Home (http://yoururlhere.com/)

Core 2.4.0 (http://yoururlhere.com/core-2.4.0)

Std-lib 2.4.0 (http://yoururlhere.com/stdlib-2.4.0)

Downloads (http://yoururlhere.com/downloads)

Home (./index.html) Classes (./index.html#classes) Methods (./index.html#methods)

In Files

- enum.c
- enumerator.c

Methods

Files

grammar.en.rdoc (./_lib/racc/rdoc

Enumerable

The `Enumerable` mixin provides collection classes with several traversal and searching methods, and with the ability to sort. The class must provide a method `each`, which yields successive members of the collection. If `Enumerable#max`, `#min`, or `#sort` is used, the objects in the collection must also implement a meaningful `<=>` operator, as these methods rely on an ordering between members of the collection.

Public Instance Methods

### all? [{ |obj| block }] → true or false

Passes each element of the collection to the given block. The method returns `true` if the block never returns `false` or `nil`. If the block is not given, Ruby adds an implicit block of `{ |obj| obj }` which will cause all? (Enumerable.html#method-i-all-3F) to return `true` when none of the collection members are `false` or `nil`.

```
%w[ant bear cat].all? { |word| word.length >= 3 } #=> true
%w[ant bear cat].all? { |word| word.length >= 4 } #=> false
[nil, true, 99].all?                              #=> false
```

### any? [{ |obj| block }] → true or false

Passes each element of the collection to the given block. The method returns `true` if the block ever returns a value other than `false` or `nil`. If the block is not given, Ruby adds an implicit block of `{ |obj| obj }` that will cause any? (Enumerable.html#method-i-any-3F) to return `true` if at least one of the collection members is not `false` or `nil`.

```
%w[ant bear cat].any? { |word| word.length >= 3 } #=> true
%w[ant bear cat].any? { |word| word.length >= 4 } #=> true
[nil, true, 99].any?                              #=> true
```

### chunk { |elt| ... } → an_enumerator

Enumerates over the items, chunking them together based on the return value of the block.

Consecutive elements which return the same block value are chunked together.

For example, consecutive even numbers and odd numbers can be chunked as follows.

```
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5].chunk { |n|
  n.even?
}.each { |even, ary|
  p [even, ary]
}
#=> [false, [3, 1]]
#   [true, [4]]
#   [false, [1, 5, 9]]
#   [true, [2, 6]]
#   [false, [5, 3, 5]]
```

This method is especially useful for sorted series of elements. The following example counts words for each initial letter.

```
open("/usr/share/dict/words", "r:iso-8859-1") { |f|
  f.chunk { |line| line.ord }.each { |ch, lines| p [ch.chr, lines.length] }
}
#=> ["\n", 1]
#   ["A", 1327]
#   ["B", 1372]
#   ["C", 1507]
#   ["D", 791]
```

/grammar_en_rdoc.html)
test.ja.rdoc (./_test/rdoc/test_ja_rdoc.html)
contributing.rdoc (./doc/contributing_rdoc.html)
contributors.rdoc (./doc/contributors_rdoc.html)
dtrace_probes.rdoc (./doc/dtrace_probes_rdoc.html)
extension.ja.rdoc (./doc/extension_ja_rdoc.html)
extension.rdoc (./doc/extension_rdoc.html)
globals.rdoc (./doc/globals_rdoc.html)
keywords.rdoc (./doc/keywords_rdoc.html)
maintainers.rdoc (./doc/maintainers_rdoc.html)
marshal.rdoc (./doc/marshal_rdoc.html)
regexp.rdoc (./doc/regexp_rdoc.html)
security.rdoc (./doc/security_rdoc.html)
standard_library.rdoc (./doc
ment_rdoc.html)
tax
ons_rdoc.html)
c.html)
rdoc.html)
ntax
ence_rdoc.html)
ments_rdoc.html)
ME_rdoc.html)

Class/Module Index

ARGF (./ARGF.html)
ArgumentError (./ArgumentError.html)
Array (./Array.html)
BasicObject (./BasicObject.html)
Binding (./Binding.html)
Class (./Class.html)
ClosedQueueError (./ClosedQueueError.html)
Comparable (./Comparable.html)
Complex (./Complex.html)
Complex::compatible (./Complex/compatible.html)
ConditionVariable (./ConditionVariable.html)
Continuation (./Continuation.html)
Data (./Data.html)
Dir (./Dir.html)
ENV (./ENV.html)
EOFError (./EOFError.html)
Encoding (./Encoding.html)
Encoding::CompatibilityError (./Encoding/Compatibility
Encoding::Converter (./Encoding/Converter.html)
Encoding::ConverterNotFoundError (./Encoding/Conve
Encoding::InvalidByteSequenceError (./Encoding/Inval
Encoding::UndefinedConversionError (./Encoding/Und
EncodingError (./EncodingError.html)
Enumerable (./Enumerable.html)
Enumerator (./Enumerator.html)
Enumerator::Generator (./Enumerator/Generator.html)
Enumerator::Lazy (./Enumerator/Lazy.html)
Enumerator::Yielder (./Enumerator/Yielder.html)
Errno (./Errno.html)
Exception (./Exception.html)
FalseClass (./FalseClass.html)
Fiber (./Fiber.html)
FiberError (./FiberError.html)
File (./File.html)
File::Constants (./File/Constants.html)
File::Stat (./File/Stat.html)
FileTest (./FileTest.html)
Float (./Float.html)
FloatDomainError (./FloatDomainError.html)
GC (./GC.html)
GC::Profiler (./GC/Profiler.html)
Hash (./Hash.html)
IO (./IO.html)
IO::EAGAINWaitReadable (./IO/EAGAINWaitReadable
IO::EAGAINWaitWritable (./IO/EAGAINWaitWritable.h
IO::EINPROGRESSWaitReadable (./IO/EINPROGRES
IO::EINPROGRESSWaitWritable (./IO/EINPROGRESS
IO::EWOULDBLOCKWaitReadable (./IO/EWOULDB
IO::EWOULDBLOCKWaitWritable (./IO/EWOULDBI
IO::WaitReadable (./IO/WaitReadable.html)
IO::WaitWritable (./IO/WaitWritable.html)
IOError (./IOError.html)
IndexError (./IndexError.html)
Integer (./Integer.html)
Interrupt (./Interrupt.html)
Kernel (./Kernel.html)
KeyError (./KeyError.html)
LoadError (./LoadError.html)

```
#   ...
```

The following key values have special meaning:

- `nil` and `:_separator` specifies that the elements should be dropped.
- `:_alone` specifies that the element should be chunked by itself.

Any other symbols that begin with an underscore will raise an error:

```
items.chunk { |item| :_underscore }
#=> RuntimeError: symbols beginning with an underscore are reserved
```

`nil` and `:_separator` can be used to ignore some elements.

For example, the sequence of hyphens in svn log can be eliminated as follows:

```
sep = "-"*72 + "\n"
IO.popen("svn log README") { |f|
  f.chunk { |line|
    line != sep || nil
  }.each { |_, lines|
    pp lines
  }
}
#=> ["r20018 | knu | 2008-10-29 13:20:42 +0900 (Wed, 29 Oct 2008) | 2 lines\n",
#    "\n",
#    "* README, README.ja: Update the portability section.\n",
#    "\n"]
#   ["r16725 | knu | 2008-05-31 23:34:23 +0900 (Sat, 31 May 2008) | 2 lines\n",
#    "\n",
#    "* README, README.ja: Add a note about default C flags.\n",
#    "\n"]
#   ...
```

Paragraphs separated by empty lines can be parsed as follows:

```
File.foreach("README").chunk { |line|
  /\A\s*\z/ !~ line || nil
}.each { |_, lines|
  pp lines
}
```

`:_alone` can be used to force items into their own chunk. For example, you can put lines that contain a URL by themselves, and chunk the rest of the lines together, like this:

```
pattern = /http/
open(filename) { |f|
  f.chunk { |line| line =~ pattern ? :_alone : true }.each { |key, lines|
    pp lines
  }
}
```

If no block is given, an enumerator to `chunk` is returned instead.

chunk_while {|elt_before, elt_after| bool } → an_enumerator

Creates an enumerator for each chunked elements. The beginnings of chunks are defined by the block.

This method split each chunk using adjacent elements, *elt_before* and *elt_after*, in the receiver enumerator. This method split chunks between *elt_before* and *elt_after* where the block returns `false`.

The block is called the length of the receiver enumerator minus one.

The result enumerator yields the chunked elements as an array. So `each` method can be called as follows:

```
enum.chunk_while { |elt_before, elt_after| bool }.each { |ary| ... }
```

Other methods of the Enumerator (Enumerator.html) class and Enumerable (Enumerable.html) module, such as `to_a`, `map`, etc., are also usable.

For example, one-by-one increasing subsequence can be chunked as follows:

```
a = [1,2,4,9,10,11,12,15,16,19,20,21]
b = a.chunk_while {|i, j| i+1 == j }
p b.to_a #=> [[1, 2], [4], [9, 10, 11, 12], [15, 16], [19, 20, 21]]
c = b.map {|a| a.length < 3 ? a : "#{a.first}-#{a.last}" }
p c #=> [[1, 2], [4], "9-12", [15, 16], "19-21"]
```

LocalJumpError (./LocalJumpError.html)
Marshal (./Marshal.html)
MatchData (./MatchData.html)
Math (./Math.html)
Math::DomainError (./Math/DomainError.html)
Method (./Method.html)
Module (./Module.html)
NameError (./NameError.html)
NilClass (./NilClass.html)
NoMemoryError (./NoMemoryError.html)
NoMethodError (./NoMethodError.html)
NotImplementedError (./NotImplementedError.html)
Numeric (./Numeric.html)
Object (./Object.html)
ace/WeakMap.html)
)
l)
atter.html)
patible.html)
ubyVM/InstructionS
SecurityError (./SecurityError.html)
Signal (./Signal.html)
SignalException (./SignalException.html)
SizedQueue (./SizedQueue.html)
StandardError (./StandardError.html)
StopIteration (./StopIteration.html)
String (./String.html)
Struct (./Struct.html)
Symbol (./Symbol.html)
SyntaxError (./SyntaxError.html)
SystemCallError (./SystemCallError.html)
SystemExit (./SystemExit.html)
SystemStackError (./SystemStackError.html)
Thread (./Thread.html)
Thread::Backtrace::Location (./Thread/Backtrace/Locatic
Thread::Mutex (./Thread/Mutex.html)
ThreadError (./ThreadError.html)
ThreadGroup (./ThreadGroup.html)
Time (./Time.html)
TracePoint (./TracePoint.html)
TrueClass (./TrueClass.html)
TypeError (./TypeError.html)
UnboundMethod (./UnboundMethod.html)
UncaughtThrowError (./UncaughtThrowError.html)
Warning (./Warning.html)
ZeroDivisionError (./ZeroDivisionError.html)
fatal (./fatal.html)

```
d = c.join(",")

p d #=> "1,2,4,9-12,15,16,19-21"
```

Increasing (non-decreasing) subsequence can be chunked as follows:

```
a = [0, 9, 2, 2, 3, 2, 7, 5, 9, 5]

p a.chunk_while {|i, j| i <= j }.to_a

#=> [[0, 9], [2, 2, 3], [2, 7], [5, 9], [5]]
```

Adjacent evens and odds can be chunked as follows: (Enumerable#chunk is another way to do it.)

```
a = [7, 5, 9, 2, 0, 7, 9, 4, 2, 0]

p a.chunk_while {|i, j| i.even? == j.even? }.to_a

#=> [[7, 5, 9], [2, 0], [7, 9], [4, 2, 0]]
```

#slice_when (Enumerable.html#method-i-slice_when) does the same, except splitting when the block returns `true` instead of `false`.

collect { |obj| block } → array
collect → an_enumerator

Returns a new array with the results of running *block* once for every element in *enum*.

If no block is given, an enumerator is returned instead.

```
(1..4).map { |i| i*i }      #=> [1, 4, 9, 16]

(1..4).collect { "cat"  }   #=> ["cat", "cat", "cat", "cat"]
```

collect_concat { |obj| block } → array
collect_concat → an_enumerator

Returns a new array with the concatenated results of running *block* once for every element in *enum*.

If no block is given, an enumerator is returned instead.

```
[1, 2, 3, 4].flat_map { |e| [e, -e] } #=> [1, -1, 2, -2, 3, -3, 4, -4]

[[1, 2], [3, 4]].flat_map { |e| e + [100] } #=> [1, 2, 100, 3, 4, 100]
```

count → int
count(item) → int
count { |obj| block } → int

Returns the number of items in `enum` through enumeration. If an argument is given, the number of items in `enum` that are equal to `item` are counted. If a block is given, it counts the number of elements yielding a true value.

```
ary = [1, 2, 4, 2]

ary.count               #=> 4

ary.count(2)            #=> 2

ary.count{ |x| x%2==0 } #=> 3
```

cycle(n=nil) { |obj| block } → nil
cycle(n=nil) → an_enumerator

Calls *block* for each element of *enum* repeatedly *n* times or forever if none or `nil` is given. If a non-positive number is given or the collection is empty, does nothing. Returns `nil` if the loop has finished without getting interrupted.

#cycle (Enumerable.html#method-i-cycle) saves elements in an internal array so changes to *enum* after the first pass have no effect.

If no block is given, an enumerator is returned instead.

```
a = ["a", "b", "c"]

a.cycle { |x| puts x }  # print, a, b, c, a, b, c,.. forever.

a.cycle(2) { |x| puts x }  # print, a, b, c, a, b, c.
```

detect(ifnone = nil) { |obj| block } → obj or nil
detect(ifnone = nil) → an_enumerator

Passes each entry in *enum* to *block*. Returns the first for which *block* is not false. If no object matches, calls *ifnone* and returns its result when it is specified, or returns `nil` otherwise.

If no block is given, an enumerator is returned instead.

```
(1..100).detect  => #<Enumerator: 1..100:detect>

(1..100).find    => #<Enumerator: 1..100:find>


(1..10).detect   { |i| i % 5 == 0 and i % 7 == 0 }   #=> nil

(1..10).find     { |i| i % 5 == 0 and i % 7 == 0 }   #=> nil

(1..100).detect  { |i| i % 5 == 0 and i % 7 == 0 }   #=> 35

(1..100).find    { |i| i % 5 == 0 and i % 7 == 0 }   #=> 35
```

drop(n) → array

Drops first n elements from *enum*, and returns rest elements in an array.

```
a = [1, 2, 3, 4, 5, 0]
a.drop(3)             #=> [4, 5, 0]
```

drop_while { |obj| block } → array
drop_while → an_enumerator

Drops elements up to, but not including, the first element for which the block returns `nil` or `false` and returns an array containing the remaining elements.

If no block is given, an enumerator is returned instead.

```
a = [1, 2, 3, 4, 5, 0]
a.drop_while { |i| i < 3 }   #=> [3, 4, 5, 0]
```

each_cons(n) { ... } → nil
each_cons(n) → an_enumerator

Iterates the given block for each array of consecutive <n> elements. If no block is given, returns an enumerator.

e.g.:

```
(1..10).each_cons(3) { |a| p a }
# outputs below
[1, 2, 3]
[2, 3, 4]
[3, 4, 5]
[4, 5, 6]
[5, 6, 7]
[6, 7, 8]
[7, 8, 9]
[8, 9, 10]
```

each_entry { |obj| block } → enum
each_entry → an_enumerator

Calls *block* once for each element in `self`, passing that element as a parameter, converting multiple values from yield to an array.

If no block is given, an enumerator is returned instead.

```
class Foo
  include Enumerable
  def each
    yield 1
    yield 1, 2
    yield
  end
end
Foo.new.each_entry{ |o| p o }
```

produces:

```
1
[1, 2]
nil
```

each_slice(n) { ... } → nil
each_slice(n) → an_enumerator

Iterates the given block for each slice of <n> elements. If no block is given, returns an enumerator.

```
(1..10).each_slice(3) { |a| p a }
# outputs below
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
[10]
```

each_with_index(*args) { |obj, i| block } → enum
each_with_index(*args) → an_enumerator

Calls *block* with two arguments, the item and its index, for each item in *enum*. Given arguments are passed through to each().

If no block is given, an enumerator is returned instead.

```
hash = Hash.new
```

```
%w(cat dog wombat).each_with_index { |item, index|
  hash[item] = index
}
hash   #=> {"cat"=>0, "dog"=>1, "wombat"=>2}
```

each_with_object(obj) { |(*args), memo_obj| ... } → obj
each_with_object(obj) → an_enumerator

Iterates the given block for each element with an arbitrary object given, and returns the initially given object.

If no block is given, returns an enumerator.

```
evens = (1..10).each_with_object([]) { |i, a| a << i*2 }
#=> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

entries(*args) → array

Returns an array containing the items in *enum*.

```
(1..7).to_a                       #=> [1, 2, 3, 4, 5, 6, 7]
{ 'a'=>1, 'b'=>2, 'c'=>3 }.to_a   #=> [["a", 1], ["b", 2], ["c", 3]]

require 'prime'
Prime.entries 10                  #=> [2, 3, 5, 7]
```

find(ifnone = nil) { |obj| block } → obj or nil
find(ifnone = nil) → an_enumerator

Passes each entry in *enum* to *block*. Returns the first for which *block* is not false. If no object matches, calls *ifnone* and returns its result when it is specified, or returns `nil` otherwise.

If no block is given, an enumerator is returned instead.

```
(1..100).detect  => #<Enumerator: 1..100:detect>
(1..100).find    => #<Enumerator: 1..100:find>

(1..10).detect   { |i| i % 5 == 0 and i % 7 == 0 }   #=> nil
(1..10).find     { |i| i % 5 == 0 and i % 7 == 0 }   #=> nil
(1..100).detect  { |i| i % 5 == 0 and i % 7 == 0 }   #=> 35
(1..100).find    { |i| i % 5 == 0 and i % 7 == 0 }   #=> 35
```

find_all { |obj| block } → array
find_all → an_enumerator

Returns an array containing all elements of `enum` for which the given `block` returns a true value.

If no block is given, an Enumerator (Enumerator.html) is returned instead.

```
(1..10).find_all { |i|  i % 3 == 0 }   #=> [3, 6, 9]

[1,2,3,4,5].select { |num|  num.even?  }   #=> [2, 4]
```

See also #reject (Enumerable.html#method-i-reject).

find_index(value) → int or nil
find_index { |obj| block } → int or nil
find_index → an_enumerator

Compares each entry in *enum* with *value* or passes to *block*. Returns the index for the first for which the evaluated value is non-false. If no object matches, returns `nil`

If neither block nor argument is given, an enumerator is returned instead.

```
(1..10).find_index  { |i| i % 5 == 0 and i % 7 == 0 }  #=> nil
(1..100).find_index { |i| i % 5 == 0 and i % 7 == 0 }  #=> 34
(1..100).find_index(50)                                #=> 49
```

first → obj or nil
first(n) → an_array

Returns the first element, or the first `n` elements, of the enumerable. If the enumerable is empty, the first form returns `nil`, and the second form returns an empty array.

```
%w[foo bar baz].first     #=> "foo"
%w[foo bar baz].first(2)  #=> ["foo", "bar"]
%w[foo bar baz].first(10) #=> ["foo", "bar", "baz"]
[].first                  #=> nil
[].first(10)              #=> []
```

flat_map { |obj| block } → array

flat_map → an_enumerator

Returns a new array with the concatenated results of running *block* once for every element in *enum*.

If no block is given, an enumerator is returned instead.

```
[1, 2, 3, 4].flat_map { |e| [e, -e] } #=> [1, -1, 2, -2, 3, -3, 4, -4]
[[1, 2], [3, 4]].flat_map { |e| e + [100] } #=> [1, 2, 100, 3, 4, 100]
```

grep(pattern) → array

grep(pattern) { |obj| block } → array

Returns an array of every element in *enum* for which `Pattern === element`. If the optional *block* is supplied, each matching element is passed to it, and the block's result is stored in the output array.

```
(1..100).grep 38..44   #=> [38, 39, 40, 41, 42, 43, 44]
c = IO.constants
c.grep(/SEEK/)         #=> [:SEEK_SET, :SEEK_CUR, :SEEK_END]
res = c.grep(/SEEK/) { |v| IO.const_get(v) }
res                    #=> [0, 1, 2]
```

grep_v(pattern) → array

grep_v(pattern) { |obj| block } → array

Inverted version of #grep (Enumerable.html#method-i-grep). Returns an array of every element in *enum* for which not `Pattern === element`.

```
(1..10).grep_v 2..5   #=> [1, 6, 7, 8, 9, 10]
res =(1..10).grep_v(2..5) { |v| v * 2 }
res                    #=> [2, 12, 14, 16, 18, 20]
```

group_by { |obj| block } → a_hash

group_by → an_enumerator

Groups the collection by result of the block. Returns a hash where the keys are the evaluated result from the block and the values are arrays of elements in the collection that correspond to the key.

If no block is given an enumerator is returned.

```
(1..6).group_by { |i| i%3 }   #=> {0=>[3, 6], 1=>[1, 4], 2=>[2, 5]}
```

include?(obj) → true or false

Returns `true` if any member of *enum* equals *obj*. Equality is tested using `==`.

```
IO.constants.include? :SEEK_SET          #=> true
IO.constants.include? :SEEK_NO_FURTHER   #=> false
IO.constants.member? :SEEK_SET          #=> true
IO.constants.member? :SEEK_NO_FURTHER   #=> false
```

inject(initial, sym) → obj
inject(sym) → obj
inject(initial) { |memo, obj| block } → obj
inject { |memo, obj| block } → obj

Combines all elements of *enum* by applying a binary operation, specified by a block or a symbol that names a method or operator.

The *inject* and *reduce* methods are aliases. There is no performance benefit to either.

If you specify a block, then for each element in *enum* the block is passed an accumulator value (*memo*) and the element. If you specify a symbol instead, then each element in the collection will be passed to the named method of *memo*. In either case, the result becomes the new value for *memo*. At the end of the iteration, the final value of *memo* is the return value for the method.

If you do not explicitly specify an *initial* value for *memo*, then the first element of collection is used as the initial value of *memo*.

```
# Sum some numbers
(5..10).reduce(:+)                             #=> 45
# Same using a block and inject
(5..10).inject { |sum, n| sum + n }            #=> 45
# Multiply some numbers
(5..10).reduce(1, :*)                          #=> 151200
# Same using a block
(5..10).inject(1) { |product, n| product * n } #=> 151200
# find the longest word
longest = %w{ cat sheep bear }.inject do |memo, word|
   memo.length > word.length ? memo : word
end
longest                                        #=> "sheep"
```

lazy → lazy_enumerator

Returns a lazy enumerator, whose methods map/collect, flat_map/collect_concat, select/find_all, reject, grep, #grep_v (Enumerable.html#method-i-grep_v), zip, take, #take_while (Enumerable.html#method-i-take_while), drop, and #drop_while (Enumerable.html#method-i-drop_while) enumerate values only on an as-needed basis. However, if a block is given to zip, values are enumerated immediately.

Example¶ (#method-i-lazy-label-Example) ↑ (#top)

The following program finds pythagorean triples:

```
def pythagorean_triples
  (1..Float::INFINITY).lazy.flat_map {|z|
    (1..z).flat_map {|x|
      (x..z).select {|y|
        x**2 + y**2 == z**2
      }.map {|y|
        [x, y, z]
      }
    }
  }
end
# show first ten pythagorean triples
p pythagorean_triples.take(10).force # take is lazy, so force is needed
p pythagorean_triples.first(10)      # first is eager
# show pythagorean triples less than 100
p pythagorean_triples.take_while { |*, z| z < 100 }.force
```

map { |obj| block } → array
map → an_enumerator

Returns a new array with the results of running *block* once for every element in *enum*.

If no block is given, an enumerator is returned instead.

```
(1..4).map { |i| i*i }      #=> [1, 4, 9, 16]
(1..4).collect { "cat"  }   #=> ["cat", "cat", "cat", "cat"]
```

max → obj
max { |a, b| block } → obj
max(n) → array
max(n) { |a, b| block } → array

Returns the object in *enum* with the maximum value. The first form assumes all objects implement `Comparable`; the second uses the block to return *a <=> b*.

```
a = %w(albatross dog horse)
a.max                                  #=> "horse"
```

```
a.max { |a, b| a.length <=> b.length }  #=> "albatross"
```

If the `n` argument is given, maximum `n` elements are returned as an array.

```
a = %w[albatross dog horse]
a.max(2)                                  #=> ["horse", "dog"]
a.max(2) {|a, b| a.length <=> b.length }  #=> ["albatross", "horse"]
```

max_by {|obj| block } → obj
max_by → an_enumerator
max_by(n) {|obj| block } → obj
max_by(n) → an_enumerator

Returns the object in *enum* that gives the maximum value from the given block.

If no block is given, an enumerator is returned instead.

```
a = %w(albatross dog horse)
a.max_by { |x| x.length }   #=> "albatross"
```

If the `n` argument is given, minimum `n` elements are returned as an array.

```
a = %w[albatross dog horse]
a.max_by(2) {|x| x.length } #=> ["albatross", "horse"]
```

enum.max_by(n) can be used to implement weighted random sampling. Following example implements and use Enumerable#wsample.

```
module Enumerable
  # weighted random sampling.
  #
  # Pavlos S. Efraimidis, Paul G. Spirakis
  # Weighted random sampling with a reservoir
  # Information Processing Letters
  # Volume 97, Issue 5 (16 March 2006)
  def wsample(n)
    self.max_by(n) {|v| rand ** (1.0/yield(v)) }
  end
end
e = (-20..20).to_a*10000
a = e.wsample(20000) {|x|
  Math.exp(-(x/5.0)**2) # normal distribution
}
# a is 20000 samples from e.
p a.length #=> 20000
h = a.group_by {|x| x }
-10.upto(10) {|x| puts "*" * (h[x].length/30.0).to_i if h[x] }
#=> *
#   ***
#   ******
#   ***********
#   ******************
#   *****************************
#   *****************************************
#   ****************************************************
#   ***************************************************************
#   ********************************************************************
#   ***********************************************************************
#   ***********************************************************************
#   **************************************************************
#   ****************************************************
#   ***************************************
#   ***************************
#   ******************
#   ***********
#   *******
#   ***
#   *
```

member?(obj) → true or false

Returns `true` if any member of *enum* equals *obj*. Equality is tested using ==.

```
IO.constants.include? :SEEK_SET          #=> true
IO.constants.include? :SEEK_NO_FURTHER   #=> false
IO.constants.member? :SEEK_SET          #=> true
IO.constants.member? :SEEK_NO_FURTHER   #=> false
```

min → obj
min { |a, b| block } → obj
min(n) → array
min(n) { |a, b| block } → array

Returns the object in *enum* with the minimum value. The first form assumes all objects implement `Comparable`; the second uses the block to return *a* <=> *b*.

```
a = %w(albatross dog horse)
a.min                                  #=> "albatross"
a.min { |a, b| a.length <=> b.length }  #=> "dog"
```

If the `n` argument is given, minimum `n` elements are returned as an array.

```
a = %w[albatross dog horse]
a.min(2)                                  #=> ["albatross", "dog"]
a.min(2) {|a, b| a.length <=> b.length }  #=> ["dog", "horse"]
```

min_by {|obj| block } → obj
min_by → an_enumerator
min_by(n) {|obj| block } → array
min_by(n) → an_enumerator

Returns the object in *enum* that gives the minimum value from the given block.

If no block is given, an enumerator is returned instead.

```
a = %w(albatross dog horse)
a.min_by { |x| x.length }   #=> "dog"
```

If the `n` argument is given, minimum `n` elements are returned as an array.

```
a = %w[albatross dog horse]
p a.min_by(2) {|x| x.length } #=> ["dog", "horse"]
```

minmax → [min, max]
minmax { |a, b| block } → [min, max]

Returns a two element array which contains the minimum and the maximum value in the enumerable. The first form assumes all objects implement `Comparable`; the second uses the block to return *a* <=> *b*.

```
a = %w(albatross dog horse)
a.minmax                                  #=> ["albatross", "horse"]
a.minmax { |a, b| a.length <=> b.length } #=> ["dog", "albatross"]
```

minmax_by { |obj| block } → [min, max]
minmax_by → an_enumerator

Returns a two element array containing the objects in *enum* that correspond to the minimum and maximum values respectively from the given block.

If no block is given, an enumerator is returned instead.

```
a = %w(albatross dog horse)
a.minmax_by { |x| x.length }   #=> ["dog", "albatross"]
```

none? [{ |obj| block }] → true or false

Passes each element of the collection to the given block. The method returns `true` if the block never returns `true` for all elements. If the block is not given, `none?` will return `true` only if none of the collection members is true.

```
%w{ant bear cat}.none? { |word| word.length == 5 } #=> true
%w{ant bear cat}.none? { |word| word.length >= 4 } #=> false
[].none?                                           #=> true
[nil].none?                                        #=> true
[nil, false].none?                                 #=> true
[nil, false, true].none?                           #=> false
```

one? [{ |obj| block }] → true or false

Passes each element of the collection to the given block. The method returns `true` if the block returns `true` exactly once. If the block is not given, `one?` will return `true` only if exactly one of the collection members is true.

```
%w{ant bear cat}.one? { |word| word.length == 4 }  #=> true
```

```
%w{ant bear cat}.one? { |word| word.length > 4 }   #=> false
%w{ant bear cat}.one? { |word| word.length < 4 }   #=> false
[ nil, true, 99 ].one?                             #=> false
[ nil, true, false ].one?                          #=> true
```

partition { |obj| block } → [true_array, false_array]

partition → an_enumerator

Returns two arrays, the first containing the elements of *enum* for which the block evaluates to true, the second containing the rest.

If no block is given, an enumerator is returned instead.

```
(1..6).partition { |v| v.even? }  #=> [[2, 4, 6], [1, 3, 5]]
```

reduce(initial, sym) → obj

reduce(sym) → obj

reduce(initial) { |memo, obj| block } → obj

reduce { |memo, obj| block } → obj

Combines all elements of *enum* by applying a binary operation, specified by a block or a symbol that names a method or operator.

The *inject* and *reduce* methods are aliases. There is no performance benefit to either.

If you specify a block, then for each element in *enum* the block is passed an accumulator value (*memo*) and the element. If you specify a symbol instead, then each element in the collection will be passed to the named method of *memo*. In either case, the result becomes the new value for *memo*. At the end of the iteration, the final value of *memo* is the return value for the method.

If you do not explicitly specify an *initial* value for *memo*, then the first element of collection is used as the initial value of *memo*.

```
# Sum some numbers
(5..10).reduce(:+)                             #=> 45
# Same using a block and inject
(5..10).inject { |sum, n| sum + n }            #=> 45
# Multiply some numbers
(5..10).reduce(1, :*)                          #=> 151200
# Same using a block
(5..10).inject(1) { |product, n| product * n } #=> 151200
# find the longest word
longest = %w{ cat sheep bear }.inject do |memo, word|
   memo.length > word.length ? memo : word
end
longest                                        #=> "sheep"
```

reject { |obj| block } → array

reject → an_enumerator

Returns an array for all elements of `enum` for which the given `block` returns `false`.

If no block is given, an Enumerator (Enumerator.html) is returned instead.

```
(1..10).reject { |i|  i % 3 == 0 }   #=> [1, 2, 4, 5, 7, 8, 10]

[1, 2, 3, 4, 5].reject { |num| num.even? } #=> [1, 3, 5]
```

See also #find_all (Enumerable.html#method-i-find_all).

reverse_each(*args) { |item| block } → enum

reverse_each(*args) → an_enumerator

Builds a temporary array and traverses that array in reverse order.

If no block is given, an enumerator is returned instead.

```
  (1..3).reverse_each { |v| p v }

produces:

  3
  2
  1
```

select { |obj| block } → array

select → an_enumerator

Returns an array containing all elements of `enum` for which the given `block` returns a true value.

If no block is given, an Enumerator (Enumerator.html) is returned instead.

```
(1..10).find_all { |i|  i % 3 == 0 }   #=> [3, 6, 9]
```

```
[1,2,3,4,5].select { |num|  num.even?  }   #=> [2, 4]
```

See also #reject (Enumerable.html#method-i-reject).

slice_after(pattern) → an_enumerator
slice_after { |elt| bool } → an_enumerator

Creates an enumerator for each chunked elements. The ends of chunks are defined by *pattern* and the block.

If `pattern === elt` returns `true` or the block returns `true` for the element, the element is end of a chunk.

The `===` and *block* is called from the first element to the last element of *enum*.

The result enumerator yields the chunked elements as an array. So `each` method can be called as follows:

```
enum.slice_after(pattern).each { |ary| ... }
enum.slice_after { |elt| bool }.each { |ary| ... }
```

Other methods of the Enumerator (Enumerator.html) class and Enumerable (Enumerable.html) module, such as `map`, etc., are also usable.

For example, continuation lines (lines end with backslash) can be concatenated as follows:

```
lines = ["foo\n", "bar\\\n", "baz\n", "\n", "qux\n"]
e = lines.slice_after(/(?<!\)\n\z/)
p e.to_a
#=> [["foo\n"], ["bar\\\n", "baz\n"], ["\n"], ["qux\n"]]
p e.map {|ll| ll[0...-1].map {|l| l.sub(/\\n\z/, "") }.join + ll.last }
#=>["foo\n", "barbaz\n", "\n", "qux\n"]
```

slice_before(pattern) → an_enumerator
slice_before { |elt| bool } → an_enumerator

Creates an enumerator for each chunked elements. The beginnings of chunks are defined by *pattern* and the block.

If `pattern === elt` returns `true` or the block returns `true` for the element, the element is beginning of a chunk.

The `===` and *block* is called from the first element to the last element of *enum*. The result for the first element is ignored.

The result enumerator yields the chunked elements as an array. So `each` method can be called as follows:

```
enum.slice_before(pattern).each { |ary| ... }
enum.slice_before { |elt| bool }.each { |ary| ... }
```

Other methods of the Enumerator (Enumerator.html) class and Enumerable (Enumerable.html) module, such as `to_a`, `map`, etc., are also usable.

For example, iteration over ChangeLog entries can be implemented as follows:

```
# iterate over ChangeLog entries.
open("ChangeLog") { |f|
  f.slice_before(/\A\S/).each { |e| pp e }
}

# same as above.  block is used instead of pattern argument.
open("ChangeLog") { |f|
  f.slice_before { |line| /\A\S/ === line }.each { |e| pp e }
}
```

"svn proplist -R" produces multiline output for each file. They can be chunked as follows:

```
IO.popen([{"LC_ALL"=>"C"}, "svn", "proplist", "-R"]) { |f|
  f.lines.slice_before(/\AProp/).each { |lines| p lines }
}
#=> ["Properties on '.':\n", "  svn:ignore\n", "  svk:merge\n"]
#   ["Properties on 'goruby.c':\n", "  svn:eol-style\n"]
#   ["Properties on 'complex.c':\n", "  svn:mime-type\n", "  svn:eol-style\n"]
#   ["Properties on 'regparse.c':\n", "  svn:eol-style\n"]
#   ...
```

If the block needs to maintain state over multiple elements, local variables can be used. For example, three or more consecutive increasing numbers can be squashed as follows (see `chunk_while` for a better way):

```
a = [0, 2, 3, 4, 6, 7, 9]
prev = a[0]
p a.slice_before { |e|
  prev, prev2 = e, prev
  prev2 + 1 != e
}.map { |es|
  es.length <= 2 ? es.join(",") : "#{es.first}-#{es.last}"
}.join(",")
#=> "0,2-4,6,7,9"
```

However local variables should be used carefully if the result enumerator is enumerated twice or more. The local variables should be initialized for each enumeration. Enumerator.new (Enumerator.html#method-c-new) can be used to do it.

```
# Word wrapping.  This assumes all characters have same width.
def wordwrap(words, maxwidth)
  Enumerator.new {|y|
    # cols is initialized in Enumerator.new.
    cols = 0
    words.slice_before { |w|
      cols += 1 if cols != 0
      cols += w.length
      if maxwidth < cols
        cols = w.length
        true
      else
        false
      end
    }.each {|ws| y.yield ws }
  }
end
text = (1..20).to_a.join(" ")
enum = wordwrap(text.split(/\s+/), 10)
puts "-"*10
enum.each { |ws| puts ws.join(" ") } # first enumeration.
puts "-"*10
enum.each { |ws| puts ws.join(" ") } # second enumeration generates same result as the first.
puts "-"*10
#=> ----------
#   1 2 3 4 5
#   6 7 8 9 10
#   11 12 13
#   14 15 16
#   17 18 19
#   20
#   ----------
#   1 2 3 4 5
#   6 7 8 9 10
#   11 12 13
#   14 15 16
#   17 18 19
#   20
#   ----------
```

mbox contains series of mails which start with Unix From line. So each mail can be extracted by slice before Unix From line.

```
# parse mbox
open("mbox") { |f|
  f.slice_before { |line|
    line.start_with? "From "
  }.each { |mail|
    unix_from = mail.shift
    i = mail.index("\n")
    header = mail[0...i]
    body = mail[(i+1)..-1]
    body.pop if body.last == "\n"
    fields = header.slice_before { |line| !" \t".include?(line[0]) }.to_a
    p unix_from
    pp fields
    pp body
  }
}
```

```
# split mails in mbox (slice before Unix From line after an empty line)
open("mbox") { |f|
  f.slice_before(emp: true) { |line, h|
    prevemp = h[:emp]
    h[:emp] = line == "\n"
    prevemp && line.start_with?("From ")
  }.each { |mail|
    mail.pop if mail.last == "\n"
    pp mail
  }
}
```

slice_when {|elt_before, elt_after| bool } → an_enumerator

Creates an enumerator for each chunked elements. The beginnings of chunks are defined by the block.

This method split each chunk using adjacent elements, *elt_before* and *elt_after*, in the receiver enumerator. This method split chunks between *elt_before* and *elt_after* where the block returns `true`.

The block is called the length of the receiver enumerator minus one.

The result enumerator yields the chunked elements as an array. So `each` method can be called as follows:

```
enum.slice_when { |elt_before, elt_after| bool }.each { |ary| ... }
```

Other methods of the Enumerator (Enumerator.html) class and Enumerable (Enumerable.html) module, such as `to_a`, `map`, etc., are also usable.

For example, one-by-one increasing subsequence can be chunked as follows:

```
a = [1,2,4,9,10,11,12,15,16,19,20,21]
b = a.slice_when {|i, j| i+1 != j }
p b.to_a #=> [[1, 2], [4], [9, 10, 11, 12], [15, 16], [19, 20, 21]]
c = b.map {|a| a.length < 3 ? a : "#{a.first}-#{a.last}" }
p c #=> [[1, 2], [4], "9-12", [15, 16], "19-21"]
d = c.join(",")
p d #=> "1,2,4,9-12,15,16,19-21"
```

Near elements (threshold: 6) in sorted array can be chunked as follows:

```
a = [3, 11, 14, 25, 28, 29, 29, 41, 55, 57]
p a.slice_when {|i, j| 6 < j - i }.to_a
#=> [[3], [11, 14], [25, 28, 29, 29], [41], [55, 57]]
```

Increasing (non-decreasing) subsequence can be chunked as follows:

```
a = [0, 9, 2, 2, 3, 2, 7, 5, 9, 5]
p a.slice_when {|i, j| i > j }.to_a
#=> [[0, 9], [2, 2, 3], [2, 7], [5, 9], [5]]
```

Adjacent evens and odds can be chunked as follows: (Enumerable#chunk is another way to do it.)

```
a = [7, 5, 9, 2, 0, 7, 9, 4, 2, 0]
p a.slice_when {|i, j| i.even? != j.even? }.to_a
#=> [[7, 5, 9], [2, 0], [7, 9], [4, 2, 0]]
```

Paragraphs (non-empty lines with trailing empty lines) can be chunked as follows: (See #chunk (Enumerable.html#method-i-chunk) to ignore empty lines.)

```
lines = ["foo\n", "bar\n", "\n", "baz\n", "qux\n"]
p lines.slice_when {|l1, l2| /\A\s*\z/ =~ l1 && /\S/ =~ l2 }.to_a
#=> [["foo\n", "bar\n", "\n"], ["baz\n", "qux\n"]]
```

#chunk_while (Enumerable.html#method-i-chunk_while) does the same, except splitting when the block returns `false` instead of `true`.

sort → array
sort { |a, b| block } → array

Returns an array containing the items in *enum* sorted, either according to their own <=> method, or by using the results of the supplied block. The block should return -1, 0, or +1 depending on the comparison between *a* and *b*. As of Ruby 1.8, the method `Enumerable#sort_by` implements a built-in Schwartzian Transform, useful when key computation or comparison is expensive.

The result is not guaranteed as stable. When comparison of two elements returns `0`, the order of the elements is unpredictable.

```
%w(rhea kea flea).sort          #=> ["flea", "kea", "rhea"]
(1..10).sort { |a, b| b <=> a }  #=> [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

sort_by { |obj| block } → array
sort_by → an_enumerator

Sorts *enum* using a set of keys generated by mapping the values in *enum* through the given block.

The result is not guaranteed as stable. When two keys are equal, the order of the corresponding elements is unpredictable.

If no block is given, an enumerator is returned instead.

```
%w{apple pear fig}.sort_by { |word| word.length}
              #=> ["fig", "pear", "apple"]
```

The current implementation of `sort_by` generates an array of tuples containing the original collection element and the mapped value. This makes `sort_by` fairly expensive when the keysets are simple.

```
require 'benchmark'

a = (1..100000).map { rand(100000) }

Benchmark.bm(10) do |b|
  b.report("Sort")    { a.sort }
  b.report("Sort by") { a.sort_by { |a| a } }
end
```

produces:

```
user     system      total        real
Sort        0.180000   0.000000   0.180000 (  0.175469)
Sort by     1.980000   0.040000   2.020000 (  2.013586)
```

However, consider the case where comparing the keys is a non-trivial operation. The following code sorts some files on modification time using the basic `sort` method.

```
files = Dir["*"]
sorted = files.sort { |a, b| File.new(a).mtime <=> File.new(b).mtime }
sorted   #=> ["mon", "tues", "wed", "thurs"]
```

This sort is inefficient: it generates two new `File` objects during every comparison. A slightly better technique is to use the `Kernel#test` method to generate the modification times directly.

```
files = Dir["*"]
sorted = files.sort { |a, b|
  test(?M, a) <=> test(?M, b)
}
sorted   #=> ["mon", "tues", "wed", "thurs"]
```

This still generates many unnecessary `Time` objects. A more efficient technique is to cache the sort keys (modification times in this case) before the sort. Perl users often call this approach a Schwartzian Transform, after Randal Schwartz. We construct a temporary array, where each element is an array containing our sort key along with the filename. We sort this array, and then extract the filename from the result.

```
sorted = Dir["*"].collect { |f|
   [test(?M, f), f]
}.sort.collect { |f| f[1] }
sorted   #=> ["mon", "tues", "wed", "thurs"]
```

This is exactly what `sort_by` does internally.

```
sorted = Dir["*"].sort_by { |f| test(?M, f) }
sorted   #=> ["mon", "tues", "wed", "thurs"]
```

sum(init=0) → number
sum(init=0) {|e| expr } → number

Returns the sum of elements in an Enumerable (Enumerable.html).

If a block is given, the block is applied to each element before addition.

If *enum* is empty, it returns *init*.

For example:

```
{ 1 => 10, 2 => 20 }.sum {|k, v| k * v }  #=> 50
(1..10).sum                               #=> 55
(1..10).sum {|v| v * 2 }                  #=> 110
[Object.new].each.sum                     #=> TypeError
```

This method can be used for non-numeric objects by explicit *init* argument.

```
{ 1 => 10, 2 => 20 }.sum([])                   #=> [1, 10, 2, 20]
"a\nb\nc".each_line.lazy.map(&:chomp).sum("")  #=> "abc"
```

#sum (Enumerable.html#method-i-sum) method may not respect method redefinition of "+" methods such as Integer#+.

take(n) → array

Returns first n elements from *enum*.

```
a = [1, 2, 3, 4, 5, 0]
a.take(3)             #=> [1, 2, 3]
a.take(30)            #=> [1, 2, 3, 4, 5, 0]
```

take_while { |obj| block } → array
take_while → an_enumerator

Passes elements to the block until the block returns `nil` or `false`, then stops iterating and returns an array of all prior elements.

If no block is given, an enumerator is returned instead.

```
a = [1, 2, 3, 4, 5, 0]
a.take_while { |i| i < 3 }   #=> [1, 2]
```

to_a(*args) → array

Returns an array containing the items in *enum*.

```
(1..7).to_a                       #=> [1, 2, 3, 4, 5, 6, 7]
{ 'a'=>1, 'b'=>2, 'c'=>3 }.to_a   #=> [["a", 1], ["b", 2], ["c", 3]]

require 'prime'
Prime.entries 10                  #=> [2, 3, 5, 7]
```

to_h(*args) → hash

Returns the result of interpreting *enum* as a list of `[key, value]` pairs.

```
%i[hello world].each_with_index.to_h
  # => {:hello => 0, :world => 1}
```

uniq → new_ary
uniq { |item| ... } → new_ary

Returns a new array by removing duplicate values in `self`.

See also Array#uniq (Array.html#method-i-uniq).

zip(arg, ...) → an_array_of_array
zip(arg, ...) { |arr| block } → nil

Takes one element from *enum* and merges corresponding elements from each *args*. This generates a sequence of *n*-element arrays, where *n* is one more than the count of arguments. The length of the resulting sequence will be `enum#size`. If the size of any argument is less than `enum#size`, `nil` values are supplied. If a block is given, it is invoked for each output array, otherwise an array of arrays is returned.

```
a = [ 4, 5, 6 ]
b = [ 7, 8, 9 ]

a.zip(b)                 #=> [[4, 7], [5, 8], [6, 9]]
[1, 2, 3].zip(a, b)      #=> [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
[1, 2].zip(a, b)         #=> [[1, 4, 7], [2, 5, 8]]
a.zip([1, 2], [8])       #=> [[4, 1, 8], [5, 2, nil], [6, nil, nil]]

c = []
a.zip(b) { |x, y| c << x + y }  #=> nil
c                               #=> [11, 13, 15]
```