Class Range < Object

A Range represents an interval—a set of values with a start and an end. Ranges may be constructed using the *s*..*e* and *s*...*e* literals or using Range.new. Ranges constructed using .. run from the start to the end inclusively. Those created using ... exclude the end value. When used as an iterator, ranges return each value in the sequence.

```
(-1..-5).to_a      # =>   []
(-5..-1).to_a      # =>   [-5, -4, -3, -2, -1]
('a'..'e').to_a    # =>   ["a", "b", "c", "d", "e"]
('a'...'e').to_a   # =>   ["a", "b", "c", "d"]
```

Ranges can be constructed using objects of any type, as long as the objects can be compared using their <=> operator and they support the succ method to return the next object in sequence.

```
class Xs                # represent a string of 'x's
  include Comparable
  attr :length
  def initialize(n)
    @length = n
  end
  def succ
    Xs.new(@length + 1)
  end
  def <=>(other)
    @length <=> other.length
  end
  def inspect
    'x' * @length
  end
end

r = Xs.new(3)..Xs.new(6)   # =>   xxx..xxxxxx
r.to_a                     # =>   [xxx, xxxx, xxxxx, xxxxxx]
r.member?(Xs.new(5))       # =>   true
```

In the previous code example, class Xs includes the Comparable module. This is because Enumerable#member? checks for equality using ==. Including Comparable ensures that the == method is defined in terms of the <=> method implemented in Xs.

Mixes in

Enumerable:

all?, any?, collect, count, cycle, detect, drop, drop_while, each_cons, each_slice, each_with_index, entries, find, find_all, find_index, first, grep, group_by, include?, inject, map, max, max_by, member?, min, min_by, minmax, minmax_by, none?, one?, partition, reduce, reject, select, sort, sort_by, take, take_while, to_a, zip

Class methods

new Range.new(*start*, *end*, *exclusive*=false) → *rng*

Constructs a range using the given *start* and *end*. If the third parameter is omitted or is false, the range will include the end object; otherwise, it will be excluded.

Instance methods

== *rng* == *obj* → true or false

Returns true if *obj* is a range whose beginning and end are the same as those in *rng* (compared using ==) and whose *exclusive* flag is the same as *rng*.

=== *rng* === *val* → true or false

If *rng* excludes its end, returns $rng.start \leq val < rng.end$. If *rng* is inclusive, returns $rng.start \leq val \leq rng.end$. Note that this implies that *val* need not be a member of the range itself (for example, a float could fall between the start and end values of a range of integers). Conveniently, the === operator is used by case statements.

```
case 74.95
when  1...50  then   puts "low"
when 50...75  then   puts "medium"
when 75...100 then   puts "high"
end
```

produces:

```
medium
```

Implemented internally by calling include?.

begin *rng*.begin → *obj*

Returns the first object of *rng*.

cover? *rng*.cover?(*obj*) → true or false

1.9

Returns true if *obj* lies between the start and end of the range. For ranges defined with $min..max$, this means $min \leq obj \leq max$. For ranges defined with $min...max$, it means $min \leq obj < max$.

```
(1..10).cover?(0)     # =>   false
(1..10).cover?(1)     # =>   true
(1..10).cover?(5)     # =>   true
(1..10).cover?(9.5)   # =>   true
(1..10).cover?(10)    # =>   true
(1...10).cover?(10)   # =>   false
```

each *rng*.each {| *i* | *block* } → *rng*

Iterates over the elements *rng*, passing each in turn to the block. Successive elements are generated using the succ method.

```
(10..15).each do |n|
   print n, ' '
end
```

produces:

```
10 11 12 13 14 15
```

end *rng*.end → *obj*

Returns the object that defines the end of *rng*.

```
(1..10).end    # =>   10
(1...10).end   # =>   10
```

eql? *rng*.eql?(*obj*) → true or false

Returns true if *obj* is a range whose beginning and end are the same as those in *rng* (compared using eql?) and whose *exclusive* flag is the same as *rng*.

exclude_end? *rng*.exclude_end? → true or false

Returns true if *rng* excludes its end value.

first *rng*.first(*n* = 1) → *obj* or *array*

1.9

Returns the first (or first *n*) elements of *rng*.

```
('aa'..'bb').first      # =>   "aa"
('aa'..'bb').first(5)   # =>   ["aa", "ab", "ac", "ad", "ae"]
```

include? *rng*.include?(*val*) → true or false

1.9

Returns true if *val* is one of the values in *rng* (that is if Range#each would return *val* at some point). If the range is defined to span numbers, this method returns true if the value lies between the start and end of the range, even if it is not actually a member (that is, it has the same behavior as Range#cover?). Otherwise, the parameter must be a member of the range.

```
r = 1..10
r.include?(5)       # =>   true
r.include?(5.5)     # =>   true
r.include?(10)      # =>   true
r = 1...10
r.include?(10)      # =>   false
r = 'a'..'z'
r.include?('b')     # =>   true
r.include?('ruby')  # =>   false
```

last *rng*.last(*n* = 1) → *obj* or *array*

1.9

Returns the last (or last *n*) elements of *rng*.

```
('aa'..'bb').last      # =>   "bb"
('aa'..'bb').last(5)   # =>   ["ax", "ay", "az", "ba", "bb"]
```

max

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

1.9 Returns the maximum value in the range. The block is used to compare values if present.

```
(-3..2).max                     # =>   2
(-3..2).max {|a,b| a*a <=> b*b }  # =>   -3
```

member?

rng.member?(*val*) → true or false

Synonym for Range#include?.

min

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

1.9 Returns the minimum value in the range. The block is used to compare values if present.

```
(-3..2).min                     # =>   -3
(-3..2).min {|a,b| a*a <=> b*b }  # =>   0
```

step

rng.step(*n*=1) ⟨ {| *obj* | *block* } ⟩ → *rng* or *enum*

1.9 Iterates over *rng*, passing each n^{th} element to the block. If the range contains numbers, addition by one is used to generate successive elements. Otherwise, step invokes succ to iterate through range elements. If no block is given, an enumerator is returned. The following code uses class Xs defined at the start of this section:

```
range = Xs.new(1)..Xs.new(10)
range.step(2) {|x| p x}
enum = range.step(3)
p enum.to_a
```

produces:

```
x
xxx
xxxxx
xxxxxxx
xxxxxxxxx
[x, xxxx, xxxxxxx, xxxxxxxxxx]
```

Here's step with numbers:

```
(1..5).step(1).to_a         # =>   [1, 2, 3, 4, 5]
(1..5).step(2).to_a         # =>   [1, 3, 5]
(1..5).step(1.5).to_a       # =>   [1.0, 2.5, 4.0]
(1.0..5.0).step(1).to_a     # =>   [1.0, 2.0, 3.0, 4.0, 5.0]
(1.0..5.0).step(2).to_a     # =>   [1.0, 3.0, 5.0]
(1.0..5.0).step(1.5).to_a   # =>   [1.0, 2.5, 4.0]
```