**Ruby Intro**

This homework is made up of two sections comprised of multiple parts and gives some basic practice in Ruby as well as getting you accustomed to making testing a regular part of your workflow.

GET STARTED EARLY it’s a \*\*LONG \*\*assignment. Ask lots of questions along the way!! We will be working through this material in lecture, but you can get started by reading Chapter 3 and trying out some of the tutorials listed at the end of this assignment.

REMEMBER that each Homework assignment (there are 2) are worth 7% of your overall grade, and the homeworks are here to help you get started, set up your environment, and learn the basic tools that you will need for your project. Step one: “learn” Ruby- easy right? No. Jump in as soon as possible and make sure you understand everything you’re doing along the way- ask if you don’t! Use Basecamp to your advantage!

Good luck!

**Learning Goals**

After completing this assignment, you will know how to:

* Write simple code that uses basic constructs in the Ruby language, including methods and arguments, conditionals, string and array manipulation, regular expressions, and basic object-oriented programming mechanisms
* Understand the Ruby project conventions for where code files and test files are located in a project's directory hierarchy
* Run individual tests or suites of tests using the RSpec unit testing tool
* Understand the basic syntax of RSpec unit tests and write your own
  + Note that we will talk about other options that can be used instead of Rspec when you begin your projects.

**Setup**

To get to your coding environment, go to https://{ContainerName}-18-cs4300and5300-1-uccs.devedu.io where ContainerName is your UCCS username (for example: [https://kwalcott-18-cs4300and5300-1-uccs.devedu.io](https://kwalcott-18-cs4300and5300-1-uccs.devedu.io/)). You will be asked for a username and password. Username is your UCCS username, and password is your UCCS student number. This will take you to a Cloud 9 environment. To get started, openb a terminal. In the upper left-hand corner, click on the hamburger (the 3 lines), View, Terminal. And you're set!

To work more fully from the terminal, install your editor of choice. For example: apt-get update && apt-get install vim or apt-get update && apt-get install emacs. To set this editor as default for all applications, export EDITOR=vi or export EDITOR=emacs.

**Commit Work Frequently**

As you edit your work, commit your code to your repository often!! After you have cloned your repository in the step below, you will be able to commit your code from anywhere within the homework directory to your private class repository.

To commit and push:

git status \\Check what files have changed AND are staged for commit. If files are missing that you want staged for commit.

If files need to be added, git add fileOrFolderName. Run git status to double check.

git commit -m "added else condition for if <0" \\Give a descriptive message for your commit.

git push origin main \\Push your code to the main branch of your remote repo named origin (in this case, you'll already have it set up to be <https://github.com/UCCS-CS4300-5300/YOURUCCSUSERNAME_GOES_HERE.git>)

Note that if you've been away for awhile or there's a possibility that code has changed in your remote repository, use git pull origin main to pull from the repo.

When running ANY git command, be sure to read all output for warnings/errors- oftentimes, the output will tell you what you need to do.

**SECTION 1: RUBY BASICS**

Clone your GitHub repo into a homework directory by running:

git clone https://github.com/UCCS-CS4300-5300/YOURUCCSUSERNAME\_GOES\_HERE.git homework

If you have done this correctly you should now see a homework directory under /root/environment/homework/.

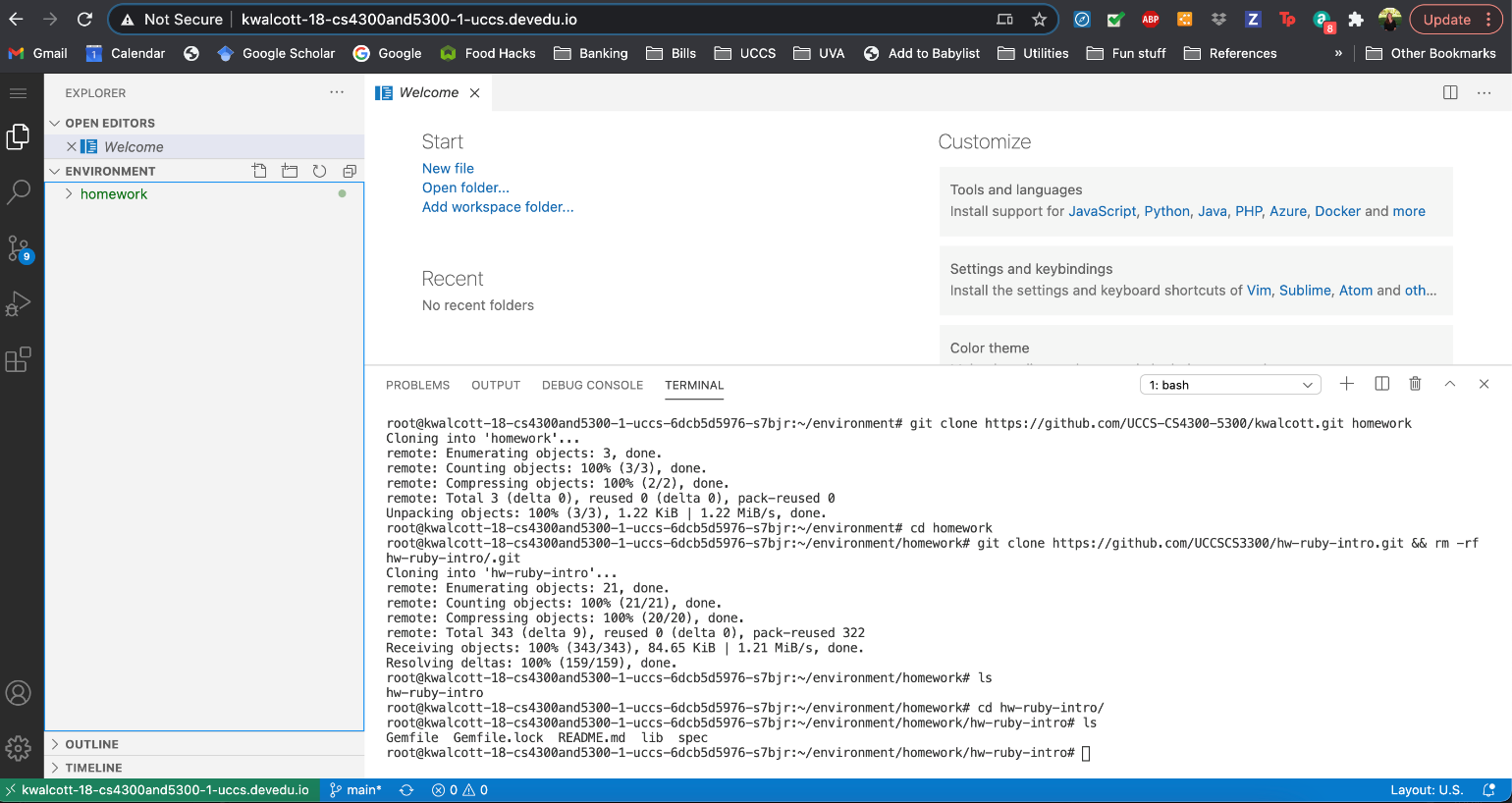
Go to your homework directory (if not there already)

cd homework

then clone the skeleton code at <https://github.com/UCCSCS3300/hw-ruby-intro> to your Cloud9 docker container. To do this, run:

git clone https://github.com/UCCSCS3300/hw-ruby-intro.git && rm -rf hw-ruby-intro/.git

in the terminal of Cloud9 running inside of the docker container. This creates a folder called hw-ruby-intro, which contains a skeleton for part 1 of your assignment, described below.

[](https://raw.githubusercontent.com/UCCS-CS4300-5300/kwalcott-public/main/gettingStarted.png)

**Overview**

The repo for this assignment follows a fairly standard Ruby convention for codebases: **the code files are stored in lib/ and the test files are stored in spec/**. (We use the RSpec unit-testing framework; if we were using Ruby's default framework, known as Test::Unit, the test files would be under test/.)

*Note: There is no autograder being used for this course. Please ignore the comment on the top of the ruby\_intro.rb file.*

You SHOULD do your own testing. Test each of the 3 parts separately, or together. The files spec/part[123]\_spec.rb contain RSpec tests for each of the three parts. For example, to test your answers to Part 1, say rspec ./spec/part1\_spec.rb. rspec with no arguments runs the tests in all the files spec/\*\_spec.rb.

* The line numbers in the RSpec error report will give you guidance as to which tests failed. (You can check the[RSpec documentation](http://rspec.info/) to see how the .rspec file can be used to customize the output format.)

To ensure you have the rspec gem installed, you need bundler and can then run bundle install like so:

$ cd /root/environment/homework/hw-ruby-intro

$ bundle install

$ rspec spec/ or rspec spec/part#\_spec.rb

When the above completes successfully you'll have RSpec installed and can run rspec from the command line to test your code.

LOOK AT THE TESTS IN THE SPEC FOLDER!! Right now, all tests will fail. Your goal is to make all tests pass once you complete the code.

**1. Arrays, Hashes, and Enumerables**

Check the[Ruby 2.x documentation](http://ruby-doc.org/) on Array, Hash and Enumerable as they could help tremendously with these exercises!

1. Define a method sum(array) that takes an array of integers as an argument and returns the sum of its elements. For an empty array it should return zero. Run associated tests via: $ rspec ./spec/part1\_spec.rb:5
2. Define a method max\_2\_sum(array) 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. Run associated tests via: $ rspec ./spec/part1\_spec.rb:24
3. Define a method sum\_to\_n?(array, n) that 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. sum\_to\_n?([], n) should return false for any value of n, by definition. Run associated tests via: $ rspec ./spec/part1\_spec.rb:43

You can check your progress on the **all** the above by running $ rspec ./spec/part1\_spec.rb.

**2. Strings and Regular Expressions**

Check the documentation on String and Regexp as they could help a lot with these exercises!

1. Define a method hello(name) that takes a string representing a name and returns the string "Hello, " concatenated with the name. Run associated tests via: $ rspec -e '#hello' spec/part2\_spec.rb
2. 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! Run associated tests via: $ rspec -e '#starts\_with\_consonant?' spec/part2\_spec.rb
3. 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! Run associated tests via: $ rspec -e '#binary\_multiple\_of\_4?' spec/part2\_spec.rb

You can check your progress on the all the above by running $ rspec ./spec/part2\_spec.rb.

**3. Object Oriented Basics**

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. Run associated tests via: $ rspec -e 'getters and setters' spec/part3\_spec.rb

The constructor should accept the ISBN number (a string, since in real life ISBN numbers can begin with zero and can include hyphens) 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. Run associated tests via: $ rspec -e 'constructor' spec/part3\_spec.rb

Include a method price\_as\_string that returns the price of the book formatted with a leading dollar sign and two decimal places, that is, a price of 20 should format as "$20.00" and a price of 33.8 should format as "$33.80". Run associated tests via: $ rspec -e '#price\_as\_string' spec/part3\_spec.rb

You can check your progress on the all the above by running rspec ./spec/part3\_spec.rb.

**More Challenges**

* Try pairing using the[one-undermanship pair programming style](http://www.agileventures.org/remote-pair-programming/pair-programming-protocols)

**SECTION 2: BEYOND THE BASICS**

For this set of code, another skeleton is provided. To get the skeleton code:

cd /root/environment/homework

git clone https://github.com/UCCSCS3300/hw-ruby-more.git && rm -rf hw-ruby-more/.git

cd hw-ruby-more

The goal of this multi-part assignment is to get you accustomed to more advanced Ruby coding and introduce you more to RSpec, the unit testing tool we will be using heavily throughout the semester. In Part 1, you ran rspec to check your code. In Part 2, you’ll look at and analyze the test code and write some test code yourself to backup your Ruby code-- note the file structure and coding structure used in Rspec and figure out what it’s doing! You’ll be writing it all on your own later. The tests you’re writing in this assignment are all UNIT LEVEL tests.

1. Run the command `bundle install`.

2. \*\*Initially, all tests are marked "pending", as indicated by the argument <code>:pending => true</code> in each <code>describe</code> block. To start working on a question, remove this option:</strong>

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. Run rspec and the modified tests will now fail (displayed in red) since you haven't written any code yet.

Note: Removing pending and having an EMPTY test will automatically make the test pass… A test runs test input, compares the output of what’s expected and what’s returned and gives a result. With nothing inside the test method, it passes, but it tests nothing. Write good test code!

Using the :pending => true option allows you to note code that you intend to write but haven’t yet. It won’t try to run the test code inside until you’re ready.

As you fill in code in the appropriate files under lib/, rerun rspec. When a test passes, it's displayed in green. Your goal is to get all tests for all parts to pass green.

(Note: when you submit your assignment, we will also run additional test cases beyond the ones given here.)

To help you get started with this section, check out this video by Samuel Joseph at Hawaii Pacific University: <https://www.youtube.com/watch?v=bII2Nxc6po4>

**Part 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

\*\*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."

**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 String class 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-doc.org](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 will run additional cases as well.

**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.

**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.

Select one of the rspec tests, copy it, and modify it so that it tests a feature of the string manipulation library you have implemented. Alternatively, you can write your own test.

Finally, create a text file and summarize the purpose of each rspec test (those that existed already AND your additional test(s)).

**Part 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.

Select one of the rspec tests, copy it, and modify it so that it tests a feature of the dessert library you have implemented. Alternatively, you can write your own test.

Finally, go to your text file and summarize the purpose of each rspec test (including your added test).

**Part 3: Rock Paper Scissors**

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 other words, Rock wins over Scissors, Scissors wins over Paper, and Paper wins over Rock.)

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

[ ["Kristen", "P"], ["Jacob", "S"] ] # Jacob would win since S > P

***A: 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(['Kristen,'P'], ['Jacob','S']) # => ['Jacob','S']

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

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

***B: Tournament***:

A rock-paper-scissors tournament is encoded as an array of games - that is, each element can be considered its own tournament.

[  
[  
[ ["Kristen", "P"], ["Ben", "S"] ],  
[ ["Justin", "R"], ["Walter", "S"] ],  
],  
[  
[ ["Allen", "S"], ["Omer", "P"] ],  
[ ["David", "R"], ["Richard", "P"] ]  
]  
]

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

Under this scenario, Ben would beat Kristen (S>P) and Justin would beat Walter (R>S), so Ben and Justin would play (Justin wins since R>S); similarly, Allen would beat Omer, Richard would beat David, and Allen and Richard would play (Allen wins since S>P); and finally Justin 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 ['Justin', 'R']). You can assume that the array is well formed (that is, there are 2n 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 A.

Select one of the rspec tests, copy it, and modify it so that it tests a feature of the rock-paper-scissors library you have implemented. Alternatively, you can write your own test.

Finally, go to your text file and summarize the purpose of each rspec test (including your added test).

**Extra Credit: Metaprogramming**

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

In lecture (and book chapter 3.5), we see 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 define attr\_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 3.5 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

Select one of the rspec tests, copy it, and modify it so that it tests a feature of the attr\_accessor history library you have implemented. Alternatively, you can write your own test.

Finally, go to your text file and summarize the purpose of each rspec test (including your added test).

**Submission**

You should have run tests on your assignment already, but we will additionally be checking your code for quality and “beauty”- not just working. We will also run several additional tests for correctness- remember to check the weird corner cases!

To submit your code and test writeup:

1. **Push your code to github and include the TravisCI file.** (This is good practice for your later assignments when you’re working with your team too!) Follow these instructions carefully!

cd /root/environment/homework/hw-ruby-intro

rm -rf .git

cd /root/environment/homework/hw-ruby-more

rm -rf .git

cd /root/environment/homework/

curl -L https://git.io/JJAly > .travis.yml

Note that these two .git folders must be removed because they contain .git associations that are no longer needed (linking you to <https://github.com/UCCSCS3300/> repos). Your private git repo is linked as a remote url in the homework folder's .git folder, which you can see by running git remote -v while in the homework folder. If you were to leave the .git folders in hw-ruby-intro or hw-ruby-more, the folders would be turned into submodules, which cannot be opened or viewed.

Make sure you are in /root/environment/homework/ - you can verify by running pwd - then run:

git add -A //Adds all files and folders in the current directory (projects) \

git status //Sanity check- make sure ALL of the files you modified are listed AND staged for commit, not just folders.

git commit -m "Committing HW1" //Adds documentation for your code commit \*\*\ git push -u origin master//Pushes your code to the origin (defined above to your class account’s link) to the master branch (branching will be discussed in class later)

* Make sure you use the correct address! DO NOT PUSH TO YOUR PERSONAL GITHUB ACCOUNT!!!!
* When it asks for a username and password, that is your GITHUB username and password, not UCCS. Read error messages, and double check that **all** of your files show up and are openable on the github website!
* Make sure your files are actually being added! If files are missing, do a git add on the missing folders or files and then commit and push again. git status will also show you changes that are staged for commit.

On the github website ([https://github.com/UCCS-CS300/](https://github.com/UCCS-CS3300-Spring2017/instructor_test_account.git)YOURUCCSUSERNAME), you should see two folders: hw-ruby-intro and hw-ruby-more, where each can be opened and contain your modified code.

*Note: Github will be covered much more later in class- for now, this is all you need. Understand what each command means and read all output!*

* If you mistype the address, follow the information here to correct the address: <https://help.github.com/articles/changing-a-remote-s-url/>

\*\* Note that you can also just type git commit. This will open up an editor if you set one up in the setup instructions. See Setup to install and set an editor. ([changeable](http://stackoverflow.com/questions/2596805/how-do-i-make-git-use-the-editor-of-my-choice-for-commits)- instructions here). Then it will show you everything that git status does and has you input your text that documents your change. -m is a shortcut to bypass the text editor, BUT it also makes you more blind to what will be committed.

To make sure that your travis.yml committed properly, check its output! Go to: <https://travis-ci.com/github/UCCS-CS4300-5300/YOURUCCSUSERNAME> The execution of your test runs will be shown and will be used in part for grading.

1. Submit your code on Canvas
2. cd /root/environment/homework/hw-ruby-intro
3. tar -czf yourUCCSname\_hw1\_part1.tar.gz \*
4. cd ../hw-ruby-more
5. tar -czf yourUCCSname\_hw1\_part2.tar.gz \*

You can now right click and download the files from Cloud9.

In Canvas, submit the two tar.gz files **AND** the text/doc file wherein you describe the purpose of each added rspec test from Part 2. **You should submit 3 files in total with your assignment on Canvas.**

**Good Reading:**

Check out the book appendices!

A.5 is a quick guide to version control and A.6 is quick guide to GitHub.

A.4 contains Unix survival skills

Check the lecture slides and Chapter 3 of the book for more Ruby information, examples, and videos.

For Ruby tutorials, check out:

* [tryruby.org](http://tryruby.org/)
* [rubymonk.com](http://rubymonk.com/)
* [code school ruby-bits](https://www.codeschool.com/courses/ruby-bits) (starts free - later paid)
* [codewars Ruby kata](http://www.codewars.com/kata/search/ruby?q=&r%5B%5D=-8&r%5B%5D=-7&r%5B%5D=-6&beta=false)
* [rubykoans.com](http://rubykoans.com/)
* [Ruby Style Guide](https://github.com/styleguide/ruby)
* [Ruby](http://www.zenspider.com/Languages/Ruby/QuickRef.html) cheat sheet