Homework 1 taught us the Ruby Object-Oriented Scripting language, iterators, metaprogramming and duck-typing.

Problem 1 had the theme "fun with strings". Part A involved writing a method that determines whether a given string (word or phrase) is a palindrome, that it reads the same backwards as forwards, ignoring case, punctuation, and non-word characters. Our solution could not use loops or iteration but instead used regular expressions. Part B defined a wordCount function, that given an input string, returned a hash whose keys are words in the string and whose values are the number of times each word appears. We could not use for loops but instead had to use iterators like each. Non-words and case had to be ignored. A word was defined as a string of characters between word boundaries (\b in ruby regular expression means word boundary).

Problem 2 was a Rock-Paper-Scissors game (RPS). Each player chose to play Rock (R), Paper (P), or Scissors (S). The rules were R beats S; S beats P; and P beats R. The Rock-Paper-Scissors game was encoded as a list, where the elements are themselves 2-element lists that encode a player's name and a player's selected move. For example: [ [ "Armando", "P"], ["Dave", "S"]] - Dave would win since S > P. Part A had us write a method rps\_game\_winner that takes a 2-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-sensitive), raise NoSuchStrategyError. Otherwise, return the name of the winning player. If both players play the same move, the first player is the winner. Part B: had us define a rock-paper-scissors tournament to be an array of games in which each player always plays the same move. A rock-paper-scissors tournament is encoded as a bracketed array of games.

[

[

[ [“Armando”, “P”], [“Dave”, “S”] ],

[ [“Richard”, “R”], [“Michael”, “S”] ],

],

[

[ [“Allen”, “S”], [“Omer”, “P”] ],

[ [“David E.”, “R”], [“Richard X.”, “P”] ]

]

]

In the above tournament, Armando will always play “P” and Dave will always play “S”. The tournament plays out as follows. Dave would beat Armand (S>P). Richard would beat Michael (R>S). The Dave and Richard would play (Richard wins since R>S). Similarly, Allen would beat Omer (S>P).

Richard X. woul beat David E. (P>R). Allen plays Richard X. (Allen wins since S>P). Finally, Richard beats Allen, since (R>S). The tournament continues until there is only a single winner. Tournaments can be nested arbitrarily deep, i.e., it may require multiple rounds to get a single winner. We could assume that the initial tournament was well-formed, that there is 2\*\*n players, and each one participates in exactly one match per round. We had to write a method, rps\_tournament\_winner that took a tournament encoded as a bracketed array and returns the winner. For the above example, it would return: [“Richard”, “R”]. A recursive solution seemed to work best.

Part 3 was a problem about anagrams. An anagram is a word obtained by rearranging the letters of another word. For example, “rats”, “tars” and “star” are anagrams of one another, as are “dictionary” and “indicatory”. We will call any array of single word anagrams an anagram group. For instance, [“rats”, “tars”, “star”] is an anagram group, as is [“dictionary”]. We wrote a method, combine\_anagrams(words), that given an array of strings, words, groups the input into anagram groups. Case did not matter in classifying strings as anagrams (but case should be preserved in the output), and the order of the anagrams in the groups does not matter. The output should be an array of anagrams (i.e. an array of arrays). A hint given was: You can quickly tell if 2 words are anagrams by sorting their letters, keeping in mind that upper vs. lower case does not matter.

Part 4 was basic object oriented programming in Ruby. Part A: We had to create a class Dessert with getters and setters for name and calories that were part of the Dessert class. We had to define instance methods healthy? which returned true if less than 200 calories, and delicious? which returns true for all desserts. Part B: We had to create a class JellyBean that extended Dessert, and add a getter and setter for flavor. We modified 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). We were allowed to use helper methods.

Part 5 covered advanced object-oriented programming (OOP), open classes and duck-typing. In lecture we saw that attr\_accessor uses metaprogramming to create getters and setters for object attributes on the fly. We had to design a method attr\_accessor\_with\_history that provided the same functionality as attr\_accessor but also tracks every value the attribute has ever taken. The following example shows the basic behavior of the new accessor.

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!']

Some hints and guidelines for us were:

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, a class in Ruby is simply an object of class Class.

The second thing to notice is that Ruby provides a method class\_eval that takes a string and evalulates 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.

#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 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 explained in our text book.

Although the existing attr\_accessor can handle multiple arguments (e.g. attr\_accessor :foo, :bar), our version just needed 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. Thus 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 return [nil, 4] rather than [nil, 1, 2, 4].

The skeleton code given to get us started was:

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 multiple strings”

end

end

class Foo

attr\_accessor\_with\_history :bar

end

Example Test Case:

f = Foo.new

f.bar = 1

f.bar = 2

f.bar\_history # => if your code works, should be [nil, 1, 2].

Part 6 was Advanced Object-oriented Programming (OOP), Metaprogramming, open classes and duck typing. Part A was a currency conversion. We had to extend the currency conversion example from lecture so that code such as the following will work.

5.dollars.in(:euros)

10.euros.in(:rupees)

The currencies we had to support were: dollars, euros, rupees, yen where the conversions were:

rupees to dollars, multiply by 0.019

yen to dollars, multiply by 0.013

euro to dollars, multiply by 1.292

Both singular and plural forms of each currency had to be acceptable, e.g. 1.dollar.in(:rupees) and 10.rupees.in(:euro) should work.

Starter code was:

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

Part B was working with Palindromes again. We adapted our solution from problem 1 part A on palindromes to: instead of writing palindrome?(“foo”) you can write “foo”.palindrome? We were given a hint that it should require fewer than 5 lines of code.

Part C was working with Palindromes, once again. We had to adapt the Palindrome solution so that it worked on Enumerables. That is: [1,2,3,2,1].palindrome? # =? true. It was not necessary that the collection's elements to be palindromes themselves—only that the top-level collection be a palindrome. A hint given was: this should require fewer than 5 lines of code. Although hashes are considered Enumerables, our solution did not need to work with hashes, though it should not error.

Problem 7 of homework 1 covered iterators, blocks and yield in Ruby. Given 2 collections (of possible different lengths), we want to get the Cartesian product of the sequences. A Cartesian product is a sequence that enumerates every possible pair from 2 collections, where the pair consists of one element from each collection. For example, the Cartesian product (denoted by x) of the sequences:

a = [:a, :b, :c] and b = [4, 5] is: a x b = [ [:a, 4], [:a, 5], [:b, 4], [:b, 5], [:c, 4], [:c, 5] ]. We had to create a constructor for the class CartesionProduct that takes 2 sequences as arguments, these values will define the behavior of your object. Define each as an instance method for CartesionProduct. Your method should return an iterator which yields the cartesion product of the 2 sequences used in the class's constructor. The iterator should yield one of the values one at a time as 2 element arrays.

It does not 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 well.

It does not matter 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 axb but [4, :a] is not. (Instead, [4, :a] is a member of the Cartesion Product bxa).

The skeleton code was:

class CartesionProduct

include Enumerable

#our code here

end

Some test cases were:

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