

Ruby: Useful Classes and Methods

Computer Science and Engineering ■ College of Engineering ■ The Ohio State University

Lecture 7

Ranges

- Instance of class (Range)

```
indices = Range.new(0, 5)
```

- But literal syntax is more common

```
nums = 1..10      # end inclusive
```

```
b = 'cab'...'cat' # end exclusive
```

- Method `to_a` converts a range to an array

```
nums.to_a      #=> [1,2,3,4,5,6,7,8,9,10]
```

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

```
(5..0).to_a    #=> []
```

- Methods `begin/end`, `first/last`

```
b.last      #=> "cat", excluded from range!
```

```
b.last 2    #=> ["car", "cas"]
```

Range Inclusion

- ❑ Operator `===` (aka “case equality”)

```
nums === 6    #=> true  
b === 'cat'   #=> false
```

- ❑ Two methods: `cover?` `include?`

- `cover?` compares to end points
- `include?` (usually) iterates through range, looking for (object value) equality

- ❑ Case statement (`case/when`) with ranges

```
case target  
when 0...mid  
  puts 'first half'  
when mid...size  
  puts 'second half'  
end
```

Strings

- A rich class: 100+ methods!
 - See www.ruby-doc.org
- Note convention on method names
 - ? suffix: polar result (*e.g.*, boolean)
 - ! suffix: dangerous (*e.g.*, changes receiver)
- Examples
 - `empty? start_with? include? length`
 - `to_f, to_i, split` *# convert string to..*
 - `upcase downcase capitalize` *# +/- !*
 - `clear replace` *# no ! (!!)*
 - `chomp chop slice` *# +/- !*
 - `sub gsub` *# +/- !*

Examples

```
s = 'hello world'
s.start_with? 'hi'  #=> false
s.length           #=> 11
'3.14'.to_f        #=> 3.14
s.upcase           #=> "HELLO WORLD", s unchanged
s.capitalize!      #=> s is now "Hello world"
s.split            #=> ["Hello", "world"]
s.split 'o'        #=> ["Hell", " w", "rld"]
s.replace 'good bye' #=> s is "good bye"
s.slice 3, 4        #=> "d by" (start, length)
s[-2, 1]           #=> "y" [start, length]
s.chomp!           #=> remove trailing \n if there
```

Arrays

□ Instance of class (Array)

```
a = Array.new 4      #=> [nil, nil, nil, nil]
```

```
a = Array.new 4, 0   #=> [0, 0, 0, 0]
```

□ But literal notation is common

```
b = [6, 2, 3.14, 'pi', []]
```

```
t = %w{hi world} #=> ["hi", "world"]
```

□ Methods for element access, modification

```
b.length #=> 5
```

```
b[0]      #=> 6 (also b.first, b.last)
```

```
b[-2]     #=> "pi"
```

```
b[10] = 4 # assignment past end of array
```

```
b.length #=> 11, size has changed!
```

```
b[2, 5]   #=> [3.14, "pi", [], nil, nil]
```

Mutators: Growing/Shrinking

- Add/remove from end: **push/pop** (<<)

```
n = [10, 20]
```

```
n.push 30, 40 #=> [10, 20, 30, 40]
```

```
n.pop #=> 40, n is now [10, 20, 30]
```

```
n << 50 #=> [10, 20, 30, 50]
```

- Add/remove from beginning: **unshift/shift**

```
n = [10, 20]
```

```
n.unshift 30, 40 #=> [30, 40, 10, 20]
```

```
n.shift #=> 30
```

- Push/shift gives FIFO queue
- All modify the receiver (but no !)

Concatenation and Difference

□ Concatenation: `+/concat`

`n = [1]`

`n.concat [3, 4]` $\Rightarrow [1, 3, 4]$

`[5, 1] + [5, 2, 3]` $\Rightarrow [5, 1, 5, 2, 3]$

`n.push [3, 4]` $\Rightarrow [1, 3, 4, [3, 4]]$

□ Difference: `-`

`n = [1, 1, 3, 3, 4, 5]`

`n - [1, 2, 4]` $\Rightarrow [3, 3, 5]$

□ Concat modifies receiver, `+/ -` do not

And Many More

□ Element order

```
[1, 2, 3, 4].reverse #=> [4, 3, 2, 1]
```

```
[1, 2, 3, 4].rotate #=> [2, 3, 4, 1]
```

```
[1, 2, 3, 4].shuffle #=> [2, 1, 4, 3]
```

```
[3, 4, 2, 1].sort #=> [1, 2, 3, 4]
```

□ Search

```
[7, 3, 5, 7, 0].find_index 7 #=> 0
```

```
[7, 3, 5, 7, 0].rindex 7 #=> 3
```

```
[7, 3, 5, 7, 0].include? 0 #=> true
```

□ Transformation

```
[1, 2, 2, 3, 1].uniq #=> [1, 2, 3]
```

```
[1, 2].fill 'a' #=> ["a", "a"], N.B. aliases!
```

```
['a', 'bbb', 'c'].join "_" #=> "a_bbb_c"
```

```
[1,2].product [3,4] #=> [[1,3],[1,4],[2,3],[2,4]]
```

```
[[1, 2], [3, 4], [5, 6]].transpose  
#=> [[1, 3, 5], [2, 4, 6]]
```

To Ponder: Array Defaults

Evaluate the ?'s

```
x = Array.new 3, 5 #=> [5, 5, 5]
```

```
x[0] += 1
```

```
x #=> ???
```

```
y = Array.new 3, [] #=> [[], [], []]
```

```
y[0] << 'hi' # adds elt to array
```

```
y #=> ???
```

Example: Password Generator

- Generate a random sequence of 8 lower case letters, without repetition
- E.g., "minbevtj"

Example: Sorting a List

- Write a program that reads in a list of names from stdin (keyboard), then prints out the list in alphabetical order in all-caps

- Hint:

- Use `gets` to read input from stdin
- Returns String up to and including newline (nil if ^d)

```
>> x = gets
```

```
Hello world
```

```
=> "Hello world\n"
```

Refactor: Array Literal

```
index = 0
names = []
while name = gets
  name.chomp!.upcase!
  names[index] = name
  index += 1
end

puts 'The sorted array:'
puts names.sort
```

Refactor: Extend Array

```
index = 0
names = []
while name = gets

  names[index] = name.chomp.upcase
  index += 1
end

puts 'The sorted array:'
puts names.sort
```

Refactor: Push

```
names = []  
while name = gets  
  
  names.push name.chomp.upcase  
  
end  
  
puts 'The sorted array: '  
puts names.sort
```

Refactor: Push Operator

```
names = []  
while name = gets  
  
    names << name.chomp.upcase  
  
end  
  
puts 'The sorted array:'  
puts names.sort
```


Refactor: Statement Modifier

```
names, name = [], ""
```

```
names << name.chomp.upcase while name = gets
```

```
puts 'The sorted array:'
```

```
puts names.sort
```

Summary: Useful Methods and Classes

- Naming convention for methods
 - Mutators marked with !, polar with ?
- Ranges
 - Inclusive, exclusive, operator ===
 - Case/when can use ranges
- Strings
 - Mutable (c.f. Java)
- Arrays
 - Can grow and shrink

Splat "Operator" *

- Split/gather arrays/elements
 - Not really an operator, must be outermost
- Parallel assignment splits/gathers a little
 - `a, b = [1, 2]      #=> a, b == 1, 2`
 - `array = 1, 2, 3    #=> array == [1, 2, 3]`
- On RHS, splats generalize split
 - `a, b, c = 1, *[2, 3]      #=> a, b, c == 1, 2, 3`
- On LHS, splat generalizes gather
 - `*r = 1      #=> [1]`
 - `a, b, *r = 1, 2, 3, 4      #=> r == [3, 4]`
 - `a, b, *r = [1, 2, 3, 4]    #=> r == [3, 4]`
 - `a, b, *r = 1, 2, 3      #=> r == [3]`

Splat in Function Definition/Use

- ❑ Ruby enforces: number of arguments equals number of parameters
- ❑ In function definitions, splat gathers up remaining arguments (*ie* var args)

```
def greet(msg, *names)
  names.each { |name|
    puts "#{msg} #{name}!" }
end
greet 'Ciao', 'Rafe', 'Sarah', 'Xi'
```

- ❑ In function calls, splat explodes arrays into multiple arguments

```
people = ['Rafe', 'Sarah', 'Xi']
greet 'Hi', *people
```