**Points to Ponder**

*Look through these now and then use them to test yourself after doing the assignment*

**Note:** This is again a long list of things to understand! Read through the list, breathe deep, do the assignment, and then come back to them.

* Conditionals and Flow Control
  + What is a "boolean"?
    - A value containing true or false
  + What are "truthy" values?
    - **"Truthiness" and "Falsiness"** are ways of saying "what evaluates to true?"" and "what evaluates to false"?
  + Are nil, 0, "0", "", 1, [], {} and -1 considered true or false?
    - In Ruby, it's simple: nil and false are false and that's it. Everything else is "truthy".
  + When do you use elsif?
    - To test for another if case in your control flow
  + When do you use unless?
    - unless is the opposite of if (which should make sense from the english of it). So it will jump into the included code... UNLESS the statement is true
  + What does <=> do?
    - The **Spaceship Operator** <=> is a special one that comes up because it actually gives three different possible outputs depending on whether the left side is greater than, less than, or equal to the right side.
    - > 1 <=> 1000
    - => -1
    - > 1 <=> 1
    - => 0
    - > 1 <=> -1000
    - => 1
  + Why might you define your own <=> method?
    - The Spaceship can be useful because, like basically everything else, it's actually a method and you can override it in your own classes. It's most commonly used in sorting methods. Imagine that you created a Personclass and you wanted to sort an array of Person objects. You first have to teach Ruby how to compare two Persons by defining the #<=>method for the Person class:
    - def Person
    - def <=> (other\_person) # to compare two people, use last names
    - self.last\_name <=> other\_person.last\_name
    - end
    - end # now we can run people\_array.sort, woohoo!
  + What do || and && and ! do?
    - && aka and, meaning both sides must be true for the full expression to evaluate to true
    - || (the pipe symbol, usually on the same key as the backslash) akaor, meaning that if EITHER of the two sides is true, the expression is true (else false)
    - ! aka not, which reverses the expression from true to false or false to true
  + What is returned by puts("woah") || true?

**Ruby is Lazy** which means two things here:

* It will only evaluate far enough to determine that the expression is definitively true or false.
* It will return whatever is returned by the last part of the expression to get evaluated (instead of just a simple true or false it relies on that returned thing being truthy or falsey).
* It will return woah
  + What is ||=?
    - ||= is a sneaky expression that takes advantage of Ruby's natural laziness -- it basically expands to thing\_a || thing\_a = thing\_b. So ifthing\_a hasn't been set to anything, it becomes thing\_b, otherwise it keeps its original value.
  + What is the ternary operator?
    - You may have seen some oddly compact and strange looking statements that appeared to be if statements under the hood. That's probably because they use the **Ternary Operator**, which is a shorthand notation for a simple if that separates the different parts using the ?and : like:
    - condition ? do\_this\_if\_true : do\_this\_if\_false
    - So:
    - > true ? puts "I like truth" : puts "not gonna happen"
    - "I like truth"
    - => nil
  + When should you use a case statement?
    - For those situations where you're really just checking to see if something equals any one of a number of clear but different options, a **case** statement can be a good substitute. It basically lets you construct a chain of logic that says "if x equalsoption\_a, do this, if it equals option\_b, do this, if it equals option\_c, do this... and otherwise do this."
    - case current\_user.energy # Assume it's an value 1-3
    - when 3
    - puts "Go run a marathon!"
    - when 2
    - puts "Go for a walk."
    - when 1
    - puts "Go take a nap"
    - else
    - puts "You're only supposed to have energy of 1,2 or 3..."
* Iteration
  + What does loop do?
    - A loop is really just code that will run a number of times until some condition is met. A variable is typically used to keep track of which iteration you are on or to otherwise increment until the condition is reached. This is called the **index variable**.
    - loop is the most basic way to loop in Ruby and it's not used all that much because the other ways to loop are much sexier. loop takes a block of code, denoted by either { ... } or do ... end (if it's over multiple lines). It will keep looping until you tell it to stop using abreak statement:
    - > loop { puts "this will not stop until you press CTRL+c" }
    - this will not stop until you press CTRL+c
    - this will not stop until you press CTRL+c
    - ... and so on
    - > i=0 # Our index variable
    - > loop do
    - > i+=1
    - > print "#{i} "
    - > break if i==10
    - > end
    - 1 2 3 4 5 6 7 8 9 10 => nil
  + What are the two ways to denote a block of code?
    - A block of code, denoted by either { ... } or do ... end (if it's over multiple lines).
  + What is an index variable?
    - A variable is typically used to keep track of which iteration you are on or to otherwise increment until the condition is reached. This is called the **index variable**.
  + How do you print out each item of a simple array [1,3,5,7] with:
    - loop?
    - while?
    - for?
    - #each?
    - #times?
  + What's the difference between while and until?
    - until is almost identical to while but, instead of running as long as the specified condition is true, it runs as long as the condition is false
  + How do you stop a loop?
    - break will **stop the current loop**. Often used with an if to specify under what conditions to do that.
  + How do you skip to the next iteration of a loop?
    - next will **jump to the next iteration**. Also usually used with an ifstatement.
  + How would you start the loop over again?
    - redo will let you restart the loop (without evaluating the condition on the first go-through), again usually with some condition
  + What are the (basic) differences between situations when you would use while vs #times vs #each?
    - Your best friends early on will be while for anything that needs to run until a certain condition is reached (like winning the game), #each for any time you want to do stuff with every item in an array or hash, and#times for the simple cases when you just want to do something a fixed number of times.
  + What does nesting loops mean and when would you use it?
    - **Nesting** loops occurs when one goes inside another, so you execute the entire inner loop for each iteration of the outer loop. You'll see those for "two-dimensional" problems, like searching through arrays within arrays, but if you find yourself nesting too often or too deeply, you probably need to reexamine how you've structured your solution overall.
* Blocks, Procs, and Lambdas:
  + How is a block like a function?
    - Blocks are just chunks of code that you can pick up and drop into another method as an input. They're often called "anonymous functions" because they have no name but behave much like functions.
  + How is a block different from a function?
    - They're like little helper functions... you don't find blocks just hanging around without some method (like #each) using them.
  + What are the two ways to declare a block?
    - You **declare a block** using squiggly braces {} if it's on one line ordo ... end if it's on multiple lines (by convention... you can use either one if you really want):
    - > [1,2,3].each { |num| print "#{num}! " }
    - 1! 2! 3! =>[1,2,3]
    - > [1,2,3].each do |num|
    - > print "#{num}!"
    - > end
    - 1! 2! 3! =>[1,2,3] # Identical to the first case.
  + How do you return data from a block?
    - Just like methods, some blocks take inputs, others do not. Some return important information, others do not. Blocks let you use the implicit **return** (whatever's on the last line) but NOT return, since that will return you from whatever method actually called the block.
  + What happens if you include a return statement in a block?
    - That will return you from whatever method actually called the block.
  + Why would you use a block instead of just creating a method?
    - Blocks are used as arguments to other methods (like #each), just like the normal arguments that you see between the parentheses... they just happen to always be listed last and on their own because they tend to take up multiple lines.
  + What does yield do?
    - **The yieldstatement**, which basically says "run the block right here". When you write your own methods, you don't even need to specially declare that you'd like to accept a block. It will just be there waiting for you when you call yield inside your method.
  + How do you pass arguments to a block from within a method?
    - yield can **pass parameters to your block** as well. See this made-up version of the #each method to get an idea of what's happening under the hood. We'll put this method into the Array class so you can call it directly on an array (like [1,2,3].my\_each) instead of having to take the array as an argument like my\_each([1,2,3]):
    - class Array
    - def my\_each
    - i = 0
    - while i < self.size
    - yield(self[i])
    - i+=1
    - end
    - self
    - end
    - end
  + How do you check whether a block was actually passed in?
    - If you want to **ask whether a block was passed** at all (to only yield in that case), use #block\_given?, or rather: yield if block\_given?
  + What is a proc?
    - What if you want to pass TWO blocks to your function? What if you want to save your block to a variable so you can use it again later? That's a job for **Procs**, aka Procedures! Actually, a block is a Proc (which is the class name for a block) and they rhyme just to confuse you.
  + What's the difference between a proc and a block?
    - The block is sort of like a stripped-down and temporary version of a Proc that Ruby included just to make it really easy to use things like those#each iterators.
    - A Proc is just a block that you save to a variable, thereby giving it a bit more permanence:
    - > my\_proc = Proc.new { |arg1| print "#{arg1}! " }
  + When would you use a proc instead of a block?
    - Most of the time, using a block is more than sufficient, especially in your early projects. Once you start seeing the need for using a Proc (like passing multiple arguments or saving it for later as a callback), you'll have Procs there waiting for you.
  + What is a closure?
    - Blocks and Procs are both a type of "closure". A closure is basically a formal, computer-science-y way of saying "a chunk of code that you can pass around but which hangs onto the variables that you gave it when you first called it". It's the blanket term used to refer to blocks and Procs and...
  + What is a lambda?
    - A lambda gives you more flexibility with what it returns (like if you want to return multiple values at once) because you can safely use the explicit return statement inside of one. With lambdas, returnwill only return from the lambda itself and not the enclosing method, which is what happens if you use return inside a block or Proc.
  + What's different between a lambda and a proc?
    - Lambdas are also much stricter than Procs about you passing them the correct number of arguments (you'll get an error if you pass the wrong number).
  + What is a Method (capital "M")?
    - "Method"'s (capitalized because they're actually a class of their own) are really just a convenient way to pass a normal method to another normal method by wrapping the symbol of its name in the word method() So what? To use the same example as we have been so far:
    - Edit: Note that #my\_each has been modified for this example to now take an argument, which the standard #each does not. We're using#my\_eachbelow.
    - class Array
    - def my\_each(some\_method)
    - i = 0
    - while i < self.size
    - some\_method.call(self[i])
    - i+=1
    - end
    - end
    - self
    - end
    - def print\_stuff(word)
    - print "#{word}! "
    - end
    - > [1,2,3].my\_each(method(:print\_stuff)) # symbolize the name!
    - 1! 2! 3! => nil
  + What do Methods basically allow you to do that could probably be pretty interesting when you're writing some more advanced programs later on?
* **Blocks** are unnamed little code chunks you can drop into other methods. Used all the time.
* **Procs** are identical to blocks but you can store them in variables, which lets you pass them into functions as explicit arguments and save them for later. Used explicitly sometimes.
* **Lambdas** are really full methods that just haven't been named. Used rarely.
* **Methods** are a way of taking actual named methods and passing them around as arguments to or returns from other methods in your code. Used rarely.
* **Closure** is just the umbrella term for all four of those things, which all somehow involve passing around chunks of code.
* Enumerable and Modules
  + What is a module?
    - "Enumerable" is actually a "module", which means it is just a bunch of methods packaged together that can (and do) get "mixed in", or included, with other classes (like Array and Hash
  + Why are modules useful?
    - That means that Ruby programmers don't have to write all those methods many different times - they just write them once, package them up asEnumerable, and tell Array and Hash to include them
  + What does #each do?
    - #each is an iterator method you've seen plenty of times before now that comes pre-packaged with the Array and Hash and Range classes and it basically just goes through each item in the object you called it on and passes it to the block that you specified.
  + What does #each return?
    - It will return the original collection that it was called on:
    - > [1,2,3].each { |num| print "#{num}! " }
    - 1! 2! 3! => [1,2,3]
  + What does #map do?
    - You've heard of #map? It's the EXACT same method as collect, just called something different. Some people visualize it in their heads as doing something and collecting the results, other people see it as re-mapping your original object through some sort of transformation. It's more conventional to use #map but both work the same way.
  + What does #map return?
    - Here's a theoretical example more like what you might see when you've got your own website built using Rails, where we may want to send only an array filled with our users' emails out to the webpage:
    - u = User.all
    - @user\_emails = u.map { |user| user.email }
  + What is the difference between #map and #collect?
  + What does #select do?
    - That's too much code and too much hassle. When all you're doing is pulling out, or selecting, certain items based on some criteria, you'd be better served using Enumerable's #select instead.
  + What does #select return?
    - It will run the block on every item of your object (whether array or hash or whatever) and return a new object that contains only those items for which the original block returned true:
    - > my\_array.select{|item| item%2==0 }
    - => [2,4,6,8,100] # wow, that was easy.
  + What is the difference between #each #map and #select?
    - #each returns the original object it was called on because it's really used for its side effects and not what it returns
    - #each\_with\_index passes not just the current item but whatever position in the array it was located in.
    - #select returns a new object (e.g. array) filled with only those original items where the block you gave it returned true
    - #map returns a new array filled with whatever gets returned by the block each time it runs.
  + What does #inject do?
    - For that we need to use #inject (aka #reduce), which passes not just the element but whatever was returned by the previous iteration into the block. You can either specify the initial value or it will just default to the first item of the array. It ultimately returns whatever the result of the last iteration is. Here's a way to sum an array:
    - > my\_array.inject(0){|running\_total, item| running\_total + item }
    - => 120
  + When might you use #inject?
    - What if we want to do something that keeps track of the result as we iterate? Like, say, summing up the elements of an array?
  + How do you check if every item in a hash fulfills a certain criteria?
    - #any? returns true/false (see the question mark?) and answers the question, "do ANY of the elements in this object pass the test in my block?". If your block returns true on any time it runs, any? will return true.
    - #all? returns true/false and answers the question, "do ALL the elements of this object pass the test in my block?". Every time the block runs it must return true for this method to return true.
  + What about if none of the elements fulfill that criteria?
    - #none? returns true only if NONE of the elements in the object return true when the block is run.
  + What (basically) is an enumerator?
    - enumerable is basically a go-between for the original collection and Enumerator
    - A class which allows both internal and external iteration
* Writing Methods
  + How many things should a method ideally do?
    - What should you put into methods? Pretty much everything should be in a method, but **each method should only do ONE thing**. If it's doing two, it's time for another method.
  + What types of things should a method modify?
    - A method should be self-contained and only use those variables that have been passed in. Don't modify global variables or otherwise have side effects for your methods. By the same token, don't destructively modify the arguments or the object your method was called on (unless it's explicitly a bang! method).
  + What should you name a method?
    - When **naming methods** the goal is to be descriptive but short. Name based on what it will return or what the major intended side effect will be. You shouldn't be missing any parts from the name because the method should only do one thing anyway. If you can't tell what the method will return based on the name, you probably need a better name.
  + What does self mean?
    - What is self? It's a word that you see a whole lot in Ruby and it's actually pretty simple... it refers to whatever object the current method was called on (the "caller"). So if I called current\_user.jump,current\_user is the caller of that method. Inside the definition of the#jump method, self would refer to the current\_user.
  + What do you need to do to create your own Ruby script file?
    - You're probably tired of running your methods in IRB (if that's what you're still doing) so it's time to learn how to break them out into a**separate script file** which you can then run in its entirety from the command line.
  + How would you run a Ruby script from the command line?
    - It's easy -- just create a file with the extension .rb, navigate to that file's directory from the command line, and run it using $ ruby filename.rb (the dollar sign is just the command prompt). You'll be able to gets from and puts to the command line now!
  + How can you check whether your script was being run from the command line?
    - A trick you can sometimes use to check whether your script is being run from the command line or as part of a larger program, for instance if you want to puts some stuff only if it's called from the CL (to debug it maybe), is to check whether $0 is the same as \_\_FILE\_\_:
    - if $0 == \_\_FILE\_\_
    - puts "I'm a command line script now!"
    - else
    - puts "Someone else is running me (I feel so violated...)"
    - end
  + What is a shebang line?
    - If you get tired of typing ruby in front of the file, you can tell your computer that the file is an executable one by using the chmodcommand. Just type $ chmod +x your\_filename.rb once and you'll be able to execute it, but you now need to let the computer know that it should use the Ruby interpreter when running the file. That requires you to include, at the very top the file (before any spaces or lines), a**Shebang Line** that points to your Ruby executable:
    - #!/usr/bin/ruby
    - ...or wherever it's located.
  + What does require do?
    - If you want to include a gem file (a library of methods, for instance) in your IRB session, you'll need to use require to bring it in. The default directory for require accesses any gems you may have downloaded from the internet.
    - You can do almost the same thing to bring your new script file into IRB (if you want to access its methods, for instance), though you have to explicitly provide the path to it. If it's in your current directory, that's easy, it's just ./filename.rb:
    - > require './your\_script.rb'
    - => true
  + What does load do?
    - If you were to change and save your script file, to get those changes in IRB you can either reload IRB entirely or use **load** instead of require.require will only load the file once, a feature designed to save you headache normally. load will load it for you regardless.
    - > load `./your\_script.rb`
    - => true
  + What is the difference between require and load?
    - require searches for the library in all the defined search paths and also appends .rb or .so to the file name you enter. It also makes sure that a library is only included once. So if your application requires library A and B and library B requries library A too A would be loaded only once.
    - With load you need to add the full name of the library and it gets loaded every time you call load - even if it already is in memory.
  + How do you access any parameters that were passed to your script file from the command line?

Another thing that you'll probably want to do at some point is to **access the variables that were passed to your script** from the command line. For instance, if you ran:

$ ruby ./string\_printer\_script "howdy" "everyone"

You've passed in "howdy" and "everyone". You access it by digging into a special constant (a variable you shouldn't try to change) called ARGV, which is an array that contains all those arguments:

# In our string\_printer\_script.rb file:

inputted\_strings = ARGV

puts "You inputted: "

inputted\_strings.each do |str|

puts "str"

end

$ ruby ./string\_printer\_script.rb "howdy" "everyone"

You inputted:

howdy

everyone

* + What does #send do?
    - Stepping away from scripts for a minute, one nifty command that you probably haven't had a chance to run into yet is **#send**, which will let you run a method. Simple. Just call it on whatever object you'd normally run the method on. Adapted from [the docs](http://ruby-doc.org/core-2.0/Object.html#method-i-send):
    - class Klass
    - def hello(string)
    - "Hello #{string}"
    - end
    - end
    - k = Klass.new
    - k.send :hello, "gentle readers" #=> "Hello gentle readers"
  + When would #send be used that's different from just running the method on an object 'normally'?
    - So simple, why should you care? Ruby lets you define new methods in runtime and this lets you call them dynamically (since the method name is no longer hardcoded). Keep that in your back pocket... you won't need it yet but it's great to know.