Home

Pages Classes Methods

Search

Table of Contents

What's Here
Querying
Comparing
Converting

Show/hide navigation

Parent

Numeric

Methods

#%
#*
#**
#+
#-
#-@
#/
#<
#<=
#<=>
#==
#===
#>
#>=
#abs
#angle
#arg
#ceil
#coerce
#denominator
#divmod
#eql?
#fdiv
#finite?
#floor
#hash
#infinite?
#inspect
#magnitude
#modulo
#nan?
#negative?
#next_float
#numerator
#phase
#positive?
#prev_float
#quo

class Float

A Float object represents a sometimes-inexact real number using the native architecture's double-precision floating point representation.

Floating point has a different arithmetic and is an inexact number. So you should know its esoteric system. See following:

- docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
- github.com/rdp/ruby_tutorials_core/wiki/Ruby-Talk-FAQ#-why-are-rubys-floats-imprecise
- en.wikipedia.org/wiki/Floating_point#Accuracy_problems

You can create a Float object explicitly with:

- A floating-point literal.

You can convert certain objects to Floats with:

- Method `Float`.

What's Here

First, what's elsewhere. Class Float:

- Inherits from class Numeric.

Here, class Float provides methods for:

- Querying
- Comparing
- Converting

Querying

- `finite?` : Returns whether `self` is finite.
- `hash` : Returns the integer hash code for `self` .
- `infinite?` : Returns whether `self` is infinite.
- `nan?` : Returns whether `self` is a NaN (not-a-number).

Comparing

- #<: Returns whether `self` is less than the given value.
- #<=: Returns whether `self` is less than or equal to the given value.
- #<=>: Returns a number indicating whether `self` is less than, equal to, or greater than the given value.
- `==` (aliased as `===` and `eql?`): Returns whether `self` is equal to the given value.
- #>: Returns whether `self` is greater than the given value.
- #>=: Returns whether `self` is greater than or equal to the given value.

Converting

- `%` (aliased as `modulo`): Returns `self` modulo the given value.
- `*` : Returns the product of `self` and the given value.
- `**` : Returns the value of `self` raised to the power of the given value.
- `+` : Returns the sum of `self` and the given value.
- `-` : Returns the difference of `self` and the given value.
- #/: Returns the quotient of `self` and the given value.
- `ceil` : Returns the smallest number greater than or equal to `self` .

- `coerce` : Returns a 2-element array containing the given value converted to a Float and `self`
- `divmod` : Returns a 2-element array containing the quotient and remainder results of dividing `self` by the given value.
- `fdiv` : Returns the Float result of dividing `self` by the given value.
- `floor` : Returns the greatest number smaller than or equal to `self` .
- `next_float` : Returns the next-larger representable Float.
- `prev_float` : Returns the next-smaller representable Float.
- `quo` : Returns the quotient from dividing `self` by the given value.
- `round` : Returns `self` rounded to the nearest value, to a given precision.
- `to_i` (aliased as `to_int`): Returns `self` truncated to an `Integer` .
- `to_s` (aliased as `inspect`): Returns a string containing the place-value representation of `self` in the given radix.
- `truncate` : Returns `self` truncated to a given precision.

Constants

DIG

The minimum number of significant decimal digits in a double-precision floating point.

Usually defaults to 15.

EPSILON

The difference between 1 and the smallest double-precision floating point number greater than 1.

Usually defaults to 2.2204460492503131e-16.

INFINITY

An expression representing positive infinity.

MANT_DIG

The number of base digits for the `double` data type.

Usually defaults to 53.

MAX

The largest possible integer in a double-precision floating point number.

Usually defaults to 1.7976931348623157e+308.

MAX_10_EXP

The largest positive exponent in a double-precision floating point where 10 raised to this power minus 1.

Usually defaults to 308.

MAX_EXP

The largest possible exponent value in a double-precision floating point.

Usually defaults to 1024.

MIN

The smallest positive normalized number in a double-precision floating point.

Usually defaults to 2.2250738585072014e-308.

If the platform supports denormalized numbers, there are numbers between zero and `Float::MIN`. 0.0.next_float returns the smallest positive floating point number including denormalized numbers.

MIN_10_EXP

The smallest negative exponent in a double-precision floating point where 10 raised to this power minus 1.

Usually defaults to -307.

MIN_EXP

The smallest possible exponent value in a double-precision floating point.

Usually defaults to -1021.

NAN

An expression representing a value which is “not a number”.

RADIX

The base of the floating point, or number of unique digits used to represent the number.

Usually defaults to 2 on most systems, which would represent a base-10 decimal.

Public Instance Methods

self % other → float

Returns `self` modulo `other` as a float.

For float `f` and real number `r`, these expressions are equivalent:

```
f % r
f-r*(f/r).floor
f.divmod(r)[1]
```

See `Numeric#divmod`.

Examples:

```
10.0 % 2               # => 0.0
10.0 % 3               # => 1.0
10.0 % 4               # => 2.0

10.0 % -2              # => 0.0
10.0 % -3              # => -2.0
10.0 % -4              # => -2.0

10.0 % 4.0             # => 2.0
10.0 % Rational(4, 1) # => 2.0
```

Also aliased as: modulo

self * other → numeric

Returns a new Float which is the product of `self` and `other`:

```
f = 3.14
f * 2              # => 6.28
f * 2.0            # => 6.28
f * Rational(1, 2) # => 1.57
f * Complex(2, 0)  # => (6.28+0.0i)
```

self ** other → numeric

Raises `self` to the power of `other`:

```
f = 3.14
f ** 2              # => 9.8596
f ** -2             # => 0.1014239928597509
f ** 2.1            # => 11.054834900588839
f ** Rational(2, 1) # => 9.8596
f ** Complex(2, 0)  # => (9.8596+0i)
```

self + other → numeric

Returns a new Float which is the sum of `self` and `other`:

```
f = 3.14
f + 1                 # => 4.140000000000001
f + 1.0               # => 4.140000000000001
f + Rational(1, 1)    # => 4.140000000000001
f + Complex(1, 0)     # => (4.140000000000001+0i)
```

self - other → numeric

Returns a new Float which is the difference of `self` and `other`:

```
f = 3.14
f - 1                 # => 2.14
f - 1.0               # => 2.14
f - Rational(1, 1)    # => 2.14
f - Complex(1, 0)     # => (2.14+0i)
```

-float → float

Returns `self`, negated.

self / other → numeric

Returns a new Float which is the result of dividing `self` by `other`:

```
f = 3.14
f / 2              # => 1.57
f / 2.0            # => 1.57
f / Rational(2, 1) # => 1.57
f / Complex(2, 0)  # => (1.57+0.0i)
```

self < other → true or false

Returns `true` if `self` is numerically less than `other`:

```
2.0 < 3              # => true
2.0 < 3.0            # => true
2.0 < Rational(3, 1) # => true
2.0 < 2.0            # => false
```

`Float::NAN < Float::NAN` returns an implementation-dependent value.

self <= other → true or false

Returns `true` if `self` is numerically less than or equal to `other`:

```
2.0 <= 3              # => true
2.0 <= 3.0            # => true
2.0 <= Rational(3, 1) # => true
2.0 <= 2.0            # => true
2.0 <= 1.0            # => false
```

`Float::NAN <= Float::NAN` returns an implementation-dependent value.

self <=> other → -1, 0, +1, or nil

Returns a value that depends on the numeric relation between `self` and `other`:

- -1, if `self` is less than `other`.
- 0, if `self` is equal to `other`.
- 1, if `self` is greater than `other`.
- `nil`, if the two values are incommensurate.

Examples:

```
2.0 <=> 2             # => 0
2.0 <=> 2.0           # => 0
2.0 <=> Rational(2, 1) # => 0
2.0 <=> Complex(2, 0)  # => 0
2.0 <=> 1.9           # => 1
2.0 <=> 2.1           # => -1
2.0 <=> 'foo'         # => nil
```

This is the basis for the tests in the `Comparable` module.

`Float::NAN <=> Float::NAN` returns an implementation-dependent value.

self == other → true or false

Returns `true` if `other` has the same value as `self`, `false` otherwise:

```
2.0 == 2              # => true
2.0 == 2.0            # => true
2.0 == Rational(2, 1) # => true
2.0 == Complex(2, 0)  # => true
```

`Float::NAN == Float::NAN` returns an implementation-dependent value.

Related: `Float#eql?` (requires `other` to be a Float).

Also aliased as: `===`

===(p1)

Returns `true` if `other` has the same value as `self`, `false` otherwise:

```
2.0 == 2              # => true
2.0 == 2.0            # => true
2.0 == Rational(2, 1) # => true
2.0 == Complex(2, 0)  # => true
```

`Float::NAN == Float::NAN` returns an implementation-dependent value.

Related: [Float#eql?](#) (requires `other` to be a Float).

Alias for: [==](#)

self > other → true or false

Returns `true` if `self` is numerically greater than `other`:

```
2.0 > 1              # => true
2.0 > 1.0            # => true
2.0 > Rational(1, 2) # => true
2.0 > 2.0            # => false
```

`Float::NAN > Float::NAN` returns an implementation-dependent value.

self >= other → true or false

Returns `true` if `self` is numerically greater than or equal to `other`:

```
2.0 >= 1              # => true
2.0 >= 1.0            # => true
2.0 >= Rational(1, 2) # => true
2.0 >= 2.0            # => true
2.0 >= 2.1            # => false
```

`Float::NAN >= Float::NAN` returns an implementation-dependent value.

abs → float

Returns the absolute value of `self`:

```
(-34.56).abs # => 34.56
-34.56.abs   # => 34.56
34.56.abs    # => 34.56
```

angle → 0 or float

Returns 0 if the value is positive, pi otherwise.

Alias for: arg

arg → 0 or float

Returns 0 if the value is positive, pi otherwise.

Also aliased as: angle, phase

ceil(ndigits = 0) → float or integer

Returns the smallest number greater than or equal to `self` with a precision of `ndigits` decimal digits.

When `ndigits` is positive, returns a float with `ndigits` digits after the decimal point (as available):

```
f = 12345.6789
f.ceil(1) # => 12345.7
f.ceil(3) # => 12345.679
f = -12345.6789
f.ceil(1) # => -12345.6
f.ceil(3) # => -12345.678
```

When `ndigits` is non-positive, returns an integer with at least `ndigits.abs` trailing zeros:

```
f = 12345.6789
f.ceil(0)  # => 12346
f.ceil(-3) # => 13000
f = -12345.6789
f.ceil(0)  # => -12345
f.ceil(-3) # => -12000
```

Note that the limited precision of floating-point arithmetic may lead to surprising results:

```
(2.1 / 0.7).ceil  #=> 4 (!)
```

Related: `Float#floor`.

coerce(other) → array

Returns a 2-element array containing `other` converted to a Float and `self`:

```
f = 3.14                # => 3.14
f.coerce(2)             # => [2.0, 3.14]
f.coerce(2.0)           # => [2.0, 3.14]
f.coerce(Rational(1, 2)) # => [0.5, 3.14]
f.coerce(Complex(1, 0))  # => [1.0, 3.14]
```

Raises an exception if a type conversion fails.

denominator → integer

Returns the denominator (always positive). The result is machine dependent.

See also `Float#numerator`.

divmod(other) → array

Returns a 2-element array `[q, r]`, where

```
q = (self/other).floor      # Quotient
r = self % other            # Remainder
```

Examples:

```
11.0.divmod(4)              # => [2, 3.0]
11.0.divmod(-4)             # => [-3, -1.0]
-11.0.divmod(4)             # => [-3, 1.0]
-11.0.divmod(-4)            # => [2, -3.0]

12.0.divmod(4)              # => [3, 0.0]
12.0.divmod(-4)             # => [-3, 0.0]
-12.0.divmod(4)             # => [-3, -0.0]
-12.0.divmod(-4)            # => [3, -0.0]

13.0.divmod(4.0)            # => [3, 1.0]
13.0.divmod(Rational(4, 1)) # => [3, 1.0]
```

eql?(other) → true or false

Returns `true` if `other` is a Float with the same value as `self`, `false` otherwise:

```
2.0.eql?(2.0)            # => true
2.0.eql?(1.0)            # => false
2.0.eql?(1)              # => false
2.0.eql?(Rational(2, 1)) # => false
2.0.eql?(Complex(2, 0))  # => false
```

`Float::NAN.eql?(Float::NAN)` returns an implementation-dependent value.

Related: `Float#==` (performs type conversions).

fdiv(p1)

Returns the quotient from dividing `self` by `other`:

```
f = 3.14
f.quo(2)              # => 1.57
f.quo(-2)             # => -1.57
f.quo(Rational(2, 1)) # => 1.57
f.quo(Complex(2, 0))  # => (1.57+0.0i)
```

Alias for: quo

finite? → true or false

Returns `true` if `self` is not `Infinity`, `-Infinity`, or `NaN`, `false` otherwise:

```
f = 2.0      # => 2.0
f.finite?    # => true
f = 1.0/0.0  # => Infinity
f.finite?    # => false
f = -1.0/0.0 # => -Infinity
f.finite?    # => false
f = 0.0/0.0  # => NaN
f.finite?    # => false
```

floor(ndigits = 0) → float or integer

Returns the largest number less than or equal to `self` with a precision of `ndigits` decimal digits.

When `ndigits` is positive, returns a float with `ndigits` digits after the decimal point (as available):

```
f = 12345.6789
f.floor(1) # => 12345.6
f.floor(3) # => 12345.678
f = -12345.6789
f.floor(1) # => -12345.7
f.floor(3) # => -12345.679
```

When `ndigits` is non-positive, returns an integer with at least `ndigits.abs` trailing zeros:

```
f = 12345.6789
f.floor(0)  # => 12345
f.floor(-3) # => 12000
f = -12345.6789
f.floor(0)  # => -12346
f.floor(-3) # => -13000
```

Note that the limited precision of floating-point arithmetic may lead to surprising results:

```
(0.3 / 0.1).floor  #=> 2 (!)
```

Related: Float#ceil.

hash → integer

Returns the integer hash value for self.

See also Object#hash.

infinite? → -1, 1, or nil

Returns:

- 1, if self is Infinity.
- -1 if self is -Infinity.
- nil, otherwise.

Examples:

```
f = 1.0/0.0  # => Infinity
f.infinite?  # => 1
f = -1.0/0.0 # => -Infinity
f.infinite?  # => -1
f = 1.0      # => 1.0
f.infinite?  # => nil
f = 0.0/0.0  # => NaN
f.infinite?  # => nil
```

inspect()

Returns a string containing a representation of self; depending of the value of self, the string representation may contain:

- A fixed-point number.
- A number in “scientific notation” (containing an exponent).
- ‘Infinity’.
- ‘-Infinity’.
- ‘NaN’ (indicating not-a-number).

  3.14.to_s # => “3.14” (10.1**50).to_s # => “1.644631821843879e+50” (10.1**500).to_s # => “Infinity” (-10.1**500).to_s # => “-Infinity” (0.0/0.0).to_s # =>

"NaN"

Alias for: to_s

magnitude()

modulo(p1)

Returns `self` modulo `other` as a float.

For float `f` and real number `r`, these expressions are equivalent:

```
f % r
f-r*(f/r).floor
f.divmod(r)[1]
```

See `Numeric#divmod`.

Examples:

```
10.0 % 2                # => 0.0
10.0 % 3                # => 1.0
10.0 % 4                # => 2.0

10.0 % -2               # => 0.0
10.0 % -3               # => -2.0
10.0 % -4               # => -2.0

10.0 % 4.0              # => 2.0
10.0 % Rational(4, 1) # => 2.0
```

Alias for: %

nan? → true or false

Returns `true` if `self` is a NaN, `false` otherwise.

```
f = -1.0      #=> -1.0
f.nan?        #=> false
f = 0.0/0.0   #=> NaN
f.nan?        #=> true
```

negative? → true or false

Returns `true` if `self` is less than 0, `false` otherwise.

next_float → float

Returns the next-larger representable Float.

These examples show the internally stored values (64-bit hexadecimal) for each Float `f` and for the corresponding `f.next_float`:

```
f = 0.0      # 0x0000000000000000
f.next_float # 0x0000000000000001

f = 0.01     # 0x3f847ae147ae147b
f.next_float # 0x3f847ae147ae147c
```

In the remaining examples here, the output is shown in the usual way (result `to_s`):

```
0.01.next_float    # => 0.010000000000000002
1.0.next_float     # => 1.0000000000000002
100.0.next_float   # => 100.00000000000001

f = 0.01
(0..3).each_with_index {|i| printf "%2d %-20a %s\n", i, f, f.to_s; f = f.next
```

Output:

```
 0 0x1.47ae147ae147bp-7 0.01
 1 0x1.47ae147ae147cp-7 0.010000000000000002
 2 0x1.47ae147ae147dp-7 0.010000000000000004
 3 0x1.47ae147ae147ep-7 0.010000000000000005

f = 0.0; 100.times { f += 0.1 }
f                           # => 9.99999999999998       # should be 10.0 in t
10-f                        # => 1.9539925233402755e-14 # the floating point
10.0.next_float-10          # => 1.7763568394002505e-15 # 1 ulp (unit in the
(10-f)/(10.0.next_float-10) # => 11.0                   # the error is 11 ulp
(10-f)/(10*Float::EPSILON)  # => 8.8                    # approximation of th
"%a" % 10                   # => "0x1.4p+3"
"%a" % f                    # => "0x1.3fffffffffff5p+3" # the last hex digit
```

Related: [Float#prev_float](#)

numerator → integer

Returns the numerator. The result is machine dependent.

```
n = 0.3.numerator    #=> 5404319552844595
d = 0.3.denominator  #=> 18014398509481984
n.fdiv(d)            #=> 0.3
```

See also [Float#denominator](#).

phase → 0 or float

Returns 0 if the value is positive, pi otherwise.

Alias for: arg

positive? → true or false

Returns `true` if `self` is greater than 0, `false` otherwise.

prev_float → float

Returns the next-smaller representable Float.

These examples show the internally stored values (64-bit hexadecimal) for each Float `f` and for the corresponding `f.pev_float`:

```
f = 5e-324   # 0x0000000000000001
f.prev_float # 0x0000000000000000

f = 0.01     # 0x3f847ae147ae147b
f.prev_float # 0x3f847ae147ae147a
```

In the remaining examples here, the output is shown in the usual way (result `to_s`):

```
0.01.prev_float   # => 0.009999999999999998
1.0.prev_float    # => 0.9999999999999999
100.0.prev_float  # => 99.99999999999999

f = 0.01
(0..3).each_with_index {|i| printf "%2d %-20a %s\n", i, f, f.to_s; f = f.prev
```

Output:

```
0 0x1.47ae147ae147bp-7 0.01
1 0x1.47ae147ae147ap-7 0.009999999999999998
2 0x1.47ae147ae1479p-7 0.009999999999999997
3 0x1.47ae147ae1478p-7 0.009999999999999995
```

Related: `Float#next_float`.

quo(other) → numeric

Returns the quotient from dividing `self` by `other`:

```
f = 3.14
f.quo(2)              # => 1.57
f.quo(-2)             # => -1.57
```

```
f.quo(Rational(2, 1)) # => 1.57
f.quo(Complex(2, 0))  # => (1.57+0.0i)
```

Also aliased as: fdiv

rationalize([eps]) → rational

Returns a simpler approximation of the value (flt-|eps| <= result <= flt+|eps|). If the optional argument `eps` is not given, it will be chosen automatically.

```
0.3.rationalize          #=> (3/10)
1.333.rationalize        #=> (1333/1000)
1.333.rationalize(0.01)  #=> (4/3)
```

See also `Float#to_r`.

round(ndigits = 0, half: :up]) → integer or float

Returns `self` rounded to the nearest value with a precision of `ndigits` decimal digits.

When `ndigits` is non-negative, returns a float with `ndigits` after the decimal point (as available):

```
f = 12345.6789
f.round(1) # => 12345.7
f.round(3) # => 12345.679
f = -12345.6789
f.round(1) # => -12345.7
f.round(3) # => -12345.679
```

When `ndigits` is negative, returns an integer with at least `ndigits.abs` trailing zeros:

```
f = 12345.6789
f.round(0)  # => 12346
f.round(-3) # => 12000
f = -12345.6789
f.round(0)  # => -12346
f.round(-3) # => -12000
```

If keyword argument `half` is given, and `self` is equidistant from the two candidate values, the rounding is according to the given `half` value:

- `:up` or `nil`: round away from zero:

```
2.5.round(half: :up)      # => 3
3.5.round(half: :up)      # => 4
```

```
(-2.5).round(half: :up)    # => -3
```

- `:down` : round toward zero:

```
2.5.round(half: :down)    # => 2
3.5.round(half: :down)    # => 3
(-2.5).round(half: :down) # => -2
```

- `:even` : round toward the candidate whose last nonzero digit is even:

```
2.5.round(half: :even)    # => 2
3.5.round(half: :even)    # => 4
(-2.5).round(half: :even) # => -2
```

Raises and exception if the value for `half` is invalid.

Related: `Float#truncate`.

to_f → self

Returns `self` (which is already a Float).

to_i → integer

Returns `self` truncated to an `Integer`.

```
1.2.to_i    # => 1
(-1.2).to_i # => -1
```

Note that the limited precision of floating-point arithmetic may lead to surprising results:

```
(0.3 / 0.1).to_i  # => 2 (!)
```

Also aliased as: to_int

to_int()

Returns `self` truncated to an `Integer`.

```
1.2.to_i    # => 1
(-1.2).to_i # => -1
```

Note that the limited precision of floating-point arithmetic may lead to surprising results:

```
(0.3 / 0.1).to_i  # => 2 (!)
```

Alias for: to_i

to_r → rational

Returns the value as a rational.

```
2.0.to_r    #=> (2/1)
2.5.to_r    #=> (5/2)
-0.75.to_r  #=> (-3/4)
0.0.to_r    #=> (0/1)
0.3.to_r    #=> (5404319552844595/18014398509481984)
```

NOTE: 0.3.to_r isn't the same as "0.3".to_r. The latter is equivalent to "3/10".to_r, but the former isn't so.

```
0.3.to_r   == 3/10r  #=> false
"0.3".to_r == 3/10r  #=> true
```

See also `Float#rationalize`.

to_s → string

Returns a string containing a representation of `self`; depending of the value of `self`, the string representation may contain:

- A fixed-point number.
- A number in "scientific notation" (containing an exponent).
- 'Infinity'.
- '-Infinity'.
- 'NaN' (indicating not-a-number).

  3.14.to_s # => "3.14" (10.1**50).to_s # => "1.644631821843879e+50" (10.1**500).to_s # => "Infinity" (-10.1**500).to_s # => "-Infinity" (0.0/0.0).to_s # => "NaN"

Also aliased as: inspect

truncate(ndigits = 0) → float or integer

Returns `self` truncated (toward zero) to a precision of `ndigits` decimal digits.

When `ndigits` is positive, returns a float with `ndigits` digits after the decimal point (as available):

```
f = 12345.6789
f.truncate(1) # => 12345.6
f.truncate(3) # => 12345.678
f = -12345.6789
f.truncate(1) # => -12345.6
f.truncate(3) # => -12345.678
```

When `ndigits` is negative, returns an integer with at least `ndigits.abs` trailing zeros:

```
f = 12345.6789
f.truncate(0)  # => 12345
f.truncate(-3) # => 12000
f = -12345.6789
f.truncate(0)  # => -12345
f.truncate(-3) # => -12000
```

Note that the limited precision of floating-point arithmetic may lead to surprising results:

```
(0.3 / 0.1).truncate  #=> 2 (!)
```

Related: `Float#round`.

zero? → true or false

Returns `true` if `self` is 0.0, `false` otherwise.