Home

Pages Classes Methods

Search

Table of Contents

What's Here
Querying
Comparing
Converting
Other

Show/hide navigation

Parent

Numeric

Methods

::sqrt
::try_convert
#%
#&
#*
#**
#+
#-
#-@
#/
#<
#<<
#<=
#<=>
#==
#===
#>
#>=
#>>
#[]
#^
#abs
#allbits?
#anybits?
#bit_length
#ceil
#ceildiv
#chr
#coerce
#denominator
#digits
#div
#divmod
#downto
#even?
#fdiv
#floor

class Integer

An Integer object represents an integer value.

You can create an Integer object explicitly with:

- An integer literal.

You can convert certain objects to Integers with:

- Method `Integer`.

An attempt to add a singleton method to an instance of this class causes an exception to be raised.

What's Here

First, what's elsewhere. Class Integer:

- Inherits from class Numeric.

Here, class Integer provides methods for:

- Querying
- Comparing
- Converting
- Other

Querying

- `allbits?`: Returns whether all bits in `self` are set.
- `anybits?`: Returns whether any bits in `self` are set.
- `nobits?`: Returns whether no bits in `self` are set.

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 `===`): 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

- ::sqrt : Returns the integer square root of the given value.
- ::try_convert : Returns the given value converted to an Integer.
- % (aliased as modulo): Returns `self` modulo the given value.
- #&: Returns the bitwise AND of `self` and 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.
- << : Returns the value of `self` after a leftward bit-shift.
- >> : Returns the value of `self` after a rightward bit-shift.
- [] : Returns a slice of bits from `self` .
- #^: Returns the bitwise EXCLUSIVE OR of `self` and the given value.
- ceil : Returns the smallest number greater than or equal to `self` .
- chr : Returns a 1-character string containing the character represented by the value of `self` .
- digits : Returns an array of integers representing the base-radix digits of `self` .
- div : Returns the integer result of dividing `self` by the given value.
- 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` .
- pow : Returns the modular exponentiation of `self` .
- pred : Returns the integer predecessor of `self` .
- remainder : Returns the remainder after dividing `self` by the given value.
- round : Returns `self` rounded to the nearest value with the given precision.
- succ (aliased as next): Returns the integer successor of `self` .
- to_f : Returns `self` converted to a Float .
- to_s (aliased as inspect): Returns a string containing the place-value representation of `self` in the given radix.
- truncate : Returns `self` truncated to the given precision.
- #|: Returns the bitwise OR of `self` and the given value.

Other

- `downto` : Calls the given block with each integer value from `self` down to the given value.
- `times` : Calls the given block `self` times with each integer in `(0..self-1)` .
- `upto` : Calls the given block with each integer value from `self` up to the given value.

Constants

GMP_VERSION

The version of loaded GMP.

Public Class Methods

sqrt(numeric) → integer

Returns the integer square root of the non-negative integer `n` , which is the largest non-negative integer less than or equal to the square root of `numeric` .

```
Integer.sqrt(0)       # => 0
Integer.sqrt(1)       # => 1
Integer.sqrt(24)      # => 4
Integer.sqrt(25)      # => 5
Integer.sqrt(10**400) # => 10**200
```

If `numeric` is not an Integer, it is converted to an Integer:

```
Integer.sqrt(Complex(4, 0))  # => 2
Integer.sqrt(Rational(4, 1)) # => 2
Integer.sqrt(4.0)            # => 2
Integer.sqrt(3.14159)        # => 1
```

This method is equivalent to `Math.sqrt(numeric).floor` , except that the result of the latter code may differ from the true value due to the limited precision of floating point arithmetic.

```
Integer.sqrt(10**46)    # => 100000000000000000000000
Math.sqrt(10**46).floor # => 99999999999999991611392
```

Raises an exception if `numeric` is negative.

try_convert(object) → object, integer, or nil

If `object` is an Integer object, returns `object`.

```
Integer.try_convert(1) # => 1
```

Otherwise if `object` responds to `:to_int`, calls `object.to_int` and returns the result.

```
Integer.try_convert(1.25) # => 1
```

Returns `nil` if `object` does not respond to `:to_int`

```
Integer.try_convert([]) # => nil
```

Raises an exception unless `object.to_int` returns an Integer object.

Public Instance Methods

self % other → real_number

Returns `self` modulo `other` as a real number.

For integer `n` and real number `r`, these expressions are equivalent:

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

See Numeric#divmod.

Examples:

```
10 % 2              # => 0
10 % 3              # => 1
10 % 4              # => 2

10 % -2             # => 0
10 % -3             # => -2
10 % -4             # => -2

10 % 3.0            # => 1.0
10 % Rational(3, 1) # => (1/1)
```

Also aliased as: modulo

self & other → integer

Bitwise AND; each bit in the result is 1 if both corresponding bits in `self` and `other` are 1, 0 otherwise:

```
"%04b" % (0b0101 & 0b0110) # => "0100"
```

Raises an exception if `other` is not an Integer.

Related: Integer#| (bitwise OR), Integer#^ (bitwise EXCLUSIVE OR).

`self * numeric → numeric_result`

Performs multiplication:

```
4 * 2              # => 8
4 * -2             # => -8
-4 * 2             # => -8
4 * 2.0            # => 8.0
4 * Rational(1, 3) # => (4/3)
4 * Complex(2, 0)  # => (8+0i)
```

`self ** numeric → numeric_result`

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

```
2 ** 3              # => 8
2 ** -3             # => (1/8)
-2 ** 3             # => -8
-2 ** -3            # => (-1/8)
2 ** 3.3            # => 9.849155306759329
2 ** Rational(3, 1) # => (8/1)
2 ** Complex(3, 0)  # => (8+0i)
```

`self + numeric → numeric_result`

Performs addition:

```
2 + 2              # => 4
-2 + 2             # => 0
-2 + -2            # => -4
2 + 2.0            # => 4.0
2 + Rational(2, 1) # => (4/1)
2 + Complex(2, 0)  # => (4+0i)
```

`self - numeric → numeric_result`

Performs subtraction:

```
4 - 2               # => 2
-4 - 2              # => -6
-4 - -2             # => -2
4 - 2.0             # => 2.0
4 - Rational(2, 1) # => (2/1)
4 - Complex(2, 0)  # => (2+0i)
```

-int → integer

Returns `self`, negated.

self / numeric → numeric_result

Performs division; for integer `numeric`, truncates the result to an integer:

```
 4 / 3              # => 1
 4 / -3             # => -2
 -4 / 3             # => -2
 -4 / -3            # => 1

For other +numeric+, returns non-integer result:

 4 / 3.0            # => 1.3333333333333333
 4 / Rational(3, 1) # => (4/3)
 4 / Complex(3, 0)  # => ((4/3)+0i)
```

self < other → true or false

Returns `true` if the value of `self` is less than that of `other`:

```
  1 < 0              # => false
  1 < 1              # => false
  1 < 2              # => true
  1 < 0.5            # => false
  1 < Rational(1, 2) # => false

Raises an exception if the comparison cannot be made.
```

self << count → integer

Returns `self` with bits shifted `count` positions to the left, or to the right if `count` is negative:

```
n = 0b11110000
"%08b" % (n << 1)  # => "111100000"
"%08b" % (n << 3)  # => "11110000000"
```

```
"%08b" % (n << -1) # => "01111000"
"%08b" % (n << -3) # => "00011110"
```

Related: Integer#>>.

self <= real → true or false

Returns true if the value of self is less than or equal to that of other:

```
1 <= 0               # => false
1 <= 1               # => true
1 <= 2               # => true
1 <= 0.5             # => false
1 <= Rational(1, 2) # => false
```

Raises an exception if the comparison cannot be made.

self <=> other → -1, 0, +1, or nil

Returns:

- -1, if self is less than other.
- 0, if self is equal to other.
- 1, if self is greater then other.
- nil, if self and other are incomparable.

Examples:

```
1 <=> 2              # => -1
1 <=> 1              # => 0
1 <=> 0              # => 1
1 <=> 'foo'          # => nil

1 <=> 1.0            # => 0
1 <=> Rational(1, 1) # => 0
1 <=> Complex(1, 0)  # => 0
```

This method is the basis for comparisons in module Comparable.

self == other → true or false

Returns true if self is numerically equal to other; false otherwise.

```
1 == 2     #=> false
1 == 1.0   #=> true
```

Related: Integer#eql? (requires other to be an Integer).

Alias for: ===

=== == other -> true or false

Returns true if self is numerically equal to other; false otherwise.

```
1 == 2      #=> false
1 == 1.0    #=> true
```

Related: Integer#eql? (requires other to be an Integer).

Also aliased as: ==

self > other → true or false

Returns true if the value of self is greater than that of other:

```
  1 > 0              # => true
  1 > 1              # => false
  1 > 2              # => false
  1 > 0.5            # => true
  1 > Rational(1, 2) # => true

Raises an exception if the comparison cannot be made.
```

self >= real → true or false

Returns true if the value of self is greater than or equal to that of other:

```
1 >= 0              # => true
1 >= 1              # => true
1 >= 2              # => false
1 >= 0.5            # => true
1 >= Rational(1, 2) # => true
```

Raises an exception if the comparison cannot be made.

self >> count → integer

Returns self with bits shifted count positions to the right, or to the left if count is negative:

```
n = 0b11110000
"%08b" % (n >> 1)  # => "01111000"
"%08b" % (n >> 3)  # => "00011110"
```

```
"%08b" % (n >> -1) # => "111100000"
"%08b" % (n >> -3) # => "111100000000"
```

Related: Integer#<<.

self[offset] → 0 or 1
self[offset, size] → integer
self[range] → integer

Returns a slice of bits from `self`.

With argument `offset`, returns the bit at the given offset, where offset 0 refers to the least significant bit:

```
n = 0b10 # => 2
n[0]     # => 0
n[1]     # => 1
n[2]     # => 0
n[3]     # => 0
```

In principle, `n[i]` is equivalent to `(n >> i) & 1`. Thus, negative index always returns zero:

```
255[-1] # => 0
```

With arguments `offset` and `size`, returns `size` bits from `self`, beginning at `offset` and including bits of greater significance:

```
n = 0b111000       # => 56
"%010b" % n[0, 10] # => "0000111000"
"%010b" % n[4, 10] # => "0000000011"
```

With argument `range`, returns `range.size` bits from `self`, beginning at `range.begin` and including bits of greater significance:

```
n = 0b111000      # => 56
"%010b" % n[0..9] # => "0000111000"
"%010b" % n[4..9] # => "0000000011"
```

Raises an exception if the slice cannot be constructed.

self ^ other → integer

Bitwise EXCLUSIVE OR; each bit in the result is 1 if the corresponding bits in `self` and `other` are different, 0 otherwise:

```
"%04b" % (0b0101 ^ 0b0110) # => "0011"
```

Raises an exception if `other` is not an Integer.

Related: Integer#& (bitwise AND), Integer#| (bitwise OR).

`abs → integer`

Returns the absolute value of `self`.

```
(-12345).abs # => 12345
-12345.abs   # => 12345
12345.abs    # => 12345
```

Also aliased as: magnitude

`allbits?(mask) → true or false`

Returns `true` if all bits that are set (=1) in `mask` are also set in `self`; returns `false` otherwise.

Example values:

```
0b1010101  self
0b1010100  mask
0b1010100  self & mask
     true  self.allbits?(mask)

0b1010100  self
0b1010101  mask
0b1010100  self & mask
    false  self.allbits?(mask)
```

Related: Integer#anybits?, Integer#nobits?.

`anybits?(mask) → true or false`

Returns `true` if any bit that is set (=1) in `mask` is also set in `self`; returns `false` otherwise.

Example values:

```
0b10000010  self
0b11111111  mask
0b10000010  self & mask
      true  self.anybits?(mask)

0b00000000  self
0b11111111  mask
```

```
0b00000000  self & mask
     false  self.anybits?(mask)
```

Related: Integer#allbits?, Integer#nobits?.

bit_length → integer

Returns the number of bits of the value of `self`, which is the bit position of the highest-order bit that is different from the sign bit (where the least significant bit has bit position 1). If there is no such bit (zero or minus one), returns zero.

This method returns `ceil(log2(self < 0 ? -self : self + 1))`.

```
(-2**1000-1).bit_length   # => 1001
(-2**1000).bit_length     # => 1000
(-2**1000+1).bit_length   # => 1000
(-2**12-1).bit_length     # => 13
(-2**12).bit_length       # => 12
(-2**12+1).bit_length     # => 12
-0x101.bit_length         # => 9
-0x100.bit_length         # => 8
-0xff.bit_length          # => 8
-2.bit_length             # => 1
-1.bit_length             # => 0
0.bit_length              # => 0
1.bit_length              # => 1
0xff.bit_length           # => 8
0x100.bit_length          # => 9
(2**12-1).bit_length      # => 12
(2**12).bit_length        # => 13
(2**12+1).bit_length      # => 13
(2**1000-1).bit_length    # => 1000
(2**1000).bit_length      # => 1001
(2**1000+1).bit_length    # => 1001
```

For Integer *n*, this method can be used to detect overflow in Array#pack:

```
if n.bit_length < 32
  [n].pack('l') # No overflow.
else
  raise 'Overflow'
end
```

ceil(ndigits = 0) → integer

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

When the precision is negative, the returned value is an integer with at least `ndigits.abs` trailing zeros:

```
555.ceil(-1)  # => 560
555.ceil(-2)  # => 600
-555.ceil(-2) # => -500
555.ceil(-3)  # => 1000
```

Returns `self` when `ndigits` is zero or positive.

```
555.ceil     # => 555
555.ceil(50) # => 555
```

Related: Integer#floor.

ceildiv(numeric) → integer

Returns the result of division `self` by `numeric`. rounded up to the nearest integer.

```
3.ceildiv(3)   # => 1
4.ceildiv(3)   # => 2

4.ceildiv(-3)  # => -1
-4.ceildiv(3)  # => -1
-4.ceildiv(-3) # => 2

3.ceildiv(1.2) # => 3
```

chr → string
chr(encoding) → string

Returns a 1-character string containing the character represented by the value of `self`, according to the given `encoding`.

```
65.chr                    # => "A"
0.chr                     # => "\x00"
255.chr                   # => "\xFF"
string = 255.chr(Encoding::UTF_8)
string.encoding           # => Encoding::UTF_8
```

Raises an exception if `self` is negative.

Related: Integer#ord.

coerce(numeric) → array

Returns an array with both a `numeric` and a `int` represented as Integer objects or Float objects.

This is achieved by converting `numeric` to an Integer or a Float.

A TypeError is raised if the numeric is not an Integer or a Float type.

```
(0x3FFFFFFFFFFFFFFF+1).coerce(42)   #=> [42, 4611686018427387904]
```

denominator → 1

Returns 1.

digits(base = 10) → array_of_integers

Returns an array of integers representing the base-radix digits of self; the first element of the array represents the least significant digit:

```
12345.digits      # => [5, 4, 3, 2, 1]
12345.digits(7)   # => [4, 6, 6, 0, 5]
12345.digits(100) # => [45, 23, 1]
```

Raises an exception if self is negative or base is less than 2.

div(numeric) → integer

Performs integer division; returns the integer result of dividing self by numeric:

```
  4.div(3)      # => 1
  4.div(-3)      # => -2
  -4.div(3)      # => -2
  -4.div(-3)      # => 1
  4.div(3.0)      # => 1
  4.div(Rational(3, 1))      # => 1

Raises an exception if +numeric+ does not have method +div+.
```

divmod(other) → array

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

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

Examples:

```
11.divmod(4)               # => [2, 3]
11.divmod(-4)              # => [-3, -1]
-11.divmod(4)              # => [-3, 1]
-11.divmod(-4)             # => [2, -3]
```

```
12.divmod(4)               # => [3, 0]
12.divmod(-4)              # => [-3, 0]
-12.divmod(4)              # => [-3, 0]
-12.divmod(-4)             # => [3, 0]

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

downto(limit) {|i| ... } → self
downto(limit) → enumerator

Calls the given block with each integer value from `self` down to `limit`; returns `self`:

```
a = []
10.downto(5) {|i| a << i }              # => 10
a                                       # => [10, 9, 8, 7, 6, 5]
a = []
0.downto(-5) {|i| a << i }              # => 0
a                                       # => [0, -1, -2, -3, -4, -5]
4.downto(5) {|i| fail 'Cannot happen' } # => 4
```

With no block given, returns an Enumerator.

even? → true or false

Returns `true` if `self` is an even number, `false` otherwise.

fdiv(numeric) → float

Returns the Float result of dividing `self` by `numeric`:

```
4.fdiv(2)      # => 2.0
4.fdiv(-2)     # => -2.0
-4.fdiv(2)     # => -2.0
4.fdiv(2.0)    # => 2.0
4.fdiv(Rational(3, 4))      # => 5.333333333333333
```

Raises an exception if `numeric` cannot be converted to a Float.

floor(ndigits = 0) → integer

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

When `ndigits` is negative, the returned value has at least `ndigits.abs` trailing zeros:

```
555.floor(-1)  # => 550
555.floor(-2)  # => 500
-555.floor(-2) # => -600
555.floor(-3)  # => 0
```

Returns `self` when `ndigits` is zero or positive.

```
555.floor     # => 555
555.floor(50) # => 555
```

Related: [Integer#ceil](#).

gcd(other_int) → integer

Returns the greatest common divisor of the two integers. The result is always positive. 0.gcd(x) and x.gcd(0) return x.abs.

```
36.gcd(60)                  #=> 12
2.gcd(2)                    #=> 2
3.gcd(-7)                   #=> 1
((1<<31)-1).gcd((1<<61)-1)  #=> 1
```

gcdlcm(other_int) → array

Returns an array with the greatest common divisor and the least common multiple of the two integers, [gcd, lcm].

```
36.gcdlcm(60)                  #=> [12, 180]
2.gcdlcm(2)                    #=> [2, 2]
3.gcdlcm(-7)                   #=> [1, 21]
((1<<31)-1).gcdlcm((1<<61)-1)  #=> [1, 4951760154835678088235319297]
```

inspect(*args)

Returns a string containing the place-value representation of `self` in radix `base` (in 2..36).

```
12345.to_s               # => "12345"
12345.to_s(2)            # => "11000000111001"
12345.to_s(8)            # => "30071"
12345.to_s(10)           # => "12345"
12345.to_s(16)           # => "3039"
12345.to_s(36)           # => "9ix"
78546939656932.to_s(36)  # => "rubyrules"
```

Raises an exception if `base` is out of range.

Alias for: to_s

integer? → true

Since `self` is already an Integer, always returns `true`.

lcm(other_int) → integer

Returns the least common multiple of the two integers. The result is always positive. 0.lcm(x) and x.lcm(0) return zero.

```
36.lcm(60)                    #=> 180
2.lcm(2)                      #=> 2
3.lcm(-7)                     #=> 21
((1<<31)-1).lcm((1<<61)-1)    #=> 4951760154835678088235319297
```

magnitude()

Alias for: abs

modulo(p1)

Returns `self` modulo `other` as a real number.

For integer `n` and real number `r`, these expressions are equivalent:

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

See `Numeric#divmod`.

Examples:

```
10 % 2              # => 0
10 % 3              # => 1
10 % 4              # => 2

10 % -2             # => 0
10 % -3             # => -2
10 % -4             # => -2

10 % 3.0            # => 1.0
10 % Rational(3, 1) # => (1/1)
```

Alias for: %

next()

Returns the successor integer of `self` (equivalent to `self + 1`):

```
1.succ  #=> 2
-1.succ #=> 0
```

Related: `Integer#pred` (predecessor value).

Alias for: succ

nobits?(mask) → true or false

Returns `true` if no bit that is set (=1) in `mask` is also set in `self`; returns `false` otherwise.

Example values:

```
0b11110000  self
0b00001111  mask
0b00000000  self & mask
      true  self.nobits?(mask)

0b00000001  self
0b11111111  mask
0b00000001  self & mask
     false  self.nobits?(mask)
```

Related: `Integer#allbits?`, `Integer#anybits?`.

numerator → self

Returns `self`.

odd? → true or false

Returns `true` if `self` is an odd number, `false` otherwise.

ord → self

Returns `self`; intended for compatibility to character literals in Ruby 1.9.

pow(numeric) → numeric
pow(integer, integer) → integer

Returns (modular) exponentiation as:

```
a.pow(b)     #=> same as a**b
a.pow(b, m)  #=> same as (a**b) % m, but avoids huge temporary values
```

pred → next_integer

Returns the predecessor of `self` (equivalent to `self - 1`):

```
1.pred  #=> 0
-1.pred #=> -2
```

Related: [Integer#succ](#) (successor value).

rationalize([eps]) → rational

Returns the value as a rational. The optional argument `eps` is always ignored.

remainder(other) → real_number

Returns the remainder after dividing `self` by `other`.

Examples:

```
11.remainder(4)              # => 3
11.remainder(-4)             # => 3
-11.remainder(4)             # => -3
-11.remainder(-4)            # => -3

12.remainder(4)              # => 0
12.remainder(-4)             # => 0
-12.remainder(4)             # => 0
-12.remainder(-4)            # => 0

13.remainder(4.0)            # => 1.0
13.remainder(Rational(4, 1)) # => (1/1)
```

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

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

When `ndigits` is negative, the returned value has at least `ndigits.abs` trailing zeros:

```
555.round(-1)      # => 560
555.round(-2)      # => 600
555.round(-3)      # => 1000
```

```
-555.round(-2)      # => -600
555.round(-4)       # => 0
```

Returns `self` when `ndigits` is zero or positive.

```
555.round      # => 555
555.round(1)   # => 555
555.round(50) # => 555
```

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:

  ```
  25.round(-1, half: :up)      # => 30
  (-25).round(-1, half: :up)   # => -30
  ```

- `:down`: round toward zero:

  ```
  25.round(-1, half: :down)    # => 20
  (-25).round(-1, half: :down) # => -20
  ```

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

  ```
  25.round(-1, half: :even)    # => 20
  15.round(-1, half: :even)    # => 20
  (-25).round(-1, half: :even) # => -20
  ```

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

Related: [Integer#truncate](#).

`size → integer`

Returns the number of bytes in the machine representation of `self`; the value is system-dependent:

```
1.size             # => 8
-1.size            # => 8
2147483647.size    # => 8
(256**10 - 1).size # => 10
(256**20 - 1).size # => 20
(256**40 - 1).size # => 40
```

`succ → next_integer`

Returns the successor integer of `self` (equivalent to `self + 1`):

```
1.succ  #=> 2
-1.succ #=> 0
```

Related: Integer#pred (predecessor value).

Also aliased as: next

times {|i| ... } → self
times → enumerator

Calls the given block `self` times with each integer in `(0..self-1)`:

```
a = []
5.times {|i| a.push(i) } # => 5
a                        # => [0, 1, 2, 3, 4]
```

With no block given, returns an `Enumerator`.

to_f → float

Converts `self` to a Float:

```
1.to_f  # => 1.0
-1.to_f # => -1.0
```

If the value of `self` does not fit in a `Float`, the result is infinity:

```
(10**400).to_f  # => Infinity
(-10**400).to_f # => -Infinity
```

to_i → self

Returns `self` (which is already an Integer).

to_int → self

Returns `self` (which is already an Integer).

to_r → rational

Returns the value as a rational.

```
1.to_r         #=> (1/1)
(1<<64).to_r   #=> (18446744073709551616/1)
```

to_s(base = 10) → string

Returns a string containing the place-value representation of `self` in radix `base` (in 2..36).

```
12345.to_s               # => "12345"
12345.to_s(2)            # => "11000000111001"
12345.to_s(8)            # => "30071"
12345.to_s(10)           # => "12345"
12345.to_s(16)           # => "3039"
12345.to_s(36)           # => "9ix"
78546939656932.to_s(36)  # => "rubyrules"
```

Raises an exception if `base` is out of range.

Also aliased as: inspect

truncate(ndigits = 0) → integer

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

When `ndigits` is negative, the returned value has at least `ndigits.abs` trailing zeros:

```
555.truncate(-1)  # => 550
555.truncate(-2)  # => 500
-555.truncate(-2) # => -500
```

Returns `self` when `ndigits` is zero or positive.

```
555.truncate     # => 555
555.truncate(50) # => 555
```

Related: `Integer#round`.

upto(limit) {|i| ... } → self
upto(limit) → enumerator

Calls the given block with each integer value from `self` up to `limit`; returns `self`:

```
a = []
5.upto(10) {|i| a << i }             # => 5
a                                    # => [5, 6, 7, 8, 9, 10]
a = []
```

```
-5.upto(0) {|i| a << i }              # => -5
a                                     # => [-5, -4, -3, -2, -1, 0]
5.upto(4) {|i| fail 'Cannot happen' } # => 5
```

With no block given, returns an Enumerator.

zero? → true or false

Returns `true` if `self` has a zero value, `false` otherwise.

self | other → integer

Bitwise OR; each bit in the result is 1 if either corresponding bit in `self` or `other` is 1, 0 otherwise:

```
"%04b" % (0b0101 | 0b0110) # => "0111"
```

Raises an exception if `other` is not an Integer.

Related: Integer#& (bitwise AND), Integer#^ (bitwise EXCLUSIVE OR).

~int → integer

One's complement: returns the value of `self` with each bit inverted.

Because an integer value is conceptually of infinite length, the result acts as if it had an infinite number of one bits to the left. In hex representations, this is displayed as two periods to the left of the digits:

```
sprintf("%X", ~0x1122334455)    # => "..FEEDDCCBBAA"
```