ECE:5820/22C:5820, SELT
Fall 2015

HW 2: More Ruby Programming Fun

Due: Friday, Sept. 18, by 11:59 p.m. Submission instructions are provided at the end of this document

Before starting this assignment, fork or clone the HW2 repo from GitHub, at:
https://github.com/SELT/hw2fall15

Important Note: Several of these problems are moderately challenging. Do NOT leave this assignment until the last minute.

In this homework you will complete some simple programming exercises to gain familiarity with the Ruby language. Skeleton code for each part of the assignment is in the lib/ directory of the repo.

NOTE: For all questions involving words or strings, you may assume that the definition of a "word" is*: a sequence of characters whose boundaries are matched by the \b construct in Ruby regexps.*

Part 1: Strings and Regexps

(a) Write a method that determines whether a given word or phrase is a palindrome — i.e. it reads the same backwards as forwards, ignoring case, punctuation, and non-word characters. (a "non-word character" is defined for our purposes as: *a character that Ruby regexps would treat as a non-word character*.) Your solution should not use loops or iteration of any kind. You will find regular-expression syntax very useful; it is reviewed briefly in the text and the website rubular.com will allow you to try out Ruby regular expressions "live". Methods you might find useful (which you will need to look up in the Ruby documentation, at ruby-doc.org) include: `String#downcase, String#gsub, String#reverse`

Examples:
```
palindrome?("A man, a plan, a canal -- Panama")  #=> true
```

```
palindrome?("Madam, I'm Adam!")  # => true
palindrome?("Abracadabra")  # => false (nil is also ok)

def palindrome?(string)
  # your code here
end
```

(b) Given a string of input, return a hash whose keys are words in the string and whose values are the number of times each word appears. Do not use for-loops. (But iterators such as `each` are permitted.) Non-words should be ignored. Case shouldn't matter. A word is defined as a string of characters between word boundaries. (Hint: the element `\b` in a Ruby Regexp means "word boundary".)
Example:

```
count_words("A man, a plan, a canal -- Panama")
# => {'a' => 3, 'man' => 1, 'canal' => 1, 'panama' => 1, 'plan' => 1}
count_words "Doo bee doo bee doo"  # => {'doo' => 3, 'bee' => 2}

def count_words(string)
  # your code here
end
```

Part 2: Nested arrays: Rock-Paper-Scissors

In a game of rock-paper-scissors, each player chooses to play Rock (R), Paper (P), or Scissors (S). The rules are: Rock beats Scissors, Scissors beats Paper, but Paper beats Rock.

A rock-paper-scissors game is encoded as a list, where the elements are 2-element lists that encode a player's name and a player's strategy.

```
 [ [ "John", "P" ], [ "Mary", "S" ] ]
 # => returns the list ["Mary", "S"] wins since S>P
```

(a) Write a method `rps_game_winner` that takes a two-element list and behaves as follows:
- If the number of players is not equal to 2, raise WrongNumberOfPlayersError
- If either player's strategy is something other than "R", "P" or "S" (case-insensitive), raise NoSuchStrategyError
- Otherwise, return the name and strategy of the winning player. If both players use the same strategy, the first player is the winner.

We'll get you started:

```
class WrongNumberOfPlayersError < StandardError ; end
class NoSuchStrategyError < StandardError ; end
```

```
def rps_game_winner(game)
  raise WrongNumberOfPlayersError unless game.length == 2
  # your code here
end
```

(b) A rock, paper, scissors tournament is encoded as a bracketed array of games - that is, each element can be considered its own tournament.

```
[
 [
   [ ["Joe", "P"], ["Mary", "S"] ],
   [ ["Bob", "R"],  ["Alice", "S"] ]
 ],
 [
   [ ["Steve", "S"], ["Jane", "P"] ],
   [ ["Ted", "R"], ["Carol", "P"] ]
 ]
]
```

Under this scenario, Mary would beat Joe (S>P), Bob would beat Alice (R>S), and then Mary and Bob would play (Bob wins since R>S); similarly, Steve would beat Jane, Carol would beat Ted, and Steve and Carol would play (Steve wins since S>P); and finally Bob would beat Steve since R>S, that is, continue until there is only a single winner.

- Write a method `rps_tournament_winner` that takes a tournament encoded as a bracketed array and returns the winner (for the above example, it should return `[“Bob”,“R”]`).
- Tournaments can be nested arbitrarily deep, i.e., it may require multiple rounds to get to a single winner. You can assume that the initial array is well formed (that is, there are 2^n players, and each one participates in exactly one match per round).

Part 3: Basic OOP in Ruby

(a) Create a class `Dessert` with getters and setters for instance variables `@name` and `@calories`. Define instance method `healthy?`, that returns `true` if a dessert has less than 200 calories, and instance method `delicious?`, that returns `true` for all desserts.

(b) Create a class `JellyBean` that extends `Dessert`, and add a getter and setter for instance variable `@flavor`. Override `delicious?` to return `false` if the flavor is `“black licorice”` (but `delicious?` should still return `true` for all other flavors and for all non-JellyBean desserts).

Here is the framework (you may define additional helper methods):

```
class Dessert
  #remember, you need to define getters and setters for
  #instance variables @name and @calories
  def initialize(name, calories)
    # YOUR CODE HERE
  end

  def healthy?
    # YOUR CODE HERE
  end

  def delicious?
    # YOUR CODE HERE
  end
end

class JellyBean < Dessert
  #Remember, you need a getter and setter for instance var. @flavor
  def initialize(name, calories, flavor)
    # YOUR CODE HERE
  end

  def delicious?
    # YOUR CODE HERE
  end
end
```

Part 4: Metaprogramming and open classes

In lecture we saw how `attr_accessor` uses metaprogramming to create getters and setters for object attributes on the fly.
Define a method `attr_accessor_with_history` that provides the same functionality `as attr_accessor` but also tracks every value the attribute has ever had:

```
class Foo
  attr_accessor_with_history :bar
end

f = Foo.new     # => #<Foo:0x127e678>
f.bar = 3       # => 3
f.bar = :wowzo  # => :wowzo
f.bar = 'boo!'  # => 'boo!'
f.bar_history # => [nil, 3, :wowzo, 'boo!']
```

Here are some important hints and things to notice to get you started:

1. The first thing to notice is that if we define `attr_accessor_with_history` in class `Class`, we can use it as in the snippet above. This is because, in Ruby, a class is simply an object of class `Class`. (If that makes your brain hurt, just don't worry about it for now. It'll come.)
2. The second thing to notice is that Ruby provides a method `class_eval` that takes a string and evaluates it in the context of the current class, that is, the class from which you're calling `attr_accessor_with_history`. This string will need to contain a method definition that implements a setter-with-history for the desired attribute `attr_name`.
3. #bar_history should always return an Array of elements, even if no values have been assigned yet.

- Don't forget that the very first time the attribute receives a value, its history array will have to be initialized.
- Don't forget that instance variables are referred to as @bar within getters and setters, as Section 3.4 of the text explains.
- Although the existing attr_accessor can handle multiple arguments (e.g. `attr_accessor :foo, :bar`), your version just needs to handle a single argument. However, it should be able to track multiple instance variables per class, with any legal class names or variable names, so it should work if used this way:
  ```
  class SomeOtherClass
   attr_accessor_with_history :foo
   attr_accessor_with_history :bar
  end
  ```
- History of instance variables should be maintained separately for each object instance. That is, if you do
  ```
  f = Foo.new
  f.bar = 1
  f.bar = 2
  f = Foo.new
  f. bar = 4
  f.bar_history
  ```
  then the last line should just return `[nil,4]`, rather than `[nil,1,2,4]`

Here is the skeleton to get you started:

```
class Class
  def attr_accessor_with_history(attr_name)
    attr_name = attr_name.to_s    # make sure it's a string
    #Use the existing attr_reader method to create the attribute’s
    # getter method
    attr_reader attr_name
    #Use the existing attr_getter method to create a getter method
```

```
    #for the @<attr_name>_history instance variable, which will hold
    #the attribute's history, as an array
    attr_reader attr_name+"_history"
     #Now for the interesting part. You need to define a setter
     #method named <attr_name>= that sets the attribute and appends
     #the  set value to the <attr_name>_history array
    class_eval "your code here, use %Q for multiline strings"
  end
end

class Foo
  attr_accessor_with_history :bar
end
f = Foo.new
f.bar = 1
f.bar = 2
f.bar_history # => if your code works, should be [nil,1,2]
```

Part 5. iterators, blocks, yield

Given two collections (of possibly different lengths), we want to get the Cartesian product of the sequences—in other words, every possible pair of N elements where one element is drawn from each collection.
For example, the Cartesian product of the sequences **`a==[:a,:b,:c]`** and **`b==[4,5]`** is:
`a×b == [[:a,4],[:a,5],[:b,4],[:b,5],[:c,4],[:c,5]]`

Create a method that accepts two sequences and ***returns an iterator*** that will yield the elements of the Cartesian product, one at a time, as a two-element array.

- It doesn't matter what order the elements are returned in. So for the above example, the ordering **`[[:a,4], [:b,4], [:c,4], [:a,5], [:b,5], [:c,5]]`** would be correct, as would any other ordering.
- It ***does matter*** that within each pair, the order of the elements matches the order in which the original sequences were provided. That is, **`[:a,4]`** is a member of the Cartesian product a×b, but **`[4,:a]`** is not. (Although **`[4,:a]`** is a member of the Cartesian product b×a.]

To start you off, skeleton code, showing possible correct results, is shown below

```
class CartesianProduct
  include Enumerable
  # your code here
end

#Examples of use
c = CartesianProduct.new([:a,:b], [4,5])
```

```
c.each { |elt| puts elt.inspect }
# [:a, 4]
# [:a, 5]
# [:b, 4]
# [:b, 5]

c = CartesianProduct.new([:a,:b], [])
c.each { |elt| puts elt.inspect }
# (nothing printed since Cartesian product
# of anything with an empty collection is empty)
```

Since we will be using automated tools to grade your homework assignments, it is important that you carefully follow these submission instructions:

Submission Instructions (Follow these EXACTLY):

1. Each part of the assignment must be in a separate file. All of the files must be in the /lib directory of your hw2fall15 repo. The files must be named `part1.rb`, `part2.rb` etc.
2. **The files should contain only the ruby code specified in the assignment--i.e. all test code and/or other extraneous code should be stripped out of the file before submission.** As described in the header of the lib/part1.rb file, you can run sanity checks on your ruby files before submitting them. This will insure that your files are compatible with our automatic grading tools. Passing the sanity checks does NOT insure that your solution is correct.
3. In the Workspace menu on the left side of your Cloud9 environment, right-click the folder icon for your hw2fall15/lib folder.
4. From the displayed menu select 'Download'. This will download your lib/folder to your host machine as a tar file. Save this file on your host machine. Do NOT uncompress it. If you host computer is a Windows machine the downloaded file should have a .tar extension. If you host computer is a Mac, this extension may be missing. If the .tar extension is missing, rename the file to add it--i.e. The file should be named lib.tar
5. Upload this single file to the HW2 drop-box on the class ICON site