Ruby (on Rails)

Week 1 Section

Extended based on Slides by Kelly Dunn, Kim Todd, Ryan Tucker

http://courses.cs.washington.edu/courses/cse190m/09sp/ruby/week1/week1.pdf

Liang Gong, Electric Engineering & Computer Science, University of California, Berkeley.

What is Ruby?

- Dynamically typed Programming Language

```
a = 1 ; a = "1" ; a = Object.new
```

- Object-oriented
- Interpreted + Just-In-Time Compiler (JIT)

hello_world.rb

```
puts "Hello world!"
```

Running Ruby Programs

- Use the Ruby interpreter

```
ruby hello_world.rb
```

  - "`ruby`" tells the computer to use the Ruby interpreter
- Interactive Ruby (irb) console

```
irb
```

  - Get immediate feedback
  - Test Ruby features

Comments

a single line comment

(**//** in Java)

```
=begin
 this is a multiline comment
 nothing in here will be
 part of the code
=end
```

(**/* … */** in Java)

Variables

- Declaration – No need to declare a "type"
- Assignment – same as in Java
- Example:

```
x = "hello world"        # String
x = 3                    # Fixnum
x = 4.5                  # Float
x = 1..10                # Range
x = 1...10               # Range
```

Objects

- Everything is an object.
  - Common Types (Classes): Numbers, Strings, Ranges
  - nil, Ruby's equivalent of null is also an object
- Uses "dot-notation" like Java objects

- You can find the class of any variable

```
x = "hello"
x.class          →    String
```

Objects and Messages

- No primitive values, everything is an object.
- Numbers are objects
- Different Classes of Numbers
  - FixNum, Float

```
3.eql?2          → false
-42.abs          → 42
3.4.round        → 3
3.6.round        → 4
3.2.ceil         → 4
3.8.floor        → 3
3.zero?          → false
```

Changing types

- You may want to treat a String a number or a number as a String
  - `to_i` – converts to an integer (Fixnum)
  - `to_f` – converts a String to a Float
  - `to_s` – converts a number to a String
- Examples

```
"3.5".to_i      →   3
"3.5".to_f      →   3.5
3.to_s          →   "3"
```

Arrays

- Are indexed by zero-based integer values
- Are declared using square brackets, [], elements are separated by commas
- Store an assortment of types within the same array

```
a = [1, 4.3, "hello", 3..7]
```

- Example:

```
a[0]  →    1
a[2]  →    "hello"
```

Arrays

- You can assign values to an array at a particular index
- Arrays increase in size if an index is specified out of bounds and fill gaps with nil
- Example:

```
a = [1, 4.3, "hello", 3..7]
a[4] = "goodbye"
a
> [1, 4.3, "hello", 3..7, "goodbye"]
a[6] = "hola"
a
> [1, 4.3, "hello", 3..7, "goodbye", nil, "hola"]
```

Negative Integer Index

- Negative integer values can be used to index values in an array

- Example:

```
a = [1, 4.3, "hello", 3..7]
a[-1]              →     3..7
a[-2]              →     "hello"
a[-3] = 83.6
a                  → [1, 83.6, "hello", 3..7]
```

if/elsif/else/end

- Must use "*elsif*" instead of "*else if*"
- Notice use of "*end*". It replaces closing curly braces in Java
- Example:

```
if age < 35
  puts "young whipper-snapper"
elsif  age < 105
  puts "80 is the new 30!"
else
  puts "wow..."
end
```

Inline "*if*" statements

- Original if-statement

```
if age < 105
  puts "don't worry, you are still young"
end
```

- Inline if-statement

```
puts "you are still young" if age < 105
```

for-loops

- for-loops can use ranges
- Example 1:

```
for i in 1..10
    puts i
end
```

```
1
2
3
4
5
6
7
8
9
10
```

- Can also use blocks (covered next week)

```
3.times do
  puts "Ryan! "
end
```

```
Ryan!
Ryan!
Ryan!
```

for-loops

- for-loops can use ranges
- Example 1:

```
for i in 1...10
    puts i
end
```

```
1
2
3
4
5
6
7
8
9
```

- Can also use blocks (covered next week)

```
3.times do
  puts "Ryan! "
end
```

```
Ryan!
Ryan!
Ryan!
```

for-loops

- for-loops can use arrays
- Example 1:

```
arr = [1, 2, 3, 4]
for i in arr
    puts i
end
```

```
1
2
3
4
```

while-loops

- Can also use blocks
- Cannot use "*i++*", "*--i*", "*i--*", "*--i*"
- Can use "*i += 1*", "*i -= 1*"
- Example:

```
i = 0
while i < 5
    puts i
    i = i + 1
end
```

Methods

- Structure

```
def method_name( parameter1, parameter2, …)
      statements
end
```

```
method_name(arg1, arg2)
method_name arg1, arg2
```

- Simple Example:

```
def print_ryan
  puts "Ryan"
end
```

```
print_ryan
> Ryan

print_ryan()
> Ryan
```

Return

- Ruby methods return the value of the last statement in the method, so…

```
def add(num1, num2)
  sum = num1 + num2
  return sum
end
```

can become

```
def add(num1, num2)
  num1 + num2
end
```

Blocks

- Blocks are simply "blocks" of code that can be passed and called (or yield)

Similar to:

–First class functions / lambda calculus

–Callback functions in Python or JavaScript

–function pointer in C/C++

Blocks

- They are defined by curly braces, ***{}***, or a ***do/end*** statement

```
{ |v| puts v }
```

```
do |v|
  p v
end
```

```
{|k, v| puts k; puts v ; ... }
```

```
do |k, v|
  p k
  p v
  ...
end
```

```
(0..10).each { |v| print v } →    012345678910
[1,2,3].each { |v| print v } →    123
4.times { |v| print v }      →    0123
```

Yield

- ***yield*** statements go hand-in-hand with blocks
- The code of a block is executed when a yield statement called

```
def func(arg1, arg2)
  yield arg1, arg2
end

func("a", "b") {|v1, v2| puts (v1 + v2) }
```

→ **ab**

Yield

- *yield* statements go hand-in-hand with blocks
- The code of a block is executed when a yield statement called

```
def func(arg1, arg2)
  yield arg1, arg2
end

func("a", "b") do |v1, v2| puts (v1 + v2) end
```

→ ab

Iterator

- Collection-like (array, range etc.) objects have iterator methods.
- Iterator takes a block as callback and yield the block with each element in the collection.

```
(0..10).each { |v| print v } →    012345678910
[1,2,3].each { |v| print v } →    123
4.times { |v| print v }      →    0123
```

Define Class

```
class C1
  def meth1
      puts "meth1"
  end

  def meth2
      puts "meth2"
  end
end

obj = C1.new  # creates C1 object
obj.meth1     # call instance method meth1
obj.meth2     # call instance method meth2
```

Re-define part of a Class

```
class C1
  def meth1
      puts "meth1"
  end
  ...
end
...
class C1
  def meth1
      puts "meth1 (2nd edition)"
  end
end

obj = C1.new  # creates C1 object
obj.meth1     # call instance method meth1

> meth1 (2nd edition)
```