Homework 1

1. Check the documentation on [Array](http://www.ruby-doc.org/core-1.9.3/Array.html), [Hash](http://www.ruby-doc.org/core-1.9.3/Hash.html) and [Enumerable](http://www.ruby-doc.org/core-1.9.3/Enumerable.html) as they could help tremendously with these exercises. :-)

* Define a method sum which takes an array of integers as an argument and returns the sum of its elements. For an empty array it should return zero.
* Define a method max\_2\_sum which takes an array of integers as an argument and returns the sum of its two largest elements. For an empty array it should return zero. For an array with just one element, it should return that element.
* Define a method sum\_to\_n? which takes an array of integers and an additional integer, n, as arguments and returns true if any two elements in the array of integers sum to n. An empty array should sum to zero by definition.

2.Check the documentation on [String](http://www.ruby-doc.org/core-1.9.3/String.html) and [Regexp](http://www.ruby-doc.org/core-1.9.3/Regexp.html) as they could help tremendously with these exercises. :-)

* Define a method hello(name) that takes a string representing a name and returns the string "Hello, " concatenated with the name.
* Define a method starts\_with\_consonant?(s) that takes a string and returns true if it starts with a consonant and false otherwise. (For our purposes, a consonant is any letter other than A, E, I, O, U.)**NOTE:** be sure it works for both upper and lower case and for nonletters!
* Define a method binary\_multiple\_of\_4?(s) that takes a string and returns true if the string represents a binary number that is a multiple of 4. **NOTE:** be sure it returns false if the string is not a valid binary number!

3.Define a class BookInStock which represents a book with an isbn number, isbn, and price of the book as a floating-point number, price, as attributes. The constructor should accept the ISBN number (a string) as the first argument and price as second argument, and should raise ArgumentError (one of Ruby's built-in exception types) if the ISBN number is the empty string or if the price is less than or equal to zero.

Include the proper getters and setters for these attributes. Include a method price\_as\_string that returns the price of the book with a leading dollar sign and trailing zeros, that is, a price of 20 should display as "$20.00" and a price of 33.8 should display as "$33.80".

4: RUBY CALISTHENICS

First off, please download the [Code skeleton](https://courses.edx.org/c4x/BerkeleyX/CS.CS169.1x/asset/public.zip) required for this homework

The goal of this multi-part assignment is to get you accustomed to basic Ruby coding and introduce you to RSpec, the unit testing tool we will be using heavily.

While we provide an explanation of how your code should work in this handout, you should get accustomed to the idea that the true specification is in the test files!

Therefore, we suggest you work on this assignment using autotest, which automatically re-runs all the RSpec tests each time you make changes to your code:

* + 1. In a terminal window, change to the root directory of this homework (the one containing subdirectories lib/ and spec/) and run the command autotest. RSpec expects to find code files under lib/ and the corresponding spec files under spec/.
    2. Initially, all tests are marked "pending", as indicated by the argument :pending => true in each describe block. To start working on a question, remove this option:

e.g. in fun\_with\_strings\_spec.rb, change:

describe 'palindrome detection', :pending => true do

to:

describe 'palindrome detection' do

and save the spec file. autotest will detect the change and automatically re-run the tests in that group, which will now fail (displayed in red) since you haven't written any code yet.

As you fill in code in the appropriate files under lib/, each time you save changes to that file the tests will automatically be re-run. When a test passes, it's displayed in green. Your goal is to get all tests for all parts to pass green.

4-1: FUN WITH STRINGS

Specs: spec/fun\_with\_strings\_spec.rb

In this problem, you'll implement three functions that perform basic string processing. You can start from the template fun\_with\_strings.rb

***Part A — Palindromes***:

A palindrome is a word or phrase that reads the same forwards as backwards, ignoring case, punctuation, and nonword characters. (A "nonword character" is defined for our purposes as "a character that Ruby regular expressions would treat as a nonword character".)

You will write a method palindrome? that returns true if and only if its receiver is a palindrome.

As you can see in the template fun\_with\_strings.rb, we arrange to mix your method into the Stringclass so it can be called like this:

"redivider".palindrome? # => should return true "adam".palindrome? # => should return false or nil

Your solution shouldn't use loops or iteration of any kind. Instead, you will find regular-expression syntax very useful; it's reviewed briefly in the book, and the website [rubular.com](http://www.rubular.com/) lets you try out Ruby regular expressions "live". Some methods that you might find useful (which you'll have to look up in Ruby documentation, [ruby-docorg](http://ruby-doc.org/)) include: String#downcase, String#gsub, String#reverse.

The spec file contains a number of test cases. At a minimum, all should pass before you submit your code. We may run additional cases as well.

***Part B — Word Count***: Define a function count\_words that, given an input string, return a hash whose keys are words in the string and whose values are the number of times each word appears:

"To be or not to be" # => {"to"=>2, "be"=>2, "or"=>1, "not"=>1}

Your solution shouldn't use for-loops, but iterators like each are permitted. As before, nonwords and case should be ignored. A word is defined as a string of characters between word boundaries.

***Part C — Anagrams***:

An anagram group is a group of words such that any one can be converted into any other just by rearranging the letters. For example, "rats", "tars" and "star" are an anagram group.

Given a space separated list of words in a single string, 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.

4-2: BASIC OBJECT-ORIENTED PROGRAMMING FOR DESSERT

Specs: spec/dessert\_spec.rb

1. Create a class Dessert with getters and setters for name and calories. The constructor should accept arguments for name and calories.
2. Define instance methods healthy?, which returns true if and only if a dessert has less than 200 calories, and delicious?, which returns true for all desserts.
3. Create a class JellyBean that inherits from Dessert. The constructor should accept a single argument giving the jelly bean's flavor; a newly-created jelly bean should have 5 calories and its name should be the flavor plus "jelly bean", for example, "strawberry jelly bean".
4. Add a getter and setter for the flavor.
5. Modify delicious? to return false if the flavor is licorice, but true for all other flavors. The behavior of delicious? for non-jelly-bean desserts should be unchanged.

4-3: ROCK PAPER SCISSORS

 (100 points possible)

Specs: spec/rock\_paper\_scissors\_spec.rb

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

In a round of rock-paper-scissors, each player's name and strategy is encoded as an array of two elements

[ ["Armando", "P"], ["Dave", "S"] ] # Dave would win since S > P

***1. Game Winner***:

Create a RockPaperScissors class with a class method winner that takes two 2-element arrays like those above, and returns the one representing the winner:

RockPaperScissors.winner(['Armando','P'], ['Dave','S']) # => ['Dave','S']

If either player's strategy is something other than "R", "P" or "S" (case-insensitive), the method should raise aRockPaperScissors::NoSuchStrategyError exception.

If both players use the same strategy, the first player is the winner.

***2. Tournament***:

A rock-paper-scissors tournament is encoded as an 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"] ] ] ]

In the tournament above Armando will always play P and Dave will always play S. This tournament plays out as follows:

Under this scenario, Dave would beat Armando (S>P) and Richard would beat Michael (R>S), so 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, pairwise play continues until there is only a single winner.

Write a method `RockPaperScissors.tournament\_winner' that takes a tournament encoded as an array and returns the winner (for the above example, it should return ['Richard', 'R']). You can assume that the array is well formed (that is, there are 2^n players, and each one participates in exactly one match per round).

HINT: Formulate the problem as a recursive one whose base case you solved in part 1.

4-4: RUBY METAPROGRAMMING

 (100 points possible)

Specs: spec/attr\_accessor\_with\_history\_spec.rb

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 f.bar = 3 # => 3 f.bar = :wowzo # => :wowzo f.bar = 'boo!' # => 'boo!' f.bar\_history # => [nil, 3, :wowzo]

(Calling bar\_history before bar's setter is ever called should return nil.)

History of instance variables should be maintained separately for each object instance. that is:

f = Foo.new f.bar = 1 ; f.bar = 2 g = Foo.new g.bar = 3 ; g.bar = 4 g.bar\_history

then the last line should just return [nil,3], rather than [nil,1,3].

If you're interested in how the template works, the first thing to notice is that if we defineattr\_accessor\_with\_history in class Class, we can use it as in the snippet above. This is because a Ruby class like Foo or String is actually just an object of class Class. (If that makes your brain hurt, just don't worry about it for now. It'll come.)

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.

HINTS:

* Don't forget that the very first time the attribute receives a value, its history array will have to be initialized.
* An attribute's initial value is always nil by default, so if foo\_history is referenced before foo has ever been assigned, the correct answer is nil, but after the first assignment to foo, the correct value for foo\_history would be [nil].
* Don't forget that instance variables are referred to as @bar within getters and setters, as Section 3.4 of the ESaaS textbook 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.
* Your implementation should be general enough to work in the context of any class and for attributes of any (legal) variable name