

```
sage: RDF(-1).sign_mantissa_exponent()
(-1, 4503599627370496, -52)
```

TESTS:

```
sage: RDF('+0').sign_mantissa_exponent()
(1, 0, 0)
sage: RDF('-0').sign_mantissa_exponent()
(-1, 0, 0)
```

sin()

Return the sine of `self`.

EXAMPLES:

```
sage: RDF(2).sin()
0.9092974268256817
```

sincos()

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

EXAMPLES:

```
sage: t = RDF.pi()/6
sage: t.sincos()
(0.49999999999999994, 0.8660254037844387)
```

sinh()

Return the hyperbolic sine of `self`.

EXAMPLES:

```
sage: q = RDF.pi()/12
sage: q.sinh()
0.26480022760227073
```

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

The square root function.

INPUT:

- `extend` – bool (default: `True`); if `True`, return a square root in a complex field if necessary if `self` is negative; otherwise raise a `ValueError`.
- `all` – bool (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

sage: r = RDF(4344)
sage: r.sqrt()
65.90902821313632
sage: r.sqrt()^2 - r
0.0

sage: r = RDF(-2.0)
sage: r.sqrt()
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)
[1.4142135623730951*I, -1.4142135623730951*I]
```

str()

Return the informal string representation of `self`.

EXAMPLES:

```
sage: a = RDF('4.5'); a.str()
'4.5'
sage: a = RDF('49203480923840.2923904823048'); a.str()
'4.92034809238e+13'
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'
```

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
```

tan()

Return the tangent of `self`.

EXAMPLES:

```
sage: q = RDF.pi()/3
sage: q.tan()
1.7320508075688767
sage: q = RDF.pi()/6
sage: q.tan()
0.5773502691896256
```

tanh()

Return the hyperbolic tangent of `self`.

EXAMPLES:

```
sage: q = RDF.pi()/12
sage: q.tanh()
0.25597778924568454
```

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
```

ulp()

Returns 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)
sage: a.ulp()
4.440892098500626e-16
sage: b = a + a.ulp()
```

Adding or subtracting an ulp always gives a different number:

```
sage: a + a.ulp() == a
False
sage: a - a.ulp() == a
False
sage: b + b.ulp() == b
False
sage: 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: 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
```

The ulp is always positive:

```
sage: RDF(-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
```

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
```

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
```

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: RDF(2).zeta()  # rel tol 1e-15
1.6449340668482269
sage: RDF.pi()^2/6
1.6449340668482264
sage: RDF(-2).zeta()
0.0
sage: RDF(1).zeta()
+infinity
```

sage.rings.real_double.**RealDoubleField**()

Return the unique instance of the `real double field`.

EXAMPLES:

```
sage: RealDoubleField() is RealDoubleField()
True
```

class sage.rings.real_double.**RealDoubleField_class**

Bases: `sage.rings.ring.Field`

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
False
sage: RDF == RealDoubleField()    # RDF is the shorthand
```

```
True

sage: RDF(1)
1.0
sage: RDF(2/3)
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 coerce nonconstant polynomial to float
sage: RDF(QQ['x'](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: 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
```

NaN()

Return Not-a-Number `NaN`.

EXAMPLES:

```
sage: RDF.NaN()
NaN
```

algebraic_closure()

Return the algebraic closure of `self`, i.e., the complex double field.

EXAMPLES:

```
sage: RDF.algebraic_closure()
Complex Double Field
```

characteristic()

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

EXAMPLES:

```
sage: 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()
Complex Double Field
```

construction()
Returns 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
sage: RDF == c(S)
True
```

euler_constant()
Return Euler's gamma constant to double precision.

EXAMPLES:

```
sage: 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
```

gen(*n=0*)
Return the generator of the real double field.

EXAMPLES:

```
sage: RDF.0
1.0
sage: RDF.gens()
(1.0,)
```

is_exact()
Returns `False`, because doubles are not exact.

EXAMPLE:

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

is_finite()
Return `False`, since the field of real numbers is not finite.

Technical note: There exists an upper bound on the double representation.

EXAMPLES:

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

log2()
Return $\log(2)$ to the precision of this field.

EXAMPLES:

```
sage: RDF.log2()
0.6931471805599453
```

```
sage: RDF(2).log()
0.6931471805599453
```

name()

The name of self.

EXAMPLES:

```
sage: RDF.name()
'RealDoubleField'
```

nan()

Return Not-a-Number NaN.

EXAMPLES:

```
sage: RDF.NaN()
NaN
```

ngens()

Return the number of generators which is always 1.

EXAMPLES:

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

pi()

Returns π to double-precision.

EXAMPLES:

```
sage: RDF.pi()
3.141592653589793
sage: RDF.pi().sqrt()/2
0.8862269254527579
```

prec()

Return the precision of this real double field in bits.

Always returns 53.

EXAMPLES:

```
sage: RDF.precision()
53
```

precision()

Return the precision of this real double field in bits.

Always returns 53.

EXAMPLES:

```
sage: 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()
0.7369454235661859
sage: RDF.random_element(min=100, max=110)
102.8159473516245
```

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)
Real Field with 52 bits of precision
sage: RDF.to_prec(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
```

class `sage.rings.real_double.`**ToRDF**

Bases: `sage.categories.morphism.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 type '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 type 'float'
  To:   Real Double Field
sage: f(3.5)
```

```
3.5
```

`sage.rings.real_double.`**`is_RealDoubleElement`**`(x)`

Check if `x` is an element of the real double field.

EXAMPLE:

```
sage: from sage.rings.real_double import is_RealDoubleElement
sage: is_RealDoubleElement(RDF(3))
True
sage: is_RealDoubleElement(RIF(3))
False
```

`sage.rings.real_double.`**`is_RealDoubleField`**`(x)`

Returns `True` if `x` is the field of real double precision numbers.

EXAMPLES:

```
sage: from sage.rings.real_double import is_RealDoubleField
sage: is_RealDoubleField(RDF)
True
sage: is_RealDoubleField(RealField(53))
False
```

`sage.rings.real_double.`**`pool_stats`**`()`

Statistics for the real double pool.

EXAMPLES:

We first pull all elements from the pool (making sure it is empty to illustrate how the pool works):

```
sage: from sage.rings.real_double import time_alloc_list, pool_stats
sage: L = time_alloc_list(50)
sage: pool_stats()
Used pool 0 / 0 times
Pool contains 0 / 50 items
```

During the operation (in this example, addition), we end up with two temporary elements. After completion of the operation, they are added to the pool:

```
sage: RDF(2.1) + RDF(2.2)
4.300000000000001
sage: pool_stats()
Used pool 0 / 0 times
Pool contains 2 / 50 items
```

Next when we call `time_alloc_list()`, the "created" elements are actually pulled from the pool:

```
sage: time_alloc_list(3)
[2.2, 2.1, 0.0]
```

Note that the number of objects left in the pool depends on the garbage collector:

```
sage: pool_stats()
Used pool 0 / 0 times
Pool contains 1 / 50 items
```

`sage.rings.real_double.`**`time_alloc`**`(n)`

Allocate `n` `RealDoubleElement` instances.

EXAMPLES:

Since this does not store anything in a python object, the created elements will not be sent to the garbage collector. Therefore they remain in the pool:

```
sage: from sage.rings.real_double import time_alloc, pool_stats
sage: pool_stats()
Used pool 0 / 0 times
Pool contains 7 / 50 items
sage: time_alloc(25)
sage: pool_stats()
Used pool 0 / 0 times
Pool contains 7 / 50 items
```

sage.rings.real_double.**time_alloc_list**(*n*)

Allocate a list of length n of RealDoubleElement instances.

EXAMPLES:

During the operation (in this example, addition), we end up with two temporary elements. After completion of the operation, they are added to the pool:

```
sage: from sage.rings.real_double import time_alloc_list
sage: RDF(2.1) + RDF(2.2)
4.300000000000001
```

Next when we call time_alloc_list(), the "created" elements are actually pulled from the pool:

```
sage: time_alloc_list(2)
[2.2, 2.1]
```

1.6 Double Precision 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)
<type 'sage.rings.complex_double.ComplexDoubleElement'>
sage: ComplexDoubleElement(sqrt(2),3)
1.4142135623730951 + 3.0*I
sage: parent(CDF(-2))
Complex Double Field
```

```
sage: CC == CDF
False
sage: CDF is ComplexDoubleField()     # CDF is the shorthand
True
sage: 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 (trac ticket #14082)

class `sage.rings.complex_double.`**`ComplexDoubleElement`**

Bases: `sage.structure.element.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
```

`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
```

`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 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 $|a_1 - b_1| \le |a_1 + b_1|$, or equivalently $\Re(b_1/a_1) \ge 0$. 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).

EXAMPLES:

```
sage: i = CDF(I)
sage: (1+i).agm(2-i)  # rel tol 1e-15
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
```

Some degenerate cases:

```
sage: CDF(0).agm(a)
0.0
sage: a.agm(0)
0.0
sage: a.agm(-a)
0.0
```

algdep(n)

Returns 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 won't be if z is a good approximation to an algebraic number of degree less than n.

ALGORITHM: Uses the PARI C-library algdep command.

EXAMPLES:

```
sage: z = (1/2)*(1 + RDF(sqrt(3)) *CDF.0); z   # abs tol 1e-16
0.5 + 0.8660254037844387*I
sage: p = z.algdep(5); p
x^3 + 1
sage: p.factor()
(x + 1) * (x^2 - x + 1)
sage: abs(z^2 - z + 1) < 1e-14
True

sage: CDF(0,2).algdep(10)
x^2 + 4
sage: CDF(1,5).algdep(2)
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
```

arccosh()

This function returns the complex hyperbolic arccosine of the complex number z, $\operatorname{arccosh}(z)$. The branch cut is on the real axis, less than 1.

EXAMPLES:

```
sage: CDF(1,1).arccosh()
1.0612750619050357 + 0.9045568943023814*I
```

arccot()

This function returns the complex arccotangent of the complex number z, $\operatorname{arccot}(z) = \arctan(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arccot()   # rel tol 1e-15
0.5535743588970452 - 0.4023594781085251*I
```

arccoth()

This function returns the complex hyperbolic arccotangent of the complex number z, $\operatorname{arccoth}(z) = \operatorname{arctanh}(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arccoth()   # rel tol 1e-15
0.4023594781085251 - 0.5535743588970452*I
```

arccsc()

This function returns the complex arccosecant of the complex number z, $\operatorname{arccsc}(z) = \arcsin(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arccsc()   # rel tol 1e-15
0.45227844715119064 - 0.5306375309525178*I
```

arccsch()

This function returns the complex hyperbolic arccosecant of the complex number z, $\operatorname{arccsch}(z) = \arcsin(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arccsch()   # rel tol 1e-15
0.5306375309525178 - 0.45227844715119064*I
```

arcsec()

This function returns the complex arcsecant of the complex number z, $\operatorname{arcsec}(z) = \arccos(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arcsec()   # rel tol 1e-15
1.118517879643706 + 0.5306375309525178*I
```

arcsech()

This function returns the complex hyperbolic arcsecant of the complex number z, $\operatorname{arcsech}(z) = \operatorname{arccosh}(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arcsech()   # rel tol 1e-15
0.5306375309525176 - 1.118517879643706*I
```

arcsin()

This function returns the complex arcsine of the complex number z, $\arcsin(z)$. 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
```

arcsinh()

This function returns the complex hyperbolic arcsine of the complex number z, $\operatorname{arcsinh}(z)$. The branch cuts are on the imaginary axis, below $-i$ and above i.

EXAMPLES:

```
sage: CDF(1,1).arcsinh()
1.0612750619050357 + 0.6662394324925152*I
```

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
```

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
```

arg()

This function returns the argument of `self`, the complex number z, denoted by $\arg(z)$, where $-\pi < \arg(z) <= \pi$.

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
```

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()
1.5707963267948966
sage: CDF(-1).argument()
3.141592653589793
sage: CDF(-1 - 0.000001*i).argument()
-3.1415916535897934
```

conj()

This function returns the complex conjugate of the complex number z:

$$\overline{z} = x - iy.$$

EXAMPLES:

```
sage: z = CDF(2,3); z.conj()
2.0 - 3.0*I
```

conjugate()

This function returns the complex conjugate of the complex number z:

$$\overline{z} = x - iy.$$

EXAMPLES:

```
sage: z = CDF(2,3); 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
```

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
```

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
```

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
```

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
```

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
```

dilog()

Returns 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()
-0.059474798673809476 + 2.0726479717747566*I
sage: CDF(10000000,10000000).dilog()
-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` - (bool, default: `False`), if `True`, omit the $e^{\pi i z/12}$ factor.

OUTPUT: a complex double number

ALGORITHM: Uses the PARI C library.

The η function is

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

EXAMPLES:

We compute a few values of `eta()`:

```
sage: CDF(0,1).eta()
0.7682254223260566
sage: CDF(1,1).eta()
0.7420487758365647 + 0.1988313702299107*I
sage: CDF(25,1).eta()
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
-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)
0.9981290699259585
```

We compute `eta()` to low precision directly from the definition:

```
sage: z = CDF(1,1); z.eta()
0.7420487758365647 + 0.1988313702299107*I
sage: i = CDF(0,1); pi = CDF(pi)
sage: exp(pi * i * z / 12) * prod([1-exp(2*pi*i*n*z) for n in range(1,10)])
0.7420487758365647 + 0.19883137022991068*I
```

The optional argument allows us to omit the fractional part:

```
sage: z.eta(omit_frac=True)
0.9981290699259585
sage: pi = CDF(pi)
sage: prod([1-exp(2*pi*i*n*z) for n in range(1,10)])  # abs tol 1e-12
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
```

You can also use functional notation:

```
sage: z = CDF(1,1)
sage: eta(z)
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
```

We numerically verify a famous identity to the precision of a double:

```
sage: z = CDF(0, 2*pi); z
6.283185307179586*I
```

```
sage: exp(z)  # rel tol 1e-4
1.0 - 2.4492935982947064e-16*I
```

gamma()

Return the gamma function $\Gamma(z)$ evaluated at `self`, the complex number z.

EXAMPLES:

```
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
```

gamma_inc(*t*)

Return the incomplete gamma function evaluated at this complex number.

EXAMPLES:

```
sage: CDF(1,1).gamma_inc(CDF(2,3))
0.0020969148636468277 - 0.059981913655449706*I
sage: CDF(1,1).gamma_inc(5)
-0.001378130936215849 + 0.006519820023119819*I
sage: CDF(2,0).gamma_inc(CDF(1,1))
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
```

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
```

is_infinity()

Check if `self` is ∞.

EXAMPLES:

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

is_integer()

Returns True if this number is a integer

EXAMPLES:

```
sage: CDF(0.5).is_integer()
False
sage: CDF(I).is_integer()
False
sage: CDF(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
```

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
```

is_square()

This function always returns `True` as **C** is algebraically closed.

EXAMPLES:

```
sage: CDF(-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
```

This is the only example different from the GSL:

```
sage: CDF(0,0).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
```

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
```

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(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

sage: log(abs(ComplexField(200)(1.1,0.1)))
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:

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

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_number.ComplexNumber.norm()`

EXAMPLES:

```
sage: CDF(2,3).norm()
13.0
```

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

The n-th root function.

INPUT:

- all – bool (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 + 1.9
sage: abs(sum(a.nth_root(111, all=True)))  # rel tol 0.1
1.1057313523818259e-13
```

prec ()

Returns the precision of this number (to be more similar to ComplexNumber). Always returns 53.

EXAMPLES:

```
sage: CDF(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
```

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
```

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
```

sech ()

This function returns the complex hyperbolic secant of the complex number z:

$$\text{sech}(z) = \frac{1}{\cosh(z)}.$$

EXAMPLES:

```
sage: CDF(1,1).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
```

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
```

sqrt(*all=False*, ***kwds*)

The square root function.

INPUT:

- `all` - bool (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
```

We compute the square root of -1:

```
sage: a = CDF(-1)
sage: 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]
```

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
```

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
```

zeta()

Return the Riemann zeta function evaluated at this complex number.

EXAMPLES:

```
sage: z = CDF(1, 1)
sage: z.zeta()
0.5821580597520036 - 0.9268485643308071*I
sage: zeta(z)
0.5821580597520036 - 0.9268485643308071*I
sage: zeta(CDF(1))
Infinity
```

sage.rings.complex_double.**ComplexDoubleField**()

Returns the field of double precision complex numbers.

EXAMPLE:

```
sage: ComplexDoubleField()
Complex Double Field
sage: ComplexDoubleField() is CDF
True
```

class sage.rings.complex_double.**ComplexDoubleField_class**

Bases: sage.rings.ring.Field

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()

Returns the algebraic closure of self, i.e., the complex double field.

EXAMPLES:

```
sage: CDF.algebraic_closure()
Complex Double Field
```

characteristic()

Return the characteristic of the complex double field, which is 0.

EXAMPLES:

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

construction()

Returns 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
```

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
```

is_exact()

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

EXAMPLES:

```
sage: 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
```

pi()

Returns π as a double precision complex number.

EXAMPLES:

```
sage: 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
```

precision()

Return the precision of this complex double field (to be more similar to ComplexField). Always returns 53.

EXAMPLES:

```
sage: 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()
-0.43681052967509904 + 0.7369454235661859*I
sage: CDF.random_element(-10,10,-10,10)
-7.088740263015161 - 9.54135400334003*I
sage: CDF.random_element(-10^20,10^20,-2,2)
-7.587654737635711e+19 + 0.925549022838656*I
```

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
```

to_prec(*prec*)

Returns 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
```

zeta(*n=2*)

Return a primitive n-th root of unity in this CDF, for $n \geq 1$.

INPUT:

- n – a 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)
1.0
sage: CDF.zeta()
-1.0
sage: CDF.zeta() == CDF.zeta(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
```

class `sage.rings.complex_double.`**`ComplexToCDF`**

Bases: `sage.categories.morphism.Morphism`

Fast morphism for anything such that the elements have attributes `.real` and `.imag` (e.g. numpy complex types).

EXAMPLES:

```
sage: import numpy
sage: f = CDF.coerce_map_from(numpy.complex_)
sage: f(numpy.complex_(I))
1.0*I
sage: f(numpy.complex_(I)).parent()
Complex Double Field
```

class `sage.rings.complex_double.`**`FloatToCDF`**

Bases: `sage.categories.morphism.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 type '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 type 'float'
  To:   Complex Double Field
sage: f(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)
False
sage: is_ComplexDoubleElement(CDF(0))
True
```

sage.rings.complex_double.**is_ComplexDoubleField**(*x*)

Return True if x is the complex double field.

EXAMPLE:

```
sage: from sage.rings.complex_double import is_ComplexDoubleField
sage: is_ComplexDoubleField(CDF)
True
sage: is_ComplexDoubleField(ComplexField(53))
False
```

CHAPTER

TWO

INTERVAL ARITHMETIC

Sage implements real and complex interval arithmetic using MPFI (RealIntervalField, ComplexIntervalField) and arb (RealBallField, ComplexBallField).

2.1 Arbitrary Precision Real Intervals

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): trac ticket #3893: `random_element()` should pass on `*args` and `**kwds`.
- Travis Scrimshaw (2012-10-20): Fixing scientific notation output to fix trac ticket #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))
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
```

```
sage: RIF(123, 125)
124.?
sage: RIF(123, 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.?2'
sage: RIF(123, 127).str(error_digits=1)
'125.?2'
sage: v = RIF(-e, pi); v
0.?e1
sage: v.str(error_digits=1)
'1.?4'
sage: v.str(error_digits=5)
'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'
```

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 .. 1.2]` (which does not even contain the true value, 1.25), or to print it as `[1.2 .. 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 == 1.25
True
sage: a == 2
False
```

COMPARISONS:

Comparison operations (==, !=, <, <=, >, >=) return `True` if every value in the first interval has the given relation to every value in the second interval. The `cmp(a, b)` function works differently; it compares two intervals lexicographically. (However, the behavior is not specified if given a non-interval and an 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.)

- if `a > b`, then `cmp(a, b) == 1`; if `a == b`, then `cmp(a,b) == 0`; if `a < b`, then `cmp(a, b) == -1`
- `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
- It is not the case that if `cmp(a, b) == 0` then `a == b`, or that if `cmp(a, b) == 1` then `a > b`, or that if `cmp(a, b) == -1` then `a < b`
- There are values `a` and `b` such that `a <= b` but neither `a < b` nor `a == b` hold.

Note: Intervals `a` and `b` overlap iff `not(a != b)`.

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
sage: cmp(RIF(0), RIF(0, 1))
-1
sage: cmp(RIF(0, 1), RIF(0))
1
sage: cmp(RIF(0, 1), RIF(1))
-1
sage: cmp(RIF(0, 1), RIF(0, 1))
0
```

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
```

Comparison by equality shows what the semi-infinite intervals actually coerce to:

```
sage: RIF(1,oo) == oo
True
sage: RIF(-oo,-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
```

TESTS:

Comparisons with numpy types are right (see trac ticket #17758 and trac ticket #18076):

```
sage: import numpy
sage: RIF(0,1) < numpy.float('2')
True
sage: RIF(0,1) <= numpy.float('1')
True
sage: RIF(0,1) <= numpy.float('0.5')
False
sage: RIF(2) == numpy.int8('2')
True
sage: numpy.int8('2') == RIF(2)
True
```

`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` – a 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` – an integer between 2 and 36
- `pad` – (default: 0) an integer
- `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('1.0000000000000000000000000000000000')
1
sage: RealInterval('1.2345678901234567890123456789012345')
1.23456789012345678901234567890123450?
sage: RealInterval(29308290382930840239842390482, 3^20).str(style='brackets')
'[3.48678440100000000000000000e9 .. 2.93082903829308402398423904820e28]'
```

TESTS:

Make sure we've rounded up `log(10,2)` enough to guarantee sufficient precision (trac ticket #10164). This is a little tricky because at the time of writing, we don't support intervals long enough to trip the error. However, at least we can make sure that we either do it correctly or fail noisily:

```
sage: ks = 5*10**5, 10**6
sage: for k in ks:
...       try:
...           z = RealInterval("1." + "1"*k)
...           assert len(str(z))-4 >= k
...       except TypeError:
...           pass
```

sage.rings.real_mpfi.**RealIntervalField**(*prec=53*, *sci_not=False*)

Construct a RealIntervalField_class, with caching.

INPUT:

- prec – (integer) precision; default = 53: 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 – (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
```

See the documentation for RealIntervalField_class for many more examples.

class sage.rings.real_mpfi.**RealIntervalFieldElement**

Bases: sage.structure.element.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()
2.14159265358979
```

alea()

Return a floating-point number picked at random from the interval.

EXAMPLES:

```
sage: RIF(1, 2).alea() # random
1.34696133696137
```

algdep(*n*)

Returns 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:

Uses the PARI C-library algdep command.

EXAMPLES:

```
sage: r = sqrt(RIF(2)); r
1.414213562373095?
sage: r.algdep(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.algdep(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.algdep(5)
x^2 - 2
```

Even on this extremely imprecise interval we get an answer which is technically correct:

```
sage: RIF(-1, 1).algdep(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
```

arccosh()

Return the hyperbolic inverse cosine of `self`.

EXAMPLES:

```
sage: q = RIF.pi()/2
sage: 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
```

arccsch()

Return the inverse hyperbolic cosecant of `self`.

EXAMPLES:

```
sage: RealIntervalField(100)(2).arccsch()
0.481211825059603447497758913425?
sage: (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
```

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
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
```

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
```

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
```

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
```

bisection()

Returns 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()
(1.00000000000000, 1.50000000000000)
sage: b.lower(), b.upper()
(1.50000000000000, 2.00000000000000)

sage: I = RIF(e, pi)
sage: a, b = I.bisection()
sage: a.intersection(b) == I.center()
True
sage: a.union(b).endpoints() == I.endpoints()
True
```

ceil()

Return the celing 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')
'[-12.000000000 .. -10.000000000]'
sage: a.ceil()
-10.?
sage: ceil(a).str(style='brackets')
'[-11.000000000 .. -9.0000000000]'
```

ceiling()

Return the celing 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')
'[-12.000000000 .. -10.000000000]'
sage: a.ceil()
-10.?
sage: ceil(a).str(style='brackets')
'[-11.000000000 .. -9.0000000000]'
```

center()

Compute the center of the interval $[a..b]$ which is $(a+b)/2$.

EXAMPLES:

```
sage: RIF(1, 2).center()
1.50000000000000
```

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
```

cos()

Return the cosine of self.

EXAMPLES:

```
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?
```

TESTS:

This looped forever with an earlier version of MPFI, but now it works:

```
sage: RIF(-1, 1).cos().str(style='brackets')
'[0.54030230586813965 .. 1.0000000000000000]'
```

cosh()

Return the hyperbolic cosine of self.

EXAMPLES:

```
sage: q = RIF.pi()/12
sage: q.cosh()
1.034465640095511?
```

cot()

Return the cotangent of self.

EXAMPLES:

```
sage: RealIntervalField(100)(2).cot()
-0.457657554360285763750277410432?
```

coth()

Return the hyperbolic cotangent of self.

EXAMPLES:

```
sage: RealIntervalField(100)(2).coth()
1.03731472072754809587780976477?
```

csc()

Return the cosecant of self.

EXAMPLES:

```
sage: RealIntervalField(100)(2).csc()
1.099750170294616466756697397026?
```

csch()

Return the hyperbolic cosecant of `self`.

EXAMPLES:

```
sage: RealIntervalField(100)(2).csch()
0.275720564771783207758351482163?
```

diameter()

If 0 is in `self`, then return `absolute_diameter()`, otherwise return `relative_diameter()`.

EXAMPLES:

```
sage: RIF(1, 2).diameter()
0.666666666666667
sage: RIF(1, 2).absolute_diameter()
1.00000000000000
sage: RIF(1, 2).relative_diameter()
0.666666666666667
sage: RIF(pi).diameter()
1.41357985842823e-16
sage: RIF(pi).absolute_diameter()
4.44089209850063e-16
sage: RIF(pi).relative_diameter()
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
```

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()
(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()
(1.00000000000000, 2.00000000000000)
sage: RIF(pi).endpoints()
(3.14159265358979, 3.14159265358980)
sage: a = CIF(RIF(1,2), RIF(3,4))
sage: a.real().endpoints()
(1.00000000000000, 2.00000000000000)
```

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
```

exp()

Returns $e^{\texttt{self}}$

EXAMPLES:

```
sage: r = RIF(0.0)
sage: r.exp()
1

sage: r = RIF(32.3)
sage: a = r.exp(); a
1.065888472748645?e14
sage: a.log()
32.30000000000000?

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

exp2()

Returns $2^{\texttt{self}}$

EXAMPLES:

```
sage: r = RIF(0.0)
sage: r.exp2()
1

sage: r = RIF(32.0)
sage: r.exp2()
4294967296

sage: r = RIF(-32.3)
sage: 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?
```

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
```

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.

See also:

- `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:

```
sage: R = RealIntervalField()
sage: (2.99).floor()
2
sage: (2.00).floor()
2
sage: floor(RR(-5/2))
-3
sage: R = RealIntervalField(100)
sage: a = R(9.5, 11.3); a.str(style='brackets')
'[9.5000000000000000000000000000000 .. 11.300000000000000710542735760101]'
sage: floor(a).str(style='brackets')
'[9.0000000000000000000000000000000 .. 11.000000000000000000000000000000]'
sage: a.floor()
10.?
sage: ceil(a)
11.?
sage: a.ceil().str(style='brackets')
'[10.000000000000000000000000000000 .. 12.000000000000000000000000000000]'
```

fp_rank_diameter()

Computes 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(pi).fp_rank_diameter()
1
sage: RIF(12345).fp_rank_diameter()
0
sage: RIF(-sqrt(2)).fp_rank_diameter()
```

```
1
sage: RIF(5/8).fp_rank_diameter()
0
sage: RIF(5/7).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
```

Just because we have the best possible interval, doesn't mean the interval is actually small:

```
sage: a = RIF(pi)^12345678901234567890; a
[2.0985787164673874e323228496 .. +infinity]             # 32-bit
[5.8756537891115869e1388255822130839282 .. +infinity]   # 64-bit
sage: a.fp_rank_diameter()
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 $frac(x)$ 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 $floor(b) > \max(a, 0)$ then the interval obtained contains $[0, 1]$,
- if $ceil(a) < \min(b, 0)$ 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()
(0.000000000000000, 1.00000000000000)
sage: RIF(1, 1.5).frac().endpoints()
(0.000000000000000, 0.500000000000000)

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()
(0.000000000000000, 1.00000000000000)
```

```
sage: RIF(1.99, 2.00).frac().endpoints()
(0.000000000000000, 1.00000000000000)
sage: RIF(2.00, 2.025).frac().endpoints()
(0.000000000000000, 0.0250000000000000)

sage: RIF(-2.1,-0.9).frac().endpoints()
(-1.00000000000000, -0.000000000000000)
sage: RIF(-0.5,0.5).frac().endpoints()
(-0.500000000000000, 0.500000000000000)
```

gamma()

Return the gamma function evalutated on `self`.

EXAMPLES:

```
sage: RIF(1).gamma()
1
sage: RIF(5).gamma()
24
sage: a = RIF(3,4).gamma(); a
1.?e1
sage: a.lower(), a.upper()
(2.00000000000000, 6.00000000000000)
sage: RIF(-1/2).gamma()
-3.54490770181104?
sage: gamma(-1/2).n(100) in RIF(-1/2).gamma()
True
sage: 0 in (RealField(2000)(-19/3).gamma() - RealIntervalField(1000)(-19/3).gamma())
True
sage: gamma(RIF(100))
9.33262154439442?e155
sage: gamma(RIF(-10000/3))
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()
-1.60118039970055
```

Correctly detects poles:

```
sage: gamma(RIF(-3/2,-1/2))
[-infinity .. +infinity]
```

imag()

Return the imaginary part of this real interval.

(Since this is interval is real, this simply returns the zero interval.)

See also:

`real()`

EXAMPLES:

```
sage: RIF(2,3).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
```

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
```

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()
False
```

is_int()

Checks 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)
```

log(*base='e'*)

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

EXAMPLES:

```
sage: R = RealIntervalField()
sage: r = R(2); r.log()
0.6931471805599453?
sage: r = R(-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() / log(10.0)
1.204119982655925?

sage: r = RIF(39.9); r.log10()
1.600972895686749?

sage: r = RIF(0.0)
sage: r.log10()
[-infinity .. -infinity]

sage: r = RIF(-1.0)
sage: r.log10()
1.364376353841841?*I
```

log2()

Return `log` to the base 2 of `self`.

EXAMPLES:

```
sage: r = RIF(16.0)
sage: r.log2()
4

sage: r = RIF(31.9); r.log2()
4.995484518877507?

sage: r = RIF(0.0, 2.0)
sage: 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(truncate=False)
'3.1411'

sage: x = R(1.2,1.3); x.str(style='brackets')
'[1.1999 .. 1.3001]'
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
```

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()
2.00000000000000
sage: RIF(-1, 2).magnitude()
2.00000000000000
sage: parent(RIF(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, 1.00000000000000)
sage: RIF(-1, 1).max(RIF(2, 3)).endpoints()
(2.00000000000000, 3.00000000000000)
sage: RIF(-1, 1).max(RIF(-100, 100)).endpoints()
(-1.00000000000000, 100.000000000000)
sage: RIF(-1, 1).max(RIF(-100, 100), RIF(5, 10)).endpoints()
(5.00000000000000, 100.000000000000)
```

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
```

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
```

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()
(-1.00000000000000, 1.00000000000000)
```

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 ..]
```

See also:

`min()`

TESTS:

```
sage: a.max('x')
Traceback (most recent call last):
...
TypeError: unable to convert 'x' to real interval
```

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()
0.000000000000000
sage: RIF(-2, -1).mignitude()
1.00000000000000
sage: RIF(3, 4).mignitude()
3.00000000000000
sage: parent(RIF(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()
(-1.00000000000000, 1.00000000000000)
sage: RIF(-1, 1).min(RIF(-100, 100)).endpoints()
(-100.000000000000, 1.00000000000000)
sage: RIF(-1, 1).min(RIF(-100, 0)).endpoints()
(-100.000000000000, 0.000000000000000)
sage: RIF(-1, 1).min(RIF(-100, 2), RIF(-200, -3)).endpoints()
(-200.000000000000, -3.00000000000000)
```

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
```

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
```

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()
(-1.00000000000000, 1.00000000000000)
```

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 ..]
```

See also:

`max()`

TESTS:

```
sage: a.min('x')
Traceback (most recent call last):
...
TypeError: unable to convert 'x' to real interval
```

multiplicative_order()

Return n such that `self^n == 1`.

Only ± 1 have finite multiplicative order.

EXAMPLES:

```
sage: RIF(1).multiplicative_order()
1
sage: RIF(-1).multiplicative_order()
2
sage: RIF(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
```

prec()

Returns the precision of `self`.

EXAMPLES:

```
sage: RIF(2.1).precision()
53
sage: RealIntervalField(200)(2.1).precision()
200
```

precision()

Returns the precision of `self`.

EXAMPLES:

```
sage: RIF(2.1).precision()
53
sage: RealIntervalField(200)(2.1).precision()
200
```

psi()

Return the digamma function evaluated on self.

INPUT:

None.

OUTPUT:

A RealIntervalFieldElement.

EXAMPLES:

```
sage: psi_1 = RIF(1).psi()
sage: psi_1
-0.577215664901533?
sage: 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.)

EXAMPLES:

```
sage: RIF(1.2465).real() == RIF(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 RealNumber.

EXAMPLES:

```
sage: RIF(1, pi).relative_diameter()
1.03418797197910
```

round()

Return the nearest integer of this interval as an interval

EXAMPLES:

```
sage: RIF(7.2, 7.3).round()
7
sage: RIF(-3.2, -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
```

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()
3.00000000000000
sage: r.upper()
4.00000000000000
```

sec()

Return the secant of this number.

EXAMPLES:

```
sage: RealIntervalField(100)(2).sec()
-2.40299796172238098975460040142?
```

sech()

Return the hyperbolic secant of `self`.

EXAMPLES:

```
sage: RealIntervalField(100)(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()
22/7
sage: RealIntervalField(20)(pi).simplest_rational()
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
```

sin()

Return the sine of `self`.

EXAMPLES:

```
sage: R = RealIntervalField(100)
sage: R(2).sin()
0.909297426825681695396019865912?
```

sinh()

Return the hyperbolic sine of `self`.

EXAMPLES:

```
sage: q = RIF.pi()/12
sage: 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 nonpositive).

EXAMPLES:

```
sage: r = RIF(4.0)
sage: r.sqrt()
2
sage: r.sqrt()^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]'

sage: r = RIF(-2.0)
sage: r.sqrt()
Traceback (most recent call last):
...
ValueError: self (=-2) is not >= 0

sage: r = RIF(-2, 2)
sage: 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')
'[0.00000000000000000 .. 1.0000000000000000]'
sage: (RIF(-1, 1) * RIF(-1, 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
```

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()
'3.141592653589794?'
sage: RIF(pi, 22/7).str()
'3.142?'
sage: RIF(pi, 22/7).str(style='question')
'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'
sage: RIF(12^5).str(base=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'
```

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]'
```

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
```

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]'
```

We always use brackets style for `NaN` and infinities:

```
sage: RIF(pi, infinity)
[3.1415926535897931 .. +infinity]
sage: RIF(NaN)
[.. 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:

$$\text{mantissa} \cdot b^{\text{exponent}} - \text{error} \cdot b^k..\text{mantissa} \cdot b^{\text{exponent}} + \text{error} \cdot b^k$$

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, truncate=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)
[.. NaN ..]
sage: RIF(pi, infinity)
[3.1415926535897931 .. +infinity]
sage: RIF(-infinity, pi)
[-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)
1.?e-71
```

TESTS:

Check that trac ticket #13634 is fixed:

```
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?
```

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?
```

tanh()

Return the hyperbolic tangent of `self`.

EXAMPLES:

```
sage: q = RIF.pi()/11
sage: 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 non-negative 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
```

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()
3.00000000000000
sage: r.lower()
2.00000000000000
```

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')
'[1.0000000000000000 .. 3.1428571428571433]'
sage: RIF(1, 2).union(pi).str(style='brackets')
'[1.0000000000000000 .. 3.1415926535897936]'
sage: RIF(1).union(RIF(0, 2)).str(style='brackets')
'[0.00000000000000000 .. 2.0000000000000000]'
sage: RIF(1).union(RIF(-1)).str(style='brackets')
'[-1.0000000000000000 .. 1.0000000000000000]'
```

unique_ceil()

Returns the unique ceiling of this interval, if it is well defined, otherwise raises a `ValueError`.

OUTPUT:

- an integer.

See also:

`ceil()` – return the ceil as an interval (and never raise error)

EXAMPLES:

```
sage: RIF(pi).unique_ceil()
4
sage: RIF(100*pi).unique_ceil()
315
sage: RIF(100, 200).unique_ceil()
Traceback (most recent call last):
...
ValueError: interval does not have a unique ceil
```

unique_floor()

Returns the unique floor of this interval, if it is well defined, otherwise raises a `ValueError`.

OUTPUT:

- an integer.

See also:

`floor()` – return the floor as an interval (and never raise error)

EXAMPLES:

```
sage: RIF(pi).unique_floor()
3
sage: RIF(100*pi).unique_floor()
314
sage: RIF(100, 200).unique_floor()
Traceback (most recent call last):
...
ValueError: interval does not have a unique floor
```

unique_integer()

Return the unique integer in this interval, if there is exactly one, otherwise raises a `ValueError`.

EXAMPLES:

```
sage: RIF(pi).unique_integer()
Traceback (most recent call last):
...
ValueError: interval contains no integer
sage: RIF(pi, pi+1).unique_integer()
4
sage: RIF(pi, pi+2).unique_integer()
Traceback (most recent call last):
...
ValueError: interval contains more than one integer
sage: RIF(100).unique_integer()
100
```

unique_round()

Returns the unique round (nearest integer) of this interval, if it is well defined, otherwise raises a `ValueError`.

OUTPUT:

- an integer.

See also:

`round()` – return the round as an interval (and never raise error)

EXAMPLES:

```
sage: RIF(pi).unique_round()
3
sage: RIF(1000*pi).unique_round()
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()
-3
sage: (RIF(4.5).unique_round(), RIF(-4.5).unique_round())
(5, -5)
```

TESTS:

```
sage: RIF(-1/2, -1/3).unique_round()
Traceback (most recent call last):
...
ValueError: interval does not have a unique round (nearest integer)
sage: RIF(-1/2, 1/3).unique_round()
Traceback (most recent call last):
...
ValueError: interval does not have a unique round (nearest integer)
sage: RIF(-1/3, 1/3).unique_round()
0
sage: RIF(-1/2, 0).unique_round()
Traceback (most recent call last):
...
ValueError: interval does not have a unique round (nearest integer)
sage: RIF(1/2).unique_round()
1
sage: RIF(-1/2).unique_round()
-1
sage: RIF(0).unique_round()
0
```

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
```

unique_trunc()

Return the nearest integer toward zero if it is unique, otherwise raise a `ValueError`.

EXAMPLES:

```
sage: RIF(1.3,1.4).unique_trunc()
1
sage: RIF(-3.3, -3.2).unique_trunc()
-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)
```

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(truncate=False)
'3.1417'

sage: R = RealIntervalField(13)
sage: x = R(1.2,1.3); x.str(style='brackets')
'[1.1999 .. 1.3001]'
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
```

class sage.rings.real_mpfi.**RealIntervalField_class**

Bases: sage.rings.ring.Field

Class of the real interval field.

INPUT:

- prec – (integer) precision; default = 53 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 – (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
```

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)
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?
```

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]'
```

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')
'[0.33333333333333331 .. 0.66666666666666675]'
```

The extra parentheses aren't needed:

```
sage: RIF(1/3,2/3).str(style='brackets')
'[0.33333333333333331 .. 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')
'[2.7182818284590455 .. 3.1415926535897932]'
```

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))
(3.141592653589794?, False)
sage: check(RIF(RR(pi)))
(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)
```

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')
'[0.099999999999999991 .. 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
```

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)
3.14159265358979323846264338328?
sage: R(-2/19)
-0.1052631578947368421052631578948?
sage: R(e,pi).str(style='brackets')
'[2.7182818284590452353602874713512 .. 3.1415926535897932384626433832825]'
```

TESTS:

```
sage: RIF._lower_field() is RealField(53, rnd='RNDD')
True
sage: RIF._upper_field() is RealField(53, rnd='RNDU')
True
sage: RIF._middle_field() is RR
True
sage: TestSuite(RIF).run()
```

characteristic()

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

EXAMPLES:

```
sage: RealIntervalField(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
```

construction()

Returns 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
```

euler_constant()

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

EXAMPLES:

```
sage: RealIntervalField(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
```

gens()

Return a list of generators.

EXAMPLE:

```
sage: RIF.gens()
[1]
```

is_exact()

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

EXAMPLES:

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

is_finite()

Return False, since the field of real numbers is not finite.

EXAMPLES:

```
sage: RealIntervalField(10).is_finite()
False
```

log2()

Returns $\log(2)$ to the precision of this field.

EXAMPLES:

```
sage: R=RealIntervalField(100)
sage: R.log2()
0.693147180559945309417232121458?
sage: R(2).log()
0.693147180559945309417232121458?
```

name()

Return the name of self.

EXAMPLES:

```
sage: RIF.name()
'IntervalRealIntervalField53'
sage: RealIntervalField(200).name()
'IntervalRealIntervalField200'
```

ngens()

Return the number of generators of `self`, which is 1.

EXAMPLES:

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

pi()

Returns 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?
```

prec()

Return the precision of this field (in bits).

EXAMPLES:

```
sage: RIF.precision()
53
sage: RealIntervalField(200).precision()
200
```

precision()

Return the precision of this field (in bits).

EXAMPLES:

```
sage: RIF.precision()
53
sage: RealIntervalField(200).precision()
200
```

random_element(**args*, ***kwds*)

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()
0.15363619378561300?
sage: RIF.random_element()
-0.50298737524751780?
sage: RIF.random_element(-100, 100)
60.958996432224126?
```

Passes extra positional or keyword arguments through:

```
sage: RIF.random_element(min=0, max=100)
2.5572702830891970?
sage: RIF.random_element(min=-100, max=0)
-1.5803457307118123?
```

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?
```

to_prec(*prec*)

Returns 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
```

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)
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)
True
sage: sage.rings.real_mpfi.is_RealIntervalField(RealIntervalField(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))
True
sage: sage.rings.real_mpfi.is_RealIntervalFieldElement(RealIntervalField(200)(2.2))
True
```

2.2 Field of Arbitrary Precision Real Number Intervals

sage.rings.real_interval_field.**is_RealIntervalField**(*x*)

Check if x is a RealIntervalField_class.

EXAMPLES:

```
sage: from sage.rings.real_interval_field import is_RealIntervalField as is_RIF
sage: is_RIF(RIF)
True
```

sage.rings.real_interval_field.**is_RealIntervalFieldElement**(*x*)

Check if x is a RealIntervalFieldElement.

EXAMPLES:

```
sage: from sage.rings.real_interval_field import is_RealIntervalFieldElement as is_RIFE
sage: is_RIFE(RIF(2.5))
True
```

2.3 Real intervals with a fixed absolute precision

class sage.rings.real_interval_absolute.**Factory**

Bases: sage.structure.factory.UniqueFactory

create_key(*prec*)

The only piece of data is the precision.

TESTS:

```
sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: RealIntervalAbsoluteField.create_key(1000)
1000
```

create_object(*version, prec*)

Ensures uniqueness.

TESTS:

```
sage: from sage.rings.real_interval_absolute import RealIntervalAbsoluteField
sage: RealIntervalAbsoluteField(23) is RealIntervalAbsoluteField(23) # indirect doctest
True
```

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?
```

class sage.rings.real_interval_absolute.**RealIntervalAbsoluteElement**

Bases: sage.structure.element.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)
0.333333333333334?
sage: R(1.3)
1.300000000000000?
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.?
```

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()
0.333333333333333333333333333334?
sage: R(-1/3).abs()
0.333333333333333333333333333334?
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)
```

absolute_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
```

contains_zero()

Return whether self contains zero.

EXAMPLES:

```
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
```

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
```

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)
```

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
```

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
```

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
```

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
```

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
```

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)
```

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
```

sage.rings.real_interval_absolute.**RealIntervalAbsoluteField**(**args*, ***kwds*)

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
```

class sage.rings.real_interval_absolute.**RealIntervalAbsoluteField_class**

Bases: `sage.rings.ring.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
```

absprec()

Returns 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
```

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
```

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
```

2.4 Field of Arbitrary Precision Complex Intervals

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): trac ticket #3893: `random_element()` should pass on `*args` and `**kwds`.
- 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 that 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?
```

class `sage.rings.complex_interval_field.`**`ComplexIntervalField_class`**(*prec=53*)

Bases: `sage.rings.ring.Field`

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
```

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 = PolynomialRing(Q, 'x')
sage: C(S.gen())
Traceback (most recent call last):
...
TypeError: unable to coerce to a ComplexIntervalFieldElement
```

This illustrates precision:

```
sage: CIF = ComplexIntervalField(10); CIF(1/3, 2/3)
0.334? + 0.667?*I
sage: CIF
Complex Interval Field with 10 bits of precision
sage: CIF = ComplexIntervalField(100); CIF
Complex Interval Field with 100 bits of precision
sage: z = CIF(1/3, 2/3); z
0.333333333333333333333333333334? + 0.666666666666666666666666666667?*I
```

We can load and save complex numbers and the complex interval field:

```
sage: cmp(loads(z.dumps()), z)
0
sage: loads(CIF.dumps()) == CIF
True
sage: k = ComplexIntervalField(100)
sage: loads(dumps(k)) == k
True
```

This illustrates basic properties of a complex (interval) field:

```
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
sage: CIF == ComplexIntervalField(53)
False
sage: CIF == 1.1
False
sage: CIF = ComplexIntervalField(53)

sage: CIF.category()
Category of fields
sage: TestSuite(CIF).run()
```

TESTS:

This checks that trac ticket #15355 is fixed:

```
sage: x + CIF(RIF(-2,2), 0)
x + 0.?e1
sage: x + CIF(RIF(-2,2), RIF(-2,2))
x + 0.?e1 + 0.?e1*I
sage: x + RIF(-2,2)
x + 0.?e1
sage: x + CIF(RIF(3.14,3.15), RIF(3.14, 3.15))
x + 3.15? + 3.15?*I
sage: CIF(RIF(-2,2), RIF(-2,2))
0.?e1 + 0.?e1*I
sage: x + CIF(RIF(3.14,3.15), 0)
x + 3.15?
```

characteristic()

Return the characteristic of the complex (interval) field, which is 0.

EXAMPLES:

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

gen(*n=0*)

Return the generator of the complex (interval) field.

EXAMPLES:

```
sage: CIF.0
1*I
sage: CIF.gen(0)
1*I
```

is_exact()

The complex interval field is not exact.

EXAMPLES:

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

is_field(*proof=True*)

Return `True`, since the complex numbers are a field.

EXAMPLES:

```
sage: CIF.is_field()
True
```

is_finite()

Return `False`, since the complex numbers are infinite.

EXAMPLES:

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

ngens()

The number of generators of this complex (interval) field as an $\mathbf{R}$-algebra.

There is one generator, namely `sqrt(-1)`.

EXAMPLES:

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

pi()

Returns π as an element in the complex (interval) field.

EXAMPLES:

```
sage: ComplexIntervalField(100).pi()
3.14159265358979323846264338328?
```

prec()

Returns the precision of `self` (in bits).

EXAMPLES:

```
sage: CIF.prec()
53
sage: ComplexIntervalField(200).prec()
200
```

precision()

Returns the precision of `self` (in bits).

EXAMPLES:

```
sage: CIF.prec()
53
sage: ComplexIntervalField(200).prec()
200
```

random_element (**args*, ***kwds*)

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()
0.15363619378561300? - 0.50298737524751780?*I
sage: CIF.random_element(10, 20)
18.047949821611205? + 10.255727028308920?*I
```

Passes extra positional or keyword arguments through:

```
sage: CIF.random_element(max=0, min=-5)
-0.079017286535590259? - 2.8712089896087117?*I
```

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
```

to_prec (*prec*)

Returns 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
```

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 – an 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
```

sage.rings.complex_interval_field.**is_ComplexIntervalField**(*x*)

Check if x is a ComplexIntervalField.

EXAMPLES:

```
sage: from sage.rings.complex_interval_field import is_ComplexIntervalField as is_CIF
sage: is_CIF(CIF)
True
sage: is_CIF(CC)
False
```

sage.rings.complex_interval_field.**late_import**()

Import the objects/modules after build (when needed).

TESTS:

```
sage: sage.rings.complex_interval_field.late_import()
```

2.5 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_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.

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: sage.structure.element.FieldElement

A complex interval.

EXAMPLES:

```
sage: I = CIF.gen()
sage: b = 1.5 + 2.5*I
sage: TestSuite(b).run()
```

arg()

Same as `argument()`.

EXAMPLES:

```
sage: i = CIF.0
sage: (i^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: (RR('-0.001') - i).argument()
-1.571796326461564?
sage: CIF(2).argument()
0
sage: CIF(-2).argument()
3.141592653589794?
```

Here we see that if the interval crosses the negative real axis, then the argument can exceed π, and we 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?
```

bisection()

Returns 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(), z.imag().endpoints()
(2.00000000000000, 2.50000000000000) (-5.00000000000000, -4.50000000000000)
```

```
(2.50000000000000, 3.00000000000000) (-5.00000000000000, -4.50000000000000)
(2.00000000000000, 2.50000000000000) (-4.50000000000000, -4.00000000000000)
(2.50000000000000, 3.00000000000000) (-4.50000000000000, -4.00000000000000)

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
```

center()

Returns the closest floating-point approximation to the center of the interval.

EXAMPLES:

```
sage: CIF(RIF(1, 2), RIF(3, 4)).center()
1.50000000000000 + 3.50000000000000*I
```

conjugate()

Return the complex conjugate of this complex number.

EXAMPLES:

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

contains_zero()

Returns `True` if `self` is an interval containing zero.

EXAMPLES:

```
sage: CIF(0).contains_zero()
True
sage: CIF(RIF(-1, 1), 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?
```

Check that trac ticket #17285 is fixed:

```
sage: CIF(cos(2/3))
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?
```

ALGORITHM:

The implementation uses the following trigonometric identity

$$\cosh(x + iy) = \cos(y)\cosh(x) + i\sin(y)\sinh(x)$$

crosses_log_branch_cut()

Returns `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, 2.50000000000000001); 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
```

diameter()

Returns 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()
2.00000000000000
sage: CIF(RIF(-0.1, 0.1), RIF(13, 17)).diameter()
0.266666666666667
sage: CIF(RIF(-1, 1), 15).diameter()
2.00000000000000
```

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)
```

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()
(1.00000000000000 + 3.00000000000000*I,
 2.00000000000000 + 4.00000000000000*I,
 1.00000000000000 + 4.00000000000000*I,
 2.00000000000000 + 3.00000000000000*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)
```

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()
1.468693939915885157138967597326604261326956736629008722797675676310936965859512138722724502?
```

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
```

intersection(*other*)

Returns 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
```

is_exact()

Returns 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()
False
sage: CIF(e).is_exact()
False
sage: CIF(1e100).is_exact()
True
sage: (CIF(1e100) + 1).is_exact()
False
```

is_square()

This function always returns `True` as **C** is algebraically closed.

EXAMPLES:

```
sage: CIF(2, 1).is_square()
True
```

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
```

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
```

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
```

But we allow the exact input zero:

```
sage: CIF(0).log()
[-infinity .. -infinity]
```

If a base is passed from another function, we can accommodate this:

```
sage: CIF(-1,1).log(2)
0.500000000000000? + 3.399270106370396?*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
```

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()
0.000000000000000
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
```

norm()

Returns 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:

$$extnorm(c) = extnorm(a + bi) = c \cdot \overline{c} = a^2 + b^2.$$

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
```

overlaps(*other*)

Returns `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
```

plot(*pointsize=10*, ***kwds*)

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])
Graphics object consisting of 20 graphics primitives
```

Exact and nearly exact points are still visible:

```
sage: plot(CIF(pi, 1), color='red') + plot(CIF(1, e), color='purple') + plot(CIF(-1, -1))
Graphics object consisting of 6 graphics primitives
```

A demonstration that $z \mapsto z^2$ acts chaotically on $|z| = 1$:

```
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
```

prec()

Return precision of this complex number.

EXAMPLES:

```
sage: i = ComplexIntervalField(2000).0
sage: 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
```

sin()

Compute the sine of this complex interval.

EXAMPLES:

```
sage: CIF(1,1).sin()
1.298457581415978? + 0.634963914784736?*I
sage: CIF(2).sin()
0.909297426825682?
sage: CIF(0,2).sin()
3.626860407847019?*I
```

Check that trac ticket #17825 is fixed:

```
sage: CIF(sin(2/3))
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
```

ALGORITHM:

The implementation uses the following trigonometric identity

$$\sinh(x + iy) = \cos(y)\sinh(x) + i\sin(y)\cosh(x)$$

sqrt(*all=False*, ***kwds*)

The square root function.

> **Warning:** We approximate the standard branch cut along the negative real axis, with `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` – bool (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
2.00000000000000? - 1.00000000000000?*I
```

```
sage: CC(-2-I).sqrt()^2
-2.00000000000000 - 1.00000000000000*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]*
sage: CIF(-5, -1).sqrt()
0.222507880301783? - 2.247111425095870?*I
```

str (*base=10*, *style=None*)

Returns a string representation of `self`.

EXAMPLES:

```
sage: CIF(1.5).str()
'1.5000000000000000?'
sage: CIF(1.5, 2.5).str()
'1.5000000000000000? + 2.5000000000000000?*I'
sage: CIF(1.5, -2.5).str()
'1.5000000000000000? - 2.5000000000000000?*I'
sage: CIF(0, -2.5).str()
'-2.5000000000000000?*I'
sage: CIF(1.5).str(base=3)
'1.1111111111111111111111111111111112?'
sage: CIF(1, pi).str(style='brackets')
'[1.0000000000000000 .. 1.0000000000000000] + [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()
-2.18503986326152?
sage: CIF(0,2).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()
-2.18503986326152?*I
```

union (*other*)

Returns the smallest complex interval including the two complex intervals `self` and `other`.

EXAMPLES:

```
sage: CIF(0).union(CIF(5, 5)).str(style='brackets')
'[0.00000000000000000 .. 5.0000000000000000] + [0.00000000000000000 .. 5.0000000000000000]*I
```

sage.rings.complex_interval.**create_ComplexIntervalFieldElement**(*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 `ComplexIntervalField(prec=n)`, where n potentially has slightly more (controlled by pad) bits than given by s.

INPUT:

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

EXAMPLES:

```
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(1.000000000000000000000000000,2)
1 + 2*I
sage: ComplexIntervalFieldElement(1,2.000000000000000000000)
1 + 2*I
sage: ComplexIntervalFieldElement(1.234567890123456789012345, 5.4321098654321987654321)
1.234567890123456789012350? + 5.432109865432198765432000?*I
```

TESTS:

Make sure we've rounded up `log(10,2)` enough to guarantee sufficient precision (trac ticket #10164). This is a little tricky because at the time of writing, we don't support intervals long enough to trip the error. However, at least we can make sure that we either do it correctly or fail noisily:

```
sage: c_CIFE = sage.rings.complex_interval.create_ComplexIntervalFieldElement
sage: for kp in range(2,6):
...       s = '1.' + '0'*10**kp + '1'
...       try:
...           assert c_CIFE(s,0).real()-1 != 0
...           assert c_CIFE(0,s).imag()-1 != 0
...       except TypeError:
...           pass
```

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))
True
```

```
sage: is_CIFE(CC(2))
False
```

sage.rings.complex_interval.**make_ComplexIntervalFieldElement0**(*fld*, *re*, *im*)
Construct a `ComplexIntervalFieldElement` for pickling.

TESTS:

```
sage: a = CIF(1 + I)
sage: loads(dumps(a)) == a # indirect doctest
True
```

2.6 Arbitrary precision real balls using Arb

This is a binding to the Arb library for ball arithmetic. 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 own documentation, licenced under the GNU General Public License version 2, or later.

See also:

- `Complex balls using Arb`
- `Real intervals using MPFI`

2.6.1 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 Arb 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)
[3.141592653589793 +/- 5.61e-16]
sage: RBF(pi).mid(), RBF(pi).rad()
(3.14159265358979, 4.4408921e-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 +/- 2.98e+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 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
```

2.6.2 Comparison

Warning: In accordance with the semantics of 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 the same object or if both are exact and equal:

```
sage: a = RBF(1)
sage: b = RBF(1)
sage: a is b
False
sage: 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 == b
False
```

A ball is non-zero 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
```

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
```

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
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
```

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
Traceback (most recent call last):
...
ValueError: infinite but not with +/- phase
sage: 1/RBF(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)
False
sage: 1/RBF(0) <= RBF(infinity)
True
```

TESTS:

```
sage: (RBF(pi) * identity_matrix(QQ, 3)).parent()
Full MatrixSpace of 3 by 3 dense matrices over Real ball field with 53 bits precision

sage: polygen(RBF, x)^3
x^3
```

2.6.3 Classes and Methods

class `sage.rings.real_arb.`**`RealBall`**

Bases: `sage.structure.element.RingElement`

Hold one `arb_t` of the Arb library

EXAMPLES:

```
sage: a = RealBallField()(RIF(1))                     # indirect doctest
sage: b = a.psi()
sage: b
[-0.577215664901533 +/- 3.85e-16]
sage: RIF(b)
-0.577215664901533?
```

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 +/- 3.99e-3]
sage: b.is_exact()
True
sage: 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()
51
sage: RBF(1).accuracy() == RBF.maximal_accuracy()
True
sage: RBF(NaN).accuracy() == -RBF.maximal_accuracy()
True
```

See also:

`maximal_accuracy()`

add_error(*ampl*)

Increase the radius of this ball by (an upper bound on) `ampl`.

If `ampl` is negative, the radius is unchanged.

INPUT:

- `ampl` – 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)
[1.000000000000000 +/- 1.01e-16]
```

TESTS:

```
sage: RBF(1).add_error(-1)
1.000000000000000
sage: RBF(0).add_error(RBF(1, rad=2.)).endpoints()
(-3.00000000745059, 3.00000000745059)
```

agm(*other*)

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

EXAMPLES:

```
sage: RBF(1).agm(1)
1.000000000000000
sage: RBF(sqrt(2)).agm(1)^(-1)
[0.83462684167407 +/- 4.31e-15]
```

arccos()

Return the arccosine of this ball.

EXAMPLES:

```
sage: RBF(1).arccos()
0
sage: RBF(1, rad=.125r).arccos()
nan
```

arccosh()

Return the inverse hyperbolic cosine of this ball.

EXAMPLES:

```
sage: RBF(2).arccosh()
[1.316957896924817 +/- 6.61e-16]
sage: RBF(1).arccosh()
0
sage: RBF(0).arccosh()
nan
```

arcsin()

Return the arcsine of this ball.

EXAMPLES:

```
sage: RBF(1).arcsin()
[1.570796326794897 +/- 6.65e-16]
sage: RBF(1, rad=.125r).arcsin()
nan
```

arcsinh()

Return the inverse hyperbolic sine of this ball.

EXAMPLES:

```
sage: RBF(1).arcsinh()
[0.881373587019543 +/- 1.87e-16]
sage: RBF(0).arcsinh()
0
```

arctan()

Return the arctangent of this ball.

EXAMPLES:

```
sage: RBF(1).arctan()
[0.785398163397448 +/- 3.91e-16]
```

arctanh()

Return the inverse hyperbolic tangent of this ball.

EXAMPLES:

```
sage: RBF(0).arctanh()
0
sage: RBF(1/2).arctanh()
[0.549306144334055 +/- 3.32e-16]
sage: RBF(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 +/- 7.82e-5]
sage: b = RealBallField(8)(1/3).below_abs()
sage: b
[0.33 +/- 7.82e-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
```

See also:

`above_abs()`

ceil()

Return the ceil of this ball.

EXAMPLES:

```
sage: RBF(1000+1/3, 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

sage: b = RBF(2)^(2^1000)
sage: b.mid()
Traceback (most recent call last):
...
RuntimeError: unable to convert to MPFR (exponent out of range?)
```

See also:

rad(), squash()

chebyshev_T(*n*)

Evaluate the Chebyshev polynomial of the first kind T_n at this ball.

EXAMPLES:

```
sage: RBF(pi).chebyshev_T(0)
1.000000000000000
sage: RBF(pi).chebyshev_T(1) # abs tol 1e-16
[3.141592653589793 +/- 5.62e-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
```

chebyshev_U(*n*)

Evaluate the Chebyshev polynomial of the second kind U_n at this ball.

EXAMPLES:

```
sage: RBF(pi).chebyshev_U(0)
1.000000000000000
sage: RBF(pi).chebyshev_U(1)
[6.28318530717959 +/- 4.66e-15]
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
```

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

sage: RBF(1/3).contains_exact(1/3)
True
sage: RBF(sqrt(2)).contains_exact(sqrt(2))
Traceback (most recent call last):
...
TypeError: unsupported type: <type 'sage.symbolic.expression.Expression'>
```

TESTS:

```
sage: b.contains_exact(1r)
True
```

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
```

cos()

Return the cosine of this ball.

EXAMPLES:

```
sage: RBF(pi).cos() # abs tol 1e-16
[-1.00000000000000 +/- 6.69e-16]
```

See also:

`cospi()`

cosh()

Return the hyperbolic cosine of this ball.

EXAMPLES:

```
sage: RBF(1).cosh()
[1.543080634815244 +/- 5.28e-16]
```

cot()

Return the cotangent of this ball.

EXAMPLES:

```
sage: RBF(1).cot()
[0.642092615934331 +/- 4.79e-16]
sage: RBF(pi).cot()
[+/- inf]
```

coth()

Return the hyperbolic cotangent of this ball.

EXAMPLES:

```
sage: RBF(1).coth()
[1.313035285499331 +/- 4.97e-16]
sage: RBF(0).coth()
[+/- inf]
```

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()
(-0.333333333333334, -0.333333333333333)
```

See also:

`lower()`, `upper()`

exp()

Return the exponential of this ball.

EXAMPLES:

```
sage: RBF(1).exp()
[2.718281828459045 +/- 5.41e-16]
```

expm1()

Return `exp(self) - 1`, computed accurately when `self` is close to zero.

EXAMPLES:

```
sage: eps = RBF(1e-30)
sage: exp(eps) - 1
[+/- 3.16e-30]
sage: eps.expm1()
[1.000000000000000e-30 +/- 8.34e-47]
```

floor()

Return the floor of this ball.

EXAMPLES:

```
sage: RBF(1000+1/3, rad=1.r).floor()
[1.00e+3 +/- 1.01]
```

gamma()

Return the image of this ball by the Euler Gamma function.

For integer and rational arguments, `gamma()` may be faster.

EXAMPLES:

```
sage: RBF(1/2).gamma()
[1.772453850905516 +/- 3.41e-16]
```

See also:

`gamma()`

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
```

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
```

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
```

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()
True
sage: (~RBF(0)).is_infinity()
True
sage: RBF(42, rad=1.r).is_infinity()
False
```

is_negative_infinity()

Return True if this ball is the point $-\infty$.

EXAMPLES:

```
sage: 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()
True
sage: RBF(RIF(-0.5, 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
```

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
```

See also:

`is_nonzero()`

log()

Return the natural logarithm of this ball.

EXAMPLES:

```
sage: RBF(3).log()
[1.098612288668110 +/- 6.63e-16]
sage: RBF(-1/3).log()
nan
```

log1p()

Return `log(1 + self)`, computed accurately when `self` is close to zero.

EXAMPLES:

```
sage: eps = RBF(1e-30)
sage: (1 + eps).log()
[+/- 2.23e-16]
sage: eps.log1p()
[1.00000000000000e-30 +/- 2.68e-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 +/- 2.67e-16]
```

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()
-0.333333333333334
sage: RBF(-1/3).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()
```

```
(-1.00000000465662, 2.00000000651926)

sage: RBF(-infinity).max(-3, 1/3)
[0.3333333333333333 +/- 7.04e-17]
```

Note that calls involving NaNs try to return a number when possible. This is consistent with IEEE-754-2008 but may be surprising.

```
sage: RBF('nan').max(0)
0
sage: RBF('nan').max(RBF('nan'))
nan
```

See also:

`min()`

TESTS:

```
sage: RBF(0).max()
0
```

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

sage: b = RBF(2)^(2^1000)
sage: b.mid()
Traceback (most recent call last):
...
RuntimeError: unable to convert to MPFR (exponent out of range?)
```

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()
(-2.00000000651926, 1.00000000465662)

sage: RBF(infinity).min(3, 1/3)
[0.3333333333333333 +/- 7.04e-17]
```

Note that calls involving NaNs try to return a number when possible. This is consistent with IEEE-754-2008 but may be surprising.

```
sage: RBF('nan').min(0)
0
sage: RBF('nan').min(RBF('nan'))
nan
```

See also:

`max()`

TESTS:

```
sage: RBF(0).min()
0
sage: RBF(infinity).min().rad()
0.00000000
```

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))
True
sage: RBF(pi).overlaps(RBF(3))
False
```

polylog (*s*)

Return the polylogarithm $\mathrm{Li}_s(\mathrm{self})$.

EXAMPLES:

```
sage: polylog(0, -1)
-1/2
sage: RBF(-1).polylog(0)
[-0.50000000000000 +/- 1.29e-15]
sage: polylog(1, 1/2)
-log(1/2)
sage: RBF(1/2).polylog(1)
[0.6931471805599 +/- 5.02e-14]
sage: RBF(1/3).polylog(1/2)
[0.44210883528067 +/- 6.75e-15]
sage: RBF(1/3).polylog(RLF(pi))
[0.34728895057225 +/- 5.51e-15]
```

TESTS:

```
sage: RBF(1/3).polylog(2r)
[0.36621322997706 +/- 4.62e-15]
```

psi ()

Compute the digamma function with argument self.

EXAMPLES:

```
sage: RBF(1).psi()
[-0.577215664901533 +/- 3.85e-16]
```

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
```

See also:

mid(), rad_as_ball()

TESTS:

```
sage: (RBF(1, rad=.1) << (2^64)).rad()
Traceback (most recent call last):
...
RuntimeError: unable to convert the radius to MPFR (exponent out of range?)
```

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 +/- 3.13e-26]
sage: rad.is_exact()
True
sage: rad.parent()
Real ball field with 30 bits precision
```

See also:

squash(), rad()

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
```

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 that their parent's nominal precision (see real_arb for more information):

```
sage: b = RBF(pi.n(100))
sage: b.mid()
3.141592653589793238462643383
```

The `round()` method rounds such a ball to its parent's precision:

```
sage: b.round().mid()
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 +/- 5.73e-16]
sage: RBF(0).rsqrt()
nan
```

sin()

Return the sine of this ball.

EXAMPLES:

```
sage: RBF(pi).sin() # abs tol 1e-16
[+/- 5.69e-16]
```

See also:

`sinpi()`

sinh()

Return the hyperbolic sine of this ball.

EXAMPLES:

```
sage: RBF(1).sinh()
[1.175201193643801 +/- 6.18e-16]
```

sqrt()

Return the square root of this ball.

EXAMPLES:

```
sage: RBF(2).sqrt()
[1.414213562373095 +/- 2.99e-16]
sage: RBF(-1/3).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
[+/- 1.12e-16]
sage: eps.sqrt1pm1()
[5.00000000000000e-21 +/- 2.54e-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 +/- 2.99e-16]
sage: RBF(-1/3).sqrtpos()
0
sage: RBF(0, rad=2.r).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 +/- 2.83e-5]
sage: mid.is_exact()
True
sage: mid.parent()
Real ball field with 16 bits precision
```

See also:

`mid()`, `rad_as_ball()`

tan()

Return the tangent of this ball.

EXAMPLES:

```
sage: RBF(1).tan()
[1.557407724654902 +/- 3.26e-16]
sage: RBF(pi/2).tan()
[+/- inf]
```

tanh()

Return the hyperbolic tangent of this ball.

EXAMPLES:

```
sage: RBF(1).tanh()
[0.761594155955765 +/- 2.81e-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(0.11111111111111, rad=.001)
sage: b.mid()
0.111111111111110
sage: 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()
(0.000000000000000, 1.00000000000000)
```

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
```

See also:

lower(), endpoints()

zeta (*a=None*)

Return the image of this ball by the Hurwitz zeta function.

For a = 1 (or a = None), this computes the Riemann 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()
[-0.0833333333333333 +/- 4.36e-17]
sage: RBF(-1).zeta(1)
[-0.0833333333333333 +/- 6.81e-17]
sage: RBF(-1).zeta(2) # abs tol 1e-16
[-1.083333333333333 +/- 4.09e-16]
```

class sage.rings.real_arb.**RealBallField** (*precision*, *category*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.rings.ring.Field

An approximation of the field of real numbers using mid-rad intervals, also known as balls.

INPUT:

- precision – an integer ≥ 2.

EXAMPLES:

```
sage: RBF = RealBallField() # indirect doctest
sage: RBF(1)
1.000000000000000
```

```
sage: (1/2*RBF(1)) + AA(sqrt(2)) - 1 + polygen(QQ, x)
x + [0.914213562373095 +/- 4.10e-16]
```

TESTS:

```
sage: RBF.bracket(RBF(1/2), RBF(1/3))
[+/- 5.56e-17]
sage: RBF.cardinality()
+Infinity
sage: RBF.cartesian_product(QQ).an_element()**2
([1.440000000000000 +/- 4.98e-16], 1/4)
sage: RBF.coerce_embedding() is None
True
sage: loads(dumps(RBF)) is RBF
True
sage: RBF['x'].gens_dict_recursive()
{'x': x}
sage: RBF.is_finite()
False
sage: RBF.is_zero()
False
sage: RBF.one()
1.000000000000000
sage: RBF.zero()
0
```

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 precision
sage: RealBallField(3).algebraic_closure()
Complex ball field with 3 bits precision
```

bernoulli(*n*)

Return a ball enclosing the n-th Bernoulli number.

EXAMPLES:

```
sage: [RBF.bernoulli(n) for n in range(4)]
[1.000000000000000, -0.5000000000000000, [0.1666666666666667 +/- 7.04e-17], 0]
sage: RBF.bernoulli(2**20)
[-1.823002872104961e+5020717 +/- 7.16e+5020701]
sage: RBF.bernoulli(2**1000)
Traceback (most recent call last):
...
ValueError: argument too large
```

TESTS:

```
sage: RBF.bernoulli(2r)
[0.1666666666666667 +/- 7.04e-17]
sage: RBF.bernoulli(2/3)
Traceback (most recent call last):
```

```
...
TypeError: no canonical coercion from Rational Field to Integer Ring
sage: RBF.bernoulli(-1)
Traceback (most recent call last):
...
ValueError: expected a nonnegative index
```

characteristic()

Real ball fields have characteristic zero.

EXAMPLES:

```
sage: 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 precision
sage: RealBallField(3).algebraic_closure()
Complex ball field with 3 bits 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
(Completion[+Infinity], Rational Field)
sage: functor(base) is RBF
True
```

cospi(*x*)

Return a ball enclosing cos(πx).

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)
-1.000000000000000
sage: RBF.cospi(1/3)
0.5000000000000000
```

See also:

`cos()`

TESTS:

```
sage: RBF.cospi(RLF(sqrt(2)))
[-0.26625534204142 +/- 5.38e-15]
```

fibonacci(*n*)

Return a ball enclosing the `n`-th Fibonacci number.

EXAMPLES:

```
sage: [RBF.fibonacci(n) for n in xrange(7)]
[0,
1.000000000000000,
1.000000000000000,
2.000000000000000,
3.000000000000000,
5.000000000000000,
8.000000000000000]
sage: RBF.fibonacci(-2)
-1.000000000000000
sage: RBF.fibonacci(10**20)
[3.78202087472056e+20898764024997873376 +/- 4.01e+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)
24.00000000000000
sage: RBF.gamma(10**20)
[+/- 5.92e+1956570551809674821757]
sage: RBF.gamma(1/3)
[2.678938534707747 +/- 8.99e-16]
sage: RBF.gamma(-5)
nan
```

See also:

gamma()

TESTS:

```
sage: RBF.gamma(RLF(pi))
[2.2880377953400 +/- 4.29e-14]
```

gens ()

EXAMPLE:

```
sage: RBF.gens()
(1.000000000000000,)
sage: RBF.gens_dict()
{'1.000000000000000': 1.000000000000000}
```

is_exact ()

Real ball fields are not exact.

EXAMPLES:

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

is_finite ()

Real ball fields are infinite.

They already specify it via their category, but we currently need to re-implement this method due to the legacy implementation in `sage.rings.ring.Ring`.

EXAMPLES:

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

maximal_accuracy()
Return the relative accuracy of exact elements measured in bits.

OUTPUT:

An integer.

EXAMPLES:

```
sage: RBF.maximal_accuracy()
9223372036854775807 # 64-bit
2147483647          # 32-bit
```

See also:

`RealBall.accuracy()`

precision()
Return the bit precision used for operations on elements of this field.

EXAMPLES:

```
sage: RealBallField().precision()
53
```

sinpi(*x*)
Return a ball enclosing sin(πx).

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 +/- 5.15e-16]
sage: RBF.sinpi(1 + 2^(-100))
[-2.478279624546525e-30 +/- 5.90e-46]
```

See also:

`sin()`

TESTS:

```
sage: RBF.sinpi(RLF(sqrt(2)))
[-0.96390253284988 +/- 4.11e-15]
```

some_elements()
Real ball fields contain exact balls, inexact balls, infinities, and more.

EXAMPLES:

```
sage: RBF.some_elements()
[1.000000000000000,
[0.3333333333333333 +/- 7.04e-17],
[-4.733045976388941e+363922934236666733021124 +/- 3.46e+363922934236666733021108],
[+/- 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) # abs tol 5e-16
[1.202056903159594 +/- 2.87e-16]
sage: RBF.zeta(1)
nan
sage: RBF.zeta(1/2)
[-1.460354508809587 +/- 1.94e-16]
```

See also:

`zeta()`

2.7 Arbitrary precision complex balls using Arb

This is a binding to the Arb library; 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 own documentation, licenced under the GNU General Public License version 2, or later.

See also:

- `Real balls using Arb`
- `Complex interval field (using MPFI)`
- `Complex intervals (using MPFI)`

2.7.1 Data Structure

A `ComplexBall` represents a complex number with error bounds. It wraps an Arb 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 in Arb 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.

2.7.2 Comparison

Warning: In accordance with the semantics of 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 the same object or if both are exact and equal:

```
sage: a = CBF(1, 2)
sage: b = CBF(1, 2)
sage: a is b
False
sage: 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 == b
False
```

A ball is non-zero 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
```

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
```

2.7.3 Coercion

Automatic coercions work as expected:

```
sage: bpol = 1/3*CBF(i) + AA(sqrt(2)) + (polygen(RealBallField(20), 'x') + QQbar(i))
sage: bpol
x + [1.41421 +/- 5.09e-6] + [1.33333 +/- 3.97e-6]*I
sage: bpol.parent()
Univariate Polynomial Ring in x over Complex ball field with 20 bits precision
sage: bpol/3
([0.333333 +/- 4.93e-7])*x + [0.47140 +/- 5.39e-6] + [0.44444 +/- 4.98e-6]*I
```

TESTS:

```
sage: polygen(CBF, x)^3
x^3
```

2.7.4 Classes and Methods

class sage.rings.complex_arb.**ComplexBall**

Bases: sage.structure.element.RingElement

Hold one acb_t of the Arb library

EXAMPLES:

```
sage: a = ComplexBallField()(1, 1)
sage: a
1.000000000000000 + 1.000000000000000*I
```

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
```

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()
51
sage: CBF(I/2).accuracy() == CBF.base().maximal_accuracy()
True
sage: 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)
[1.000000000000000 +/- 1.01e-16] + [1.000000000000000 +/- 1.01e-16]*I
```

arg()

Return the argument of this complex ball.

EXAMPLES:

```
sage: CBF(1 + i).arg()
[0.785398163397448 +/- 3.91e-16]
sage: CBF(-1).arg()
[3.141592653589793 +/- 5.61e-16]
sage: CBF(-1).arg().parent()
Real ball field with 53 bits precision
```

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
```

See also:

`above_abs()`

conjugate()

Return the complex conjugate of this ball.

EXAMPLES:

```
sage: CBF(-2 + I/3).conjugate()
-2.000000000000000 + [-0.3333333333333333 +/- 7.04e-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))
Traceback (most recent call last):
...
TypeError: unsupported type: <type 'sage.symbolic.expression.Expression'>
```

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
```

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
```

imag()

Return the imaginary part of this ball.

OUTPUT:

A RealBall.

EXAMPLES:

```
sage: CBF = ComplexBallField()
sage: a = CBF(1/3, 1/5)
sage: a.imag()
[0.2000000000000000 +/- 4.45e-17]
```

is_exact()

Return True iff the radius of this ball is zero.

EXAMPLES:

```
sage: CBF = ComplexBallField()
sage: CBF(1).is_exact()
True
sage: CBF(1/3, 1/3).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 = ComplexBallField()
sage: CBF(pi, 1/3).is_nonzero()
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
```

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
```

is_zero()

Return True iff the midpoint and radius of this ball are both zero.

EXAMPLES:

```
sage: CBF = ComplexBallField()
sage: CBF(0).is_zero()
True
```

```
sage: CBF(RIF(-0.5, 0.5)).is_zero()
False
```

See also:

is_nonzero()

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()
0.333333333333333 + 1.00000000000000*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
```

See also:

squash()

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'))
True
sage: CBF(1, 1).overlaps(CBF(0, 'nan'))
False
```

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
```

TESTS:

```
sage: (CBF(0, 1/3) << (2^64)).rad()
Traceback (most recent call last):
...
RuntimeError: unable to convert the radius to MPFR (exponent out of range?)
```

real()

Return the real part of this ball.

OUTPUT:

A `RealBall`.

EXAMPLES:

```
sage: CBF = ComplexBallField()
sage: a = CBF(1/3, 1/5)
sage: a.real()
[0.3333333333333333 +/- 7.04e-17]
```

round()

Return a copy of this ball rounded to the precision of the parent.

EXAMPLES:

It is possible to create balls whose midpoint is more precise that their parent's nominal precision (see `real_arb` for more information):

```
sage: b = CBF(exp(I*pi/3).n(100))
sage: b.mid()
0.50000000000000000000000000000 + 0.86602540378443864676372317075*I
```

The `round()` method rounds such a ball to its parent's precision:

```
sage: b.round().mid()
0.500000000000000 + 0.866025403784439*I
```

See also:

`trim()`

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 +/- 1.49e-17] + [0.09999999999999999 +/- 1.68e-18]*I
sage: mid.parent()
```

```
Complex ball field with 53 bits precision
sage: mid.is_exact()
True
```

See also:

mid()

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
```

See also:

round()

class sage.rings.complex_arb.**ComplexBallField**(*precision*, *category*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.rings.ring.Field

An approximation of the field of complex numbers using pairs of mid-rad intervals.

INPUT:

- precision – an integer ≥ 2.

EXAMPLES:

```
sage: CBF = ComplexBallField() # indirect doctest
sage: CBF(1)
1.000000000000000
```

TESTS:

```
sage: ComplexBallField(0)
Traceback (most recent call last):
...
ValueError: Precision must be at least 2.
sage: ComplexBallField(1)
Traceback (most recent call last):
...
ValueError: Precision must be at least 2.
```

Element

alias of ComplexBall

characteristic()

Complex ball fields have characteristic zero.

EXAMPLES:

```
sage: 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
```

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 precision)
sage: functor(base) is CBF
True
```

gen(*i*)

For i = 0, return the imaginary unit in this complex ball field.

EXAMPLE:

```
sage: CBF.0
1.000000000000000*I
sage: CBF.gen(1)
Traceback (most recent call last):
...
ValueError: only one generator
```

gens()

Return the tuple of generators of this complex ball field, i.e. `(i,)`.

EXAMPLE:

```
sage: CBF.gens()
(1.000000000000000*I,)
sage: CBF.gens_dict()
{'1.000000000000000*I': 1.000000000000000*I}
```

is_exact()

Complex ball fields are not exact.

EXAMPLES:

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

is_finite()

Complex ball fields are infinite.

They already specify it via their category, but we currently need to re-implement this method due to the legacy implementation in `sage.rings.ring.Ring`.

EXAMPLES:

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

ngens()

Return 1 as the only generator is the imaginary unit.

EXAMPLE:

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

precision()

Return the bit precision used for operations on elements of this field.

EXAMPLES:

```
sage: 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,
 -0.5000000000000000*I,
 1.000000000000000 + [0.3333333333333333 +/- 1.49e-17]*I,
 [-0.3333333333333333 +/- 1.49e-17] + 0.2500000000000000*I,
 [-2.175556475109056e+1819614671183333366510562 +/- 1.29e+1819614671183333366510545],
 [+/- inf],
 [0.3333333333333333 +/- 1.49e-17] + [+/- inf]*I,
 [+/- inf] + [+/- inf]*I,
 nan,
 nan + nan*I,
 [+/- inf] + nan*I]
```

CHAPTER

THREE

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**()

Returns the lazy complex field.

EXAMPLES:

There is only one lazy complex field:

```
sage: ComplexLazyField() is ComplexLazyField()
True
```

class sage.rings.real_lazy.**ComplexLazyField_class**

Bases: sage.rings.real_lazy.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:

```
sage: a = CLF(-1).sqrt()
sage: a
1*I
sage: CDF(a)
1.0*I
sage: ComplexField(200)(a)
1.000000000000000000000000000000000000000000000000000000000000*I
```

TESTS:

```
sage: TestSuite(CLF).run(skip=["_test_prod"])
```

Note: The following TestSuite failure:

```
sage: CLF._test_prod()
Traceback (most recent call last):
...
AssertionError: False is not true
```

is due to (acceptable?) numerical noise:

```
sage: x = CLF.I
sage: x*x == x^2
False
sage: x*x
-1
sage: x^2
-0.9999999999999999? + 0.?e-15*I
```

---

construction ()

Returns 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
```

gen (*i=0*)

Return the i-th generator of `self`.

EXAMPLES:

```
sage: CLF.gen()
1*I
sage: ComplexField(100)(CLF.gen())
1.0000000000000000000000000000*I
```

interval_field (*prec=None*)

Returns 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)
Complex Interval Field with 333 bits of precision
sage: CLF.interval_field() is CIF
True
```

class `sage.rings.real_lazy.`**LazyAlgebraic**

Bases: `sage.rings.real_lazy.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?
```

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
0.62348980185873353052500488400423981063227473 + 0.7818314824680298087084452667405775023233

sage: a = LazyAlgebraic(CLF, QQ['x'].0^2-7, -2.0)
sage: RR(a)
-2.64575131106459
sage: RR(a)^2
7.00000000000000
```

class sage.rings.real_lazy.**LazyBinop**

Bases: sage.rings.real_lazy.LazyFieldElement

A lazy element representing a binary (usually arithmetic) operation between two other lazy elements.

EXAMPLES:

```
sage: from sage.rings.real_lazy import LazyBinop
sage: a = LazyBinop(RLF, 2, 1/3, operator.add)
sage: a
2.333333333333334?
sage: Reals(200)(a)
2.3333333333333333333333333333333333333333333333333333333333
```

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)
sage: 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)
14.0000000000000
```

A bit absurd:

```
sage: a.eval(str)
'68'
```

class sage.rings.real_lazy.**LazyConstant**

Bases: sage.rings.real_lazy.LazyFieldElement

This class represents a real or complex constant (such as `pi` or `I`).

TESTS:

```
sage: a = RLF.pi(); a
3.141592653589794?
sage: RealField(300)(a)
3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825

sage: from sage.rings.real_lazy import LazyConstant
sage: a = LazyConstant(RLF, 'euler_constant')
sage: RealField(200)(a)
0.57721566490153286060651209008240243104215933593992359880577
```

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)
1.00000000000000*I
```

class `sage.rings.real_lazy.`**LazyField**

Bases: `sage.rings.ring.Field`

The base class for lazy real fields.

> **Warning:** LazyField uses `__getattr__()`, to implement:
>
> ```
> sage: 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'>
> ```

algebraic_closure()

Returns 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
```

interval_field(*prec=None*)

Abstract method to create the corresponding interval field.

TESTS:

```
sage: RLF.interval_field() # indirect doctest
Real Interval Field with 53 bits of precision
```

class sage.rings.real_lazy.**LazyFieldElement**

Bases: sage.structure.element.FieldElement

approx()

Returns self as an element of an interval field.

EXAMPLES:

```
sage: CLF(1/6).approx()
0.1666666666666667?
sage: CLF(1/6).approx().parent()
Complex Interval Field with 53 bits of precision
```

When the absolute value is involved, the result might be real:

```
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
```

continued_fraction()

Return the continued fraction of self.

EXAMPLES:

```
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)
444927297812646558239761867973501208151173610180916865469/14141446664917497333518357185434O3
```

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
```

eval(*R*)

Abstract method for converting self into an element of R.

EXAMPLES:

```
sage: a = RLF(12)
sage: a.eval(ZZ)
12
```

class sage.rings.real_lazy.**LazyNamedUnop**

Bases: sage.rings.real_lazy.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)
2.00000000000000
```

approx()

Does something reasonable with functions that are not defined on the interval fields.

TESTS:

```
sage: from sage.rings.real_lazy import LazyNamedUnop
sage: LazyNamedUnop(RLF, 8, 'sqrt') # indirect doctest
2.828427124746190?
```

eval(*R*)

Convert `self` into an element of `R`.

TESTS:

```
sage: from sage.rings.real_lazy import LazyNamedUnop
sage: a = LazyNamedUnop(RLF, 4, 'sqrt')
sage: RR(a) # indirect doctest
2.00000000000000
sage: a.sqrt()
1.414213562373095?
sage: RealField(212)(a)
2.00000000000000000000000000000000000000000000000000000000000000
sage: float(a)
2.0
```

Now for some extra arguments:

```
sage: a = RLF(100)
sage: a.log(10)
2
sage: float(a.log(10))
2.0
```

class sage.rings.real_lazy.**LazyUnop**

Bases: sage.rings.real_lazy.LazyFieldElement

Represents 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
3.0000000000000000000000000000
```

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
```

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)
sqrt(3)
```

class `sage.rings.real_lazy.`**LazyWrapper**

Bases: `sage.rings.real_lazy.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
```

continued_fraction()

Return the continued fraction of self.

EXAMPLES:

```
sage: a = RLF(sqrt(2))
sage: a.continued_fraction()
[1; 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]
```

depth()

Returns the depth of `self` as an expression, which is always 0.

EXAMPLES:

```
sage: RLF(4).depth()
0
```

eval(*R*)

Convert `self` into an element of `R`.

EXAMPLES:

```
sage: a = RLF(12)
sage: a.eval(ZZ)
12
sage: a.eval(ZZ).parent()
Integer Ring
```

class `sage.rings.real_lazy.`**`LazyWrapperMorphism`**

Bases: `sage.categories.morphism.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)
<type 'sage.rings.real_lazy.LazyWrapper'>
sage: a._value
3
sage: 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
```

class `sage.rings.real_lazy.`**`RealLazyField_class`**

Bases: `sage.rings.real_lazy.LazyField`

This class represents the set of real numbers to unspecified precision. For the most part it simply wraps exact elements and defers evaluation until a specified precision is requested.

Its primary use is to connect the exact rings (such as number fields) to fixed precision real numbers. For example, to specify an embedding of a number field K into $\mathbf{R}$ one can map into this field and the coercion will then be able to carry the mapping to real fields of any precision.

EXAMPLES:

```
sage: a = RLF(1/3)
sage: a
0.3333333333333334?
sage: a + 1/5
0.5333333333333334?
sage: a = RLF(1/3)
```

```
sage: a
0.3333333333333334?
sage: a + 5
5.333333333333334?
sage: RealField(100)(a+5)
5.3333333333333333333333333333
```

TESTS:

```
sage: TestSuite(RLF).run()
```

construction()

Returns the functorial construction of `self`, namely, the completion of the rationals at infinity to infinite precision.

EXAMPLES:

```
sage: c, S = RLF.construction(); S
Rational Field
sage: RLF == c(S)
True
```

gen(*i=0*)

Return the i-th generator of `self`.

EXAMPLES:

```
sage: RLF.gen()
1
```

interval_field(*prec=None*)

Returns 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
```

`sage.rings.real_lazy.`**make_element**(*parent*, **args*)

Create an element of `parent`.

EXAMPLES:

```
sage: a = RLF(pi) + RLF(sqrt(1/2)) # indirect doctest
sage: loads(dumps(a)) == a
True
```

CHAPTER

FOUR

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

r

INDEX

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

Y

Z