HW 1: Ruby calisthenics

In this homework you will do some simple programming exercises to get familiar with the Ruby language. We will provide detailed automatic grading of your code.

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: fun with strings

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

Suggestion: once you have your code working, consider making it more beautiful by using techniques like method chaining, as described in ELLS 3.2.

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. Don't use for-loops. (But iterators like `each` are permitted.) Nonwords should be ignored. Case shouldn't matter. A word is defined as a string of characters between word boundaries. (Hint: the sequence `\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: 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.

```
 [ [ "Armando", "P" ], [ "Dave", "S" ] ]
 # => returns the list ["Dave", "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.

```
[
 [
   [ ["Armando", "P"], ["Dave", "S"] ],
   [ ["Richard", "R"],  ["Michael", "S"] ],
 ],
 [
   [ ["Allen", "S"], ["Omer", "P"] ],
   [ ["David E.", "R"], ["Richard X.", "P"] ]
 ]
]
```

Under this scenario, Dave would beat Armando (S>P), Richard would beat Michael (R>S), and then Dave and Richard would play (Richard wins since R>S); similarly, Allen would beat Omer, Richard X would beat David E., and Allen and Richard X. would play (Allen wins since S>P);

and finally Richard would beat Allen 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 `["Richard", "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: anagrams

An anagram is a word obtained by rearranging the letters of another word. For example, "rats", "tars" and "star" are an anagram group because they are made up of the same letters.

Given an array of strings, write a method that groups them into anagram groups and returns the array of groups. Case doesn't matter in classifying string as anagrams (but case should be preserved in the output), and the order of the anagrams in the groups doesn't matter.

Example:

```
# input:
['cars', 'for', 'potatoes', 'racs', 'four','scar', 'creams', 'scream']
#  => output:  [["cars", "racs", "scar"], ["four"], ["for"], ["potatoes"],
["creams", "scream"]]
# HINT: you can quickly tell if two words are anagrams by sorting their
#  letters, keeping in mind that upper vs lowercase doesn't matter

def combine_anagrams(words)
  #   <YOUR CODE HERE>
end
```

Part 4: Basic OOP

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

(b) Create a class JellyBean that extends Dessert, and add a getter and setter for flavor. Modify 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
  def initialize(name, calories)
    # YOUR CODE HERE
  end

  def healthy?
    # YOUR CODE HERE
  end

  def delicious?
    # YOUR CODE HERE
  end
end

class JellyBean < Dessert
  def initialize(name, calories, flavor)
    # YOUR CODE HERE
  end

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

Part 5: advanced OOP,metaprogramming, open classes and duck typing

(Exercise 3.4 from ELLS)
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!']
```

We'll start you off. Here are some 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, as ELLS mentions, 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`.

- 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 ELLS 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
    attr_reader attr_name        # create the attribute's getter
    attr_reader attr_name+"_history" # create bar_history getter
    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]
```

a) [ELLS ex. 3.11] Extend the currency-conversion example from lecture so that you can write
`5.dollars.in(:euros)`
`10.euros.in(:rupees)`
etc.

- You should support the currencies 'dollars', 'euros', 'rupees' , 'yen' where the conversions are: rupees to dollars, multiply by 0.019; yen to dollars, multiply by 0.013; euro to dollars, multiply by 1.292.
- Both the singular and plural forms of each currency should be acceptable, e.g. `1.dollar.in(:rupees)` and `10.rupees.in(:euro)` should work.

You can use the code shown in lecture as a starting point if you wish; it is shown below and is also available at pastebin http://pastebin.com/agjb5qBF

```
class Numeric
  @@currencies = {'yen' => 0.013, 'euro' => 1.292, 'rupee' => 0.019}
  def method_missing(method_id)
    singular_currency = method_id.to_s.gsub( /s$/, '')
    if @@currencies.has_key?(singular_currency)
      self * @@currencies[singular_currency]
    else
      super
    end
  end
end
```

b) Adapt your solution from the "palindromes" question so that instead of writing `palindrome?("foo")` you can write `"foo".palindrome?` HINT: this should require fewer than 5 lines of code.

c) Adapt your palindrome solution so that it works on Enumerables. That is:
`[1,2,3,2,1].palindrome?    # => true`
(It's not necessary for the collection's elements to be palindromes themselves--only that the top-level collection be a palindrome.) HINT: this should require fewer than 5 lines of code. Although hashes are considered Enumerables, your solution does not need to make sense for hashes (though it should not error).

Part 6. 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, here is a pastebin link to skeleton code(http://pastebin.com/cgSuhtPf) showing possible correct results. For your convenience the code is also 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)
```