Ruby (on Rails)

Tutorial

Syntax

About the Section

- Introduce the Ruby programming language
- Use Ruby to template web pages
- Learn about Ruby on Rails and its benefits

puts vs. print

- "puts" adds a new line after it is done
  - analogous System.out.println()
- "print" does not add a new line
  - analogous to System.out.print()

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

```
# this is a single line comment

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

Variables

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

```
x = "hello world"          # String
y = 3                      # Fixnum
z = 4.5                    # Float
r = 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
```

- You can find the methods of any variable or class

```
x = "hello"
x.methods
String.methods
```

Objects (cont.)

- There are many methods that all Objects have
- Include the "?" in the method names, it is a Ruby naming convention for boolean methods
    - nil?
    - eql?/equal?
    - ==, !=, ===
    - instance_of?
    - is_a?
    - to_s

Numbers

- Numbers are objects
- Different Classes of Numbers
  - FixNum, Float

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

String Methods

"hello world".length	→	11
"hello world".nil?	→	false
"".nil?	→	false
"ryan" > "kelly"	→	true
"hello_world!".instance_of?String	→	true
"hello" * 3	→	"hellohellohello"
"hello" + " world"	→	"hello world"
"hello world".index("w")	→	6

Operators and Logic

- Same as Java
  - Multiplication, division, addition, subtraction, etc.
- Also same as Java AND Python (WHA?!)
  - "and" and "or" as well as "&&" and "||"
- Strange things happen with Strings
  - String concatenation (+)
  - String multiplication (*)
- Case and Point: There are many ways to solve a problem in Ruby

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… gratz..."
end
```

Inline "if" statements

- Original if-statement

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

- Inline if-statement

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

for-loops

- for-loops can use ranges
- Example 1:

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

- Can also use blocks (covered next week)

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

for-loops and ranges

- You may need a more advanced range for your for-loop
- Bounds of a range can be expressions
- Example:

```
for i in 1..(2*5)
    puts i
end
```

while-loops

- Can also use blocks (next week)
- Cannot use "i++"
- Example:

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

unless

- "unless" is the logical opposite of "if"

- Example:

```
unless (age >= 105)                # if (age < 105)
  puts "young."
else
  puts "old."
end
```

until

- Similarly, "until" is the logical opposite of "while"
- Can specify a condition to have the loop stop (instead of continuing)
- Example

```
i = 0
until (i >= 5)      # while (i < 5), parenthesis not required
    puts I
    i = i + 1
end
```

Methods

- Structure

  def ***method_name***(***parameter1***, ***parameter2***, **…**)

  statements

  end

- Simple Example:

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

Parameters

- No class/type required, just name them!
- Example:

```
def cumulative_sum(num1, num2)
  sum = 0
  for i in num1..num2
        sum = sum + i
  end
  return sum
end

# call the method and print the result
puts(cumulative_sum(1,5))
```

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
```

User Input

- "gets" method obtains input from a user
- Example

```
name = gets
puts "hello " + name + "!"
```

- Use chomp to get rid of the extra line

```
puts "hello" + name.chomp + "!"
```

- chomp removes trailing new lines

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"

Constants

- In Ruby, constants begin with an Uppercase
- They should be assigned a value at most once
- This is why local variables begin with a lowercase
- Example:

```
Width = 5
def square
  puts  ("*" * Width + "\n") * Width
end
```

Arrays

- Similar to PHP, Ruby arrays…
  - Are indexed by zero-based integer values
  - Store an assortment of types within the same array
  - Are declared using square brackets, [], elements are separated by commas
- Example:

```
a = [1, 4.3, "hello", 3..7]
a[0]    →       1
a[2]    →       "hello"
```

Arrays

- You can assign values to an array at a particular index, just like PHP
- 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]
```

Hashes

- Arrays use integers as keys for a particular values (zero-based indexing)
- Hashes, also known as "associative arrays", have Objects as keys instead of integers
- Declared with curly braces, {}, and an arrow, "=>", between the key and the value
- Example:

```
h = {"greeting" => "hello", "farewell" =>"goodbye"}
h["greeting"]          →       "hello"
```

Sorting

```
a = [5, 6.7, 1.2, 8]
a.sort              →    [1.2, 5, 6.7, 8]
a                   →    [5, 6.7, 1.2, 8]
a.sort!             →    [1.2, 5, 6.7, 8]
a                   →    [1.2, 5, 6.7, 8]
a[4] = "hello"      →    [1.2, 5, 6.7, 8, "hello"]
a.sort              → Error: comparison of Float with
                    String failed
h = {"greeting" => "hello", "farewell" =>"goodbye"}
h.sort  → [["farewell", "goodbye"], ["greeting", "hello"]]
```

Blocks

- Blocks are simply "blocks" of code
- They are defined by curly braces, {}, or a do/end statement
- They are used to pass code to methods and loops

Blocks

- In Java, we were only able to pass parameters and call methods
- In Ruby, we can pass code through blocks
- We saw this last week, the times() method takes a block:

```
3.times { puts "hello" }    # the block is the code in the {}
```

Blocks and Parameters

- Blocks can also take parameters
- For example, our times() method can take a block that takes a parameter. It will then pass a parameter to are block
- Example

```
3.times {|n| puts "hello" + n.to_s}
```

- Here "n" is specified as a parameter to the block through the vertical bars "|"

Yield

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

Yield

- A yield statement can also be called with parameters that are then passed to the block
- Example:

```
3.times { |n| puts n}
```

- The "times" method calls yield with a parameter that we ignored when we just printed "hello" 3 times, but shows up when we accepted a parameter in our block

Yield Examples

Code:

```
def demo_yield
    puts "BEGINNING"
    yield
    puts "END"
end
demo_yield { puts "hello" }

def demo_yield2
    puts "BEGINNING"
    yield
    puts "MIDDLE"
    yield
    puts "END"
end
demo_yield2{ puts "hello" }
```

Output:

```
BEGINNING
hello
END
```

```
BEGINNING
hello
MIDDLE
hello
END
```

Parameters, Blocks, and Yield

- Example:

```
def demo_yield3
  yield 2
  yield "hello"
  yield 3.7
end
demo_yield3 { |n| puts n * 3}
```

- "n" is the value passed by yield to the block when yield is called with arguments

Iterators

- An iterator is simply "a method that invokes a block of code repeatedly" (Pragmatic Programmers Guide)
- Iterator examples: Array.find, Array.each, Range.each
- Examples:

```
[1,2,3,4,5].find{ |n| Math.sqrt(n).remainder(1)==0}    # finds perfect square
[2,3,4,5].find{ |n| Math.sqrt(n).remainder(1)==0}      # finds perfect square
[1,2,3,4,5].each { |i| puts i }                        #prints 1 through 5
[1,2,3,4,5].each { |i| puts i * i }       #prints 1 squared, 2 squared…, 5squared
(1..5).each{ |i| puts i*i }               #prints 1 squared, 2 squared…, 5squared
```

Iterators and Loops

- Common to use iterators instead of loops
- Avoids off-by-one (OBO) bugs
- Built-in iterators have well defined behavior
- Examples

```
0.upto(5) { |x| puts x }          # prints 0 through 5
0.step(10, 2) { |x| puts x }      # 0, 2, 4, 6, 8, 10
0.step(10,3) { |x| puts x }       # 0, 3, 6, 9
```

for...in

- Similar to PHP's foreach:
  - PHP

```
$prices = array(9.00, 5.95, 12.50)
foreach($prices as $price){
   print "The next item costs $price\n"
}
```

  - Ruby

```
prices = [9.00, 5.95, 12.50]
for price in prices
   puts "The next item costs " + price.to_s
end
```

for...in

- Previous example

```
prices = [9.00, 5.95, 12.50]
for price in prices
  puts "The next item costs " + price.to_s
end
```

- Can also be written

```
prices = [9.00, 5.95, 12.50]
prices.each do |price|
  puts "The next item costs " + price.to_s
end
```

Strings

- Strings can be referenced as Arrays
- The value returned is the a Integer equivalent of the letter at the specified index
- Example:

```
s = "hello"
s[1]          →     101
s[2]          →     108
s[1].chr      →     "e"
s[2].chr      →     "l"
```

More Strings

- chomp – returns a new String with the trailing newlines removed

- chomp! – same as chomp but modifies original string

More Strings

- split(delimiter) – returns an array of the substrings created by splitting the original string at the delimiter

- slice(starting index, length) – returns a substring of the original string beginning at the "starting index" and continuing for "length" characters

Strings Examples

s = "hello world\n"

s.chomp	→	"hello world"
s	→	"hello world\n"
s.chomp!	→	"hello world"
s	→	"hello world"
s.split(" ")	→	["hello", "world"]
s.split("l")	→	["he", "", "o wor", "d"]
s.slice(3,5)	→	"lo wo"
s	→	"hello world"
s.slice!(3,5)	→	"lo wo"
s	→	"helrld"

Iterating over String characters

Code	**Output**
`"hello".each {|n| puts n}`	"hello"
`"hello".each_byte {|n| puts n}`	104 101 108 108 111
`"hello".each_byte {|n| puts n.chr}`	h e l l o

Files as Input

- To read a file, call File.open(), passing it the the path to your file
- Passing a block to File.open() yields control to the block, passing it the opened file
- You can then call gets() on the file to get each line of the file to process individually
  - This is analogous to Java's Scanner's .nextLine()

Files as Input

- Example (bold denotes variable names)

```
File.open("file.txt") do |input|   # input is the file passed to our block
  while line = input.gets           # line is the String returned from gets()
      # process line as a String within the loop
      tokens = line.split(" ")
  end
end
```

Output to Files

- To output to a file, call File.open with an additional parameter, "w", denoting that you want to write to the file

```
File.open("file.txt", "w") do |output|
  output.puts "we are printing to a file!"
end
```

Writing from one file to another

- If a block is passed, File.open yields control to the block, passing it the file.
- To write from one file to another, you can nest File.open calls within the blocks

Writing from one file to another

```
File.open("input_file.txt") do |input|
   File.open("output_file.txt", "w") do |output|
        while line = input.gets
                output.puts line
        end
   end
end
```

References

- Web Sites
  - http://www.ruby-lang.org/en/
  - http://rubyonrails.org/
- Books
  - Programming Ruby: The Pragmatic Programmers' Guide (http://www.rubycentral.com/book/)
  - Agile Web Development with Rails
  - Rails Recipes
  - Advanced Rails Recipes