Module 2
Pair Programming, Ruby

CS W169A: Software Engineering

The code for this worksheet is available at this link: here.

1 What Would Ruby Do?

Given the following snippets of Ruby code, determine the output. If you can, find a classmate, discuss, then validate your solutions by typing the code into an interpreter. You should alternate who types and who explains the output.

(i)
```
fruit1 = "strawberry"
fruit2 = "banana"
puts fruit1.reverse
puts fruit2.reverse!
fruit1 + "␣" + fruit2
```

>> yrrebwarts
>> ananab
>> strawberry ananab

(ii)
```
class String
    @@hello = "hi␣there!"
    def hello; "world"; end
end
"smoothie".hello
```

>> world

(iii)
```
class Fruit
    def method_missing(meth)
        if meth.to_s =~ \/^tastes_(.+)\?\$\/
            "Yup,␣that␣fruit␣tastes␣#{\$1}!"
        else
            super
        end
    end
end
orange = Fruit.new
orange.bitter?
orange.tastes_sour?
orange.tastes_sweet?
```

NoMethodError

>> Yup, that fruit tastes sour!
>> Yup, that fruit tastes sweet!

Note that by convention, exclamation marks in ruby method names often indicate that the method will mutate the object it's being called on. This is why fruit2 in the last line of example one returns ananab again—we called reverse! on fruit2, whereas we only called reverse on fruit1. fruit2 was mutated; fruit1 was not.

Prefixing a variable with @@ defines it as a class variable. Prefixing it with only @ defines it as an instance variable. One must create methods that interact with these variables (e.g. getter and setter methods) in order to access them. Dot notation in ruby exclusively makes method calls; there has only been one "hello" method defined in example two, and thus this is what is called.

2 Collections

In this next part, try to rewrite each of the following method as one (short) line. One person should be the writer,while the other person explains what to write. Try alternating roles between the two exercises. (Hint: see figure 3.7 in the textbook.)

(i)
```
def foo(arr)
    res = 0
    arr.each do |n|
        res += n
    end
    res
end
```

Single Line Solution: def foo(arr); arr.reduce(:+); end

Note: Ruby allows you to write multiple lines on the same line as long as they are semicolon separated.

This method takes in a sequence and returns the sum of its elements. Keep in mind, the "|n|" is not an absolute value sign, but the Ruby convention for specifying the iterator over a sequential data structure. The "reduce" method works very much like the Python version as taught in CS 61A. It takes in a binary operation (acts on 2 variables) as a parameter, and combines elements in the list with the operation.

(ii)
```
def bar(hsh)
    res = {}
    hsh.each do |k, v|
        if v > 100
            res[k] = v
        end
    end
    res
end
```

Single Line Solution: def bar(hsh); hsh.select |k, v| v > 100; end

This method takes in a dictionary and preserves any key-value pair with a value greater than 100. We can take advantage of the "select" method that is the equivalent of Python's "filter" function. The select method takes in a "block". You can think of it as a lambda function that evaluates to true or false. Any values evaluating to true are kept. Here's a guide on how to use "select".

3 Iterators

In this part, create your own iterators with the yield statement that return the following elements. Again, alternate roles between the two exercises.

(i) Write a function fib(n) that yields the first n Fibonacci numbers in sequence and returns nil.

```
>> fib(4) { |x| puts x }
1
1
2
3
nil
```

Solution:

```
def fib(n)
    prev, curr = 0, 1
    n.times do
        puts curr
        prev, curr = curr, prev + curr
    end
end
```

This solution is essentially just the iterative version of Fibonacci that you might hopefully have recalled from CS 61A. As you can see, there's all sorts of clever syntax that you can take advantage of in Ruby, like iterating with "n.times" or multiple assignments on the same line. They might look a little tricky initially, but with practice you'll get the hang of it! :)

(ii) Write the function Array#odds which yields the odd-indexed elements of the array in sequence and returns nil.

```
>> [10, 30, 50, 70, 90].odds do |n|
..   puts n
.. end
30
70
nil
```

The verbose solution:

```
class Array
    def odds
        self.each_with_index do |val, index|
            if index % 2 == 1
                yield val
            else
                next
            end
        end
        nil
    end
end
```

We can also use the "select" command we just learned and pass in a code block that determines if the index is even.

```
class Array
    def odds
        self.values_at(* self.each_index.select {|i| i.odd?})
    end
end
```

Pretty nifty!

4 Extra Practice

Implement a linked list. Try to include the add, delete, and contains operations.

This is not the most optimal implementation, and we can definitely make it look more clever, but hopefully translates well from the Python or Java implementation that you might be used to while reinforcing some Ruby syntax. Check out how instance and class variables are declared, along with some other syntactic sugar, such as the lack of parentheses.

```
class ListNode
    attr_accessor :next
    attr_reader :value
    def initialize value
        @value = value
        @next = nil
    end
end

class LinkedList
    def initialize
        @head = nil
    end

    def add value
        if @head.nil?
            @head = ListNode.new value
        else
            node = @head
            node = node.nextwhile node.next
            node.next = ListNode.new value
        end
    end

    def contains value
        node = @head
        while node
            if node.value == value
                return true
            end
            node = node.next
        end
        return false
    end

    def delete value
        if @head.value == value
            @head = @head.next
            return true
        end
```

```
        node = @head
        while node = node.next
            if node.next and node.next.value == value
                node.next = node.next.next
                return true
            end
        end
        return false
    end
end
```