Module
Enumerable

Relies on: each, <=>

The Enumerable mixin provides collection classes with 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, #sort, or #sort_by is used, the objects in the collection must also implement a meaningful <=> operator, because these methods rely on an ordering between members of the collection.

1.9 Ruby 1.9 adds a substantial number of methods to this module, as well as changing the semantics of many others. Even experienced Ruby programmers should probably read this section carefully.

Instance methods

all? *enum*.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} (that is all? will return true only if none of the collection members is 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? *enum*.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 is, any? will return true if at least one of the collection members is not false or nil). See also Enumerable#none? and Enumerable#one?.

```
%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
```

collect *enum*.collect {| *obj* | *block* } → *array* or *enumerator*

Returns a new array containing the results of running *block* once for every element in *enum*. Returns an Enumerator object if no block is given.

```
(1..4).collect {|i| i*i }    # =>   [1, 4, 9, 16]
(1..4).collect { "cat"  }    # =>   ["cat", "cat", "cat", "cat"]
(1..4).collect(&:even?)      # =>   [false, true, false, true]
```

count *enum*.count(*obj*) → *int*
enum.count {| *obj* | *block* } → *int*

1.9 Returns the count of objects in *enum* that equal *obj* or for which the block returns a true value. Returns the count of all elements in *enum* if neither a block nor an argument is given.

```
(1..4).count                    # =>   4
(1..4).count(3)                 # =>   1
(1..4).count {|obj| obj > 2 }   # =>   2
```

cycle *enum*.cycle {| *obj* | *block* } → nilor *enumerator*
enum.cycle(*times*) {| *obj* | *block* } → nilor *enumerator*

1.9 Returns nil if *enum* has no elements; otherwise, passes the elements, one at a time to the block. When it reaches the end, it repeats. The number of times it repeats is set by the parameter. If the parameter is missing, cycles forever. Equivalent to *enum*.to_a.cycle. See also Array#cycle. Returns an Enumerator object if no block is given.

```
('a'..'c').cycle(2)       # =>   #<Enumerator:0x0a503c>
('a'..'c').cycle(2).to_a  # =>   ["a", "b", "c", "a", "b", "c"]
```

detect *enum*.detect(*ifnone* = nil) {| *obj* | *block* } → *obj* or nil or *enumerator*

Passes each entry in *enum* to *block*. Returns the first for which *block* is not false. Returns nil if no object matches unless the proc *ifnone* is given, in which case it is called and its result is returned. Returns an Enumerator object if no block is given.

```
(1..10).detect  {|i| i % 5 == 0 and i % 7 == 0 }   # =>   nil
(1..100).detect {|i| i % 5 == 0 and i % 7 == 0 }   # =>   35
sorry = lambda { "not found" }
(1..10).detect(sorry) {|i| i > 50}                 # =>   "not found"
```

drop *enum*.drop(*n*) → *an_array*

1.9 Returns an array containing all but the first *n* elements of *enum*.

```
[ 1, 1, 2, 3, 5, 8, 13 ].drop(4)    # =>   [5, 8, 13]
[ 1, 1, 2, 3, 5, 8, 13 ].drop(99)   # =>   []
```

drop_while *enum*.drop_while {| *item* | *block* } → *an_array* or *enumerator*

1.9 Passes elements in turn to the block until the block does not return a true value. Starting with that element, copies the remainder to an array and returns it. Returns an Enumerator object if no block is given.

```
[ 1, 1, 2, 3, 5, 8, 13 ].drop_while {|item| item < 6 }   # =>   [8, 13]
```

each_cons *enum*.each_cons(*length*) {| *array* | *block* } → nil or *enumerator*

1.9 Passes to the block each consecutive subarray of size *length* from self. Returns an Enumerator object if no block is given.

```
(1..4).each_cons(2) {|array| p array }
```

produces:

```
[1, 2]
[2, 3]
[3, 4]
```

each_slice *enum*.each_slice(*length*) {| *array* | *block* } → nil or *enumerator*

1.9 Divides *enum* into slices of size *length*, passing each in turn to the block. Returns an Enumerator object if no block is given.

```
(1..10).each_slice(4) {|array| p array }
```

produces:

```
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10]
```

each_with_index *enum*.each_with_index(⟨ args ⟩*) {| *obj, index* | *block* } → *enum* or *enumerator*

1.9 Calls *block*, passing in successive items from *enum* and the corresponding index. If any arguments are given, they are passed to each during the iteration. Returns an Enumerator object if no block is given.

```
%w(cat dog wombat).each_with_index do |item, index|
  puts "#{item} is at position #{index}"
end
```

produces:

```
cat is at position 0
dog is at position 1
wombat is at position 2
```

each_with_object *enum*.each_with_object(*memo*) → *memo* or *enumerator*

1.9 Calls *block* with two arguments, the item and the memo object, for each item in *enum*. Returns an Enumerator object if no block is given.

```
hash = %w(cat dog wombat).each_with_object({}) do |item, memo|
  memo[item] = item.upcase.reverse
end
hash   # =>   {"cat"=>"TAC", "dog"=>"GOD", "wombat"=>"TABMOW"}
```

entries *enum*.entries → *array*

Synonym for Enumerable#to_a.

find *enum*.find(*ifnone* = nil) {| *obj* | *block* } → *obj* or nil

Synonym for Enumerable#detect.

find_all *enum*.find_all {| *obj* | *block* } → *array* or *enumerator*

Returns an array containing all elements of *enum* for which *block* is not false (see also Enumerable#reject). Returns an Enumerator object if no block is given.

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

find_index *enum*.find_index {| *obj* | *block* } → *int* or nil or *enumerator*

1.9 Returns the index of the first item for which the given block returns a true value or returns or nil if the block only ever returns false. *block* is not false (see also Enumerable#reject). Returns an Enumerator object if no block is given.

```
%w{ant bat cat dog}.find_index {|item|  item =~ /g/ }   # =>   3
%w{ant bat cat dog}.find_index {|item|  item =~ /h/ }   # =>   nil
```

first

enum.first → *an_object* or nil
enum.first(*n*) → *an_array*

1.9 With no parameters, returns the first item of *enum* or nil. With a parameter, returns the first *n* items of *enum*.

```
%w{ant bat cat dog}.find_index.first     # =>   "ant"
%w{ant bat cat dog}.find_index.first(2)  # =>   ["ant", "bat"]
```

grep

enum.grep(*pattern*) → *array*
enum.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]
[ 123, 9**11, 12.34 ].grep(Integer)    # =>   [123, 31381059609]
```

group_by

enum.group_by {| *item* | *block* } → *hash* or *enumerator*

1.9 Partitions *enum* by calling the block for each item and using the result returned by the block to group the items into buckets. Returns a hash where the keys are the objects returned by the block, and the values for a key are those items for which the block returned that object. Returns an Enumerator object if no block is given.

```
p (1..5).group_by {|item| item.even? ? "even" : "odd" }
```

produces:

```
{"odd"=>[1, 3, 5], "even"=>[2, 4]}
```

include?

enum.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
```

inject

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

1.9 Combines the items in *enum* by iterating over them. For each item, passes an accumulator object (called *memo* in the examples) and the item itself to the block, or invokes memo.send(sym, obj). At each step, *memo* is set to the value returned by the block on the previous step. The value returned by inject is the final value returned by the block. The first two forms let you supply an initial value for *memo*. The second two forms use the first element of the collection as the initial value (and skip that element while iterating). Some languages call this operation foldl or reduce. Ruby supports the latter as an alias for inject.

```
# Sum some numbers. These forms do the same thing
(5..10).inject(0) {|sum, n| sum + n }          # =>   45
(5..10).inject {|sum, n| sum + n }             # =>   45
(5..10).inject(0, :+)                          # =>   45
(5..10).inject(:+)                             # =>   45

# Multiply some numbers
(5..10).inject(1) {|product, n| product * n }  # =>   151200

# find the longest word
longest_word = %w{ cat sheep bear }.inject do |memo, word|
   memo.length > word.length ? memo : word
end
longest_word                                   # =>   "sheep"

# find the length of the longest word
longest_length = %w{ cat sheep bear }.inject(0) do |memo, word|
   memo >= word.length ? memo : word.length
end
longest_length                                 # =>   5
```

map

enum.map {| *obj* | *block* } → *array*

Synonym for Enumerable#collect.

max

enum.max → *obj*
enum.max {| *a,b* | *block* } → *obj*

Returns the object in *enum* with the maximum value. The first form assumes all objects implement <=>; 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"
```

max_by

1.9

enum.max_by {| *item* | *block* } → *obj* or *enumerator*

Passes each item in the collection to the block. Returns the item corresponding to the largest value returned by the block. Returns an Enumerator object if no block is given.

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

member?

enum.member?(*obj*) → true or false

Synonym for Enumerable#include?.

min

enum.min → *obj*
enum.min {| *a,b* | *block* } → *obj*

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"
```

min_by

enum.min_by {| *a,b* | *block* } → *obj* or *enumerator*

1.9

Passes each item in the collection to the block. Returns the item corresponding to the smallest value returned by the block. Returns an Enumerator object if no block is given.

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

minmax

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

1.9

Compares the elements of self using either <=> of the given block, returning the minimum and maximum values.

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

minmax_by

enum.minmax_by {| *a,b* | *block* } → [*min, max*] or *enumerator*

1.9

Passes each item in the collection to the block. Returns the items corresponding to the smallest and largest values returned by the block. Returns an Enumerator object if no block is given.

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

none?

enum.none? ⟨ {| *obj* | *block* } ⟩ → true or false

1.9

Passes each element of the collection to the given block. The method returns true if the block never returns a value other than false or nil. If the block is not given, Ruby adds an implicit block of {|obj| obj} (that is, any? will return true if at least one of the collection members is not false or nil). See also Enumerable#any? and Enumerable#one?.

```
%w{ ant bear cat}.none? {|word| word.length >= 3}   # =>   false
%w{ ant bear cat}.none? {|word| word.length >  3}   # =>   false
[ nil, true, 99 ].none?                             # =>   false
```

one?

enum.one? ⟨ {| *obj* | *block* } ⟩ → true or false

1.9

Passes each element of the collection to the given block. The method returns true if the block returns true exactly one time. If the block is not given, Ruby adds an implicit block of

{|obj| obj} (that is, any? will return true if at least one of the collection members is not false or nil). See also Enumerable#any? and Enumerable#none?.

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

partition *enum*.partition {| *obj* | *block* } → [*true_array*, *false_array*] or *enumerator*

Returns two arrays, the first containing the elements of *enum* for which the block evaluates to true, the second containing the rest. Returns an Enumerator object if no block is given.

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

reduce *enum*.reduce(*initial*) {| memo, obj | *block* } → *obj*
enum.reduce(*initial*, *sym*) → *obj*
enum.reduce {| memo, obj | *block* } → *obj*
enum.reduce(*sym*) → *obj*

1.9 Synonym for Enumerable#inject.

reject *enum*.reject {| *obj* | *block* } → *array* or *enumerator*

Returns an array containing the elements of *enum* for which *block* is false (see also Enumerable#find_all). Returns an Enumerator object if no block is given.

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

reverse_each *enum*.reverse_each {| *obj* | *block* } → *enum*

1.9 Invokes the block with the elements of *enum* in reverse order. Creates an intermediate array internally, so this might be expensive on large collections. Returns an Enumerator object if no block is given.

```
(1..5).reverse_each {|i|  print i, " " }
```

produces:

```
5 4 3 2 1
```

select *enum*.select {| *obj* | *block* } → *array*

Synonym for Enumerable#find_all.

sort *enum*.sort → *array*
enum.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*. See also Enumerable#sort_by.

```
%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 *enum*.sort_by {| *obj* | *block* } → *array*

Sorts *enum* using keys generated by mapping the values in *enum* through the given block, using the result of that block for element comparison.

```
sorted = %w{ apple pear fig }.sort_by {|word| word.length}

sorted   # =>   ["fig", "pear", "apple"]
```

Internally, 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'
include Benchmark
a = (1..100000).map {rand(100000)}
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        0.030000   0.000000   0.030000 (  0.030295)
Sort by     0.140000   0.010000   0.150000 (  0.144596)
```

However, in cases where comparing the keys is a nontrivial operation, the algorithm used by sort_by is considerably faster.[1]

sort_by can also be useful for multilevel sorts. One trick, which relies on the fact that arrays are compared element by element, is to have the block return an array of each of the comparison keys. For example, to sort a list of words first on their length and then alphabetically, you could write the following:

```
words = %w{ puma cat bass ant aardvark gnu fish }
sorted = words.sort_by {|w| [w.length, w] }
sorted   # =>   ["ant", "cat", "gnu", "bass", "fish", "puma", "aardvark"]
```

Returns an Enumerator object if no block is given.

take *enum*.take(*n*) → *array*

1.9

Returns an array containing the first *n* items from *enum*.

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

take_while *enum*.take_while {| *item* | *block* } → *array* or *enumerator*

1.9

Passes successive items to the block, adding them to the result array until the block returns false or nil. Returns an Enumerator object if no block is given.

1. It caches the sort keys before the sort. Perl users often call this approach a Schwartzian Transform, named after Randal Schwartz.

```
(1..7).take_while {|item| item < 3 }         # =>   [1, 2]
[ 2, 4, 6, 9, 11, 16 ].take_while(&:even?)   # =>   [2, 4, 6]
```

to_a

enum.to_a(*args) → *array*

1.9

Returns an array containing the items in *enum*. This is done using the each method. Any arguments passed to to_a are passed to each.

```
(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]]
```

zip

enum.zip(⟨ *arg* ⟩$^+$) → *array*
enum.zip(⟨ *arg* ⟩$^+$) {| *arr* | *block* } → nil

Converts any arguments to arrays and then merges elements of *enum* with corresponding elements from each argument. The result is an array containing the same number of elements as *enum*. Each element is a *n*-element array, where *n* is one more than the count of arguments. If the size of any argument is less than the number of elements in *enum*, nil values are supplied. If a block given, it is invoked for each output array; otherwise, an array of arrays is returned.

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

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