Class **Numeric** < Object

Subclasses: Float, Integer

1.9 Numeric is the fundamental base type for the abstract class Integer and the concrete number classes Bignum, Complex, Float, Fixnum, and Rational. Many methods in Numeric are overridden in child classes, and Numeric takes some liberties by calling methods in these child classes. A complete list of the methods defined in all five classes is shown in Table 27.13 on page 618.

Mixes in

Comparable:

```
<, <=, ==, >=, >, between?
```

Instance methods

+@ — *+num* → *num*

Unary Plus—Returns the receiver's value.

-@ — *–num* → *numeric*

Unary Minus—Returns the receiver's value, negated.

<=> — *num* <=> *other* → 0 or nil

Returns zero if *num* equals *other* and returns nil otherwise.

abs — *num*.abs → *numeric*

Returns the absolute value of *num*.

```
12.abs          # =>   12
(-34.56).abs    # =>   34.56
-34.56.abs      # =>   34.56
```

abs2 — *num*.abs2 → *numeric*

1.9 Returns the square of (the absolute value of) *num*.

```
12.abs2          # =>   144
(-34.56).abs2    # =>   1194.3936
-34.56.abs2      # =>   1194.3936
```

angle — *num*.angle → *numeric*

1.9 For noncomplex numbers, returns π for negative numbers, 0 otherwise. See Complex for more details.

arg — *num*.arg → *numeric*

1.9 Synonym for Numeric#angle.

ceil

num.ceil → *int*

Returns the smallest Integer greater than or equal to *num*. Class Numeric achieves this by converting itself to a Float and then invoking Float#ceil.

```
1.ceil       # =>   1
1.2.ceil     # =>   2
(-1.2).ceil  # =>   -1
(-1.0).ceil  # =>   -1
```

coerce

num.coerce(*numeric*) → *array*

coerce is both an instance method of Numeric and part of a type conversion protocol. When a number is asked to perform an operation and it is passed a parameter of a class different from its own, it must first coerce both itself and that parameter into a common class so that the operation makes sense. For example, in the expression $1 + 2.5$, the Fixnum 1 must be converted to a Float to make it compatible with 2.5. This conversion is performed by coerce. For all numeric objects, coerce is straightforward: if *numeric* is the same type as *num*, returns an array containing *numeric* and *num*. Otherwise, returns an array with both *numeric* and *num* represented as Float objects.

```
1.coerce(2.5)   # =>   [2.5, 1.0]
1.2.coerce(3)   # =>   [3.0, 1.2]
1.coerce(2)     # =>   [2, 1]
```

If a numeric object is asked to operate on a non-numeric, it tries to invoke coerce on that other object. For example, if you write this:

```
1 + "2"
```

Ruby will effectively execute the code as follows:

```
n1, n2 = "2".coerce(1)
n2 + n1
```

In the more general case, this won't work, because most non-numerics don't define a coerce method. However, you can use this (if you feel so inclined) to implement part of Perl's automatic conversion of strings to numbers in expressions.

```
class String
  def coerce(other)
    case other
    when Integer
      begin
        return other, Integer(self)
      rescue
        return Float(other), Float(self)
      end
    when Float
      return other, Float(self)
    else super
    end
  end
end
```

```
1   + "2"     # =>   3
1   - "2.3"   # =>   -1.3
1.2 + "2.3"   # =>   3.5
1.5 - "2"     # =>   -0.5
```

coerce is discussed further on page 380.

conj

num.conj → *num*

1.9 Synonym for Numeric#conjugate.

conjugate

num.conjugate → *num*

1.9 Returns the complex conjugate of *num*. For noncomplex numbers, returns *num*.

denominator

num.denominator → *integer*

1.9 Returns the denominator of the rational representation of *num*.

```
1.denominator     # =>   1
1.5.denominator   # =>   2
num = 1.0/3
num.to_r          # =>   (6004799503160661/18014398509481984)
num.denominator   # =>   18014398509481984
```

div

num.div(*numeric*) → *int*

Uses / to perform division and then converts the result to an integer. Numeric does not define the / operator; this is left to subclasses.

divmod

num.divmod(*numeric*) → *array*

Returns an array containing the quotient and modulus obtained by dividing *num* by *numeric*. If q,r = x.divmod(y), $q = floor(float(x)/float(y))$ and $x = q \times y + r$. The quotient is rounded toward $-\infty$. See Table 27.14 on page 619 for examples.

eql?

num.eql?(*numeric*) → true or false

Returns true if *num* and *numeric* are the same type and have equal values.

```
1 == 1.0          # =>   true
1.eql?(1.0)       # =>   false
(1.0).eql?(1.0)   # =>   true
```

fdiv

num.fdiv(*numeric*) → *numeric*

1.9 Synonym for Numeric#quo.

floor

num.floor → *int*

Returns the largest integer less than or equal to *num*. Numeric implements this by converting *int* to a Float and invoking Float#floor.

```
1.floor      # =>   1
(-1).floor   # =>   -1
```

Table 27.13: Methods defined in class Numeric and its subclasses. A ✓ means that the method is defined in the corresponding class.

	Numeric	Integer	Fixnum	Bignum	Float
%	–	–	✓	✓	✓
&	–	–	✓	✓	–
*	–	–	✓	✓	✓
**	–	–	✓	✓	✓
+	–	–	✓	✓	✓
+@	✓	–	–	–	–
-	–	–	✓	✓	✓
-@	✓	–	✓	✓	✓
/	–	–	✓	✓	✓
<	–	–	✓	–	✓
<<	–	–	✓	✓	–
<=	–	–	✓	–	✓
<=>	✓	–	✓	✓	✓
==	–	–	✓	✓	✓
>	–	–	✓	–	✓
>=	–	–	✓	–	✓
>>	–	–	✓	✓	–
[ ]	–	–	✓	✓	–
^	–	–	✓	✓	–
abs	✓	–	✓	✓	✓
abs2	✓	–	–	–	–
angle	✓	–	–	–	–
arg	✓	–	–	–	–
ceil	✓	✓	–	–	✓
chr	–	✓	–	–	–
coerce	✓	–	–	✓	✓
conj	✓	–	–	–	–
conjugate	✓	–	–	–	–
denominator	✓	✓	–	–	–
div	✓	–	✓	✓	–
divmod	✓	–	✓	✓	✓
downto	–	✓	–	–	–
eql?	✓	–	–	✓	✓
even?	–	✓	✓	✓	–
fdiv	✓	–	✓	✓	✓
finite?	–	–	–	–	✓
floor	✓	✓	–	–	✓
gcd	–	✓	–	–	–
gcdlcm	–	✓	–	–	–
hash	–	–	–	✓	✓
imag	✓	–	–	–	–
imaginary	✓	–	–	–	–
infinite?	–	–	–	–	✓
integer?	✓	✓	–	–	–
lcm	–	✓	–	–	–
magnitude	✓	–	✓	✓	✓
modulo	✓	–	✓	✓	✓
nan?	–	–	–	–	✓
next	–	✓	–	–	–
nonzero?	✓	–	–	–	–
numerator	✓	✓	–	–	–
odd?	–	✓	✓	✓	–
ord	–	✓	–	–	–
phase	✓	–	–	–	–
polar	✓	–	–	–	–
pred	–	✓	–	–	–
quo	✓	–	–	–	✓
real	✓	–	–	–	–
real?	✓	–	–	–	–
rect	✓	–	–	–	–
rectangular	✓	–	–	–	–
remainder	✓	–	–	✓	–
round	✓	✓	–	–	✓
singleton_method_added	✓	–	–	–	–
size	–	–	✓	✓	–
step	✓	–	–	–	–
succ	–	✓	✓	–	–
times	–	✓	–	–	–
to_c	✓	–	–	–	–
to_f	–	–	✓	✓	✓
to_i	–	✓	–	–	✓
to_int	✓	✓	–	–	✓
to_r	–	✓	–	–	✓
to_s	–	–	✓	✓	✓
truncate	✓	✓	–	–	✓
upto	–	✓	–	–	–
zero?	✓	–	✓	–	✓
|	–	–	✓	✓	–
~	–	–	✓	✓	–

Table 27.14. Difference between modulo and remainder. The modulo operator ("%") always has the sign of the divisor whereas remainder has the sign of the dividend.

a	b	a.divmod(b)	a / b	a.modulo(b)	a.remainder(b)
13	4	3, 1	3	1	1
13	−4	−4, −3	−4	−3	1
−13	4	−4, 3	−4	3	−1
−13	−4	3, −1	3	−1	−1
11.5	4	2, 3.5	2.875	3.5	3.5
11.5	−4	−3, −0.5	−2.875	−0.5	3.5
−11.5	4	−3, 0.5	−2.875	0.5	−3.5
−11.5	−4	2, −3.5	2.875	−3.5	−3.5

imag

num.imag → 0

1.9

Synonym for Numeric#imaginary.

imaginary

num.image → 0

1.9

Returns the imaginary part of *num*. Always 0 unless *num* is a complex number.

```
1.imaginary   # =>   0
```

integer?

num.integer? → true or false

Returns true if *num* is an Integer (including Fixnum and Bignum).

magnitude

num.magnitude → *int* or *float*

1.9

Returns the magnitude of *num*(the distance of *num* from the origin of the number line. See also Complex#magnitude.

```
3.magnitude      # =>   3
-3.0.magnitude   # =>   3.0
```

modulo

num.modulo(*numeric*) → *numeric*

Equivalent to *num*.divmod(*numeric*)[1].

nonzero?

num.nonzero? → *num* or nil

Returns *num* if *num* is not zero and returns nil otherwise. This behavior is useful when chaining comparisons.

```
a = %w( z Bb bB bb BB a aA Aa AA A )
b = a.sort {|a,b| (a.downcase <=> b.downcase).nonzero? || a <=> b }
b   # =>   ["A", "a", "AA", "Aa", "aA", "BB", "Bb", "bB", "bb", "z"]
```

numerator

num.numerator → *integer*

1.9

Returns the numerator of the rational representation of *num*.

```
1.numerator     # =>   1
1.5.numerator   # =>   3
num = 1.0/3
num.to_r        # =>   (6004799503160661/18014398509481984)
num.numerator   # =>   6004799503160661
```

phase

num.phase → [*magnitude*, *angle*]

1.9

Returns the phase angle of *num*. See Complex for more information. For noncomplex numbers, returns 0 if *num* is nonnegative, π otherwise.

```
123.polar   # =>   [123, 0]
```

polar

num.polar → [*magnitude*, *angle*]

1.9

Returns *num* in polar form. See Complex for more information. For noncomplex numbers, returns [*num*,0].

```
123.polar   # =>   [123, 0]
```

quo

num.quo(*numeric*) → *numeric*

1.9

Equivalent to Numeric#/ but overridden in subclasses. The intent of quo is to return the most accurate result of division (in context). Thus, 1.quo(2) will equal the rational number $\frac{1}{2}$, while 1/2 equals 0.

real

num.real → *num*

1.9

Returns the real part of *num*. Always *num* unless *num* is a complex number.

```
1.real     # =>   1
1.5.real   # =>   1.5
```

real?

num.real? → true

1.9

All the built-in numeric classes except Complex represent scalar types and hence respond true to real?.

```
1.real?              # =>   true
1.0.real?            # =>   true
Complex(1,0).real?   # =>   false
```

rect

num.rect → [*num*, 0]

1.9

Returns an array containing the real and imaginary components of *num*. See also Complex#rect.

```
1.5.rect   # =>   [1.5, 0]
```

rectangular

num.rectangular → [*num*, 0]

1.9

Synonym for Numeric#rect.

remainder

num.remainder(*numeric*) → *numeric*

If *num* and *numeric* have different signs, returns *mod*−*numeric*; otherwise, returns *mod*. In both cases, *mod* is the value *num*.modulo(*numeric*). The differences between remainder and modulo (%) are shown in Table 27.14 on page 619.

round

num.round → *int*

Rounds *num* to the nearest integer. Numeric implements this by converting *int* to a Float and invoking Float#round.

step

num.step(*end_num*, *step*) {| *i* | *block* } → *num*

Invokes *block* with the sequence of numbers starting at *num*, incremented by *step* on each call. The loop finishes when the value to be passed to the block is greater than *end_num* (if *step* is positive) or less than *end_num* (if *step* is negative). If all the arguments are integers, the loop operates using an integer counter. If any of the arguments are floating-point numbers, all are converted to floats, and the loop is executed $\lfloor n + n * \epsilon \rfloor + 1$ times, where $n = (end\_num - num)/step$. Otherwise, the loop starts at *num*, uses either the < or > operator to compare the counter against *end_num*, and increments itself using the + operator.

```
1.step(10, 2) {|i| print i, " " }
Math::E.step(Math::PI, 0.2) {|f| print f, " " }
```

produces:

```
1 3 5 7 9
2.71828182845905 2.91828182845905 3.11828182845905
```

to_c

1.9

num.to_c → *complex*

Returns *num* as a complex number.

```
123.to_c   # =>   123+0i
```

to_int

num.to_int → *int*

Invokes the child class's to_i method to convert *num* to an integer.

truncate

num.truncate → *int*

Returns *num* truncated to an integer. Numeric implements this by converting its value to a float and invoking Float#truncate.

zero?

num.zero? → true or false

Returns true if *num* has a zero value.