**Points to Ponder**

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

**Note:** We recommend you read through the list, do the assignment, and then test yourself by seeing if you can answer all these questions.

* **Numbers, Operators and Expressions:**
  + What's the difference between an Integer and a Float?
    - When doing mathematical operations, Ruby expects the result to be the same type as the inputs, so dividing two integers by each other will produce an integer... whether you want to or not:
    - > 5 / 3
    - => 1
    - To fix this, you need to make one of the inputs a different data type that can handle decimals, like a floating point number (float):
    - > 5.0 / 3 # as long as one of them is a float...
    - => 1.6666666666666667 # ... the result is a float
  + Why should you be careful when converting back and forth between integers and floats?
    - Converting between integers and floats is easy -- just use to\_i andto\_f respectively:
    - > 5.0234.to\_i
    - => 5
    - > 5.to\_f
    - => 5.0
  + What's the difference between =, ==, and ===?
    - = is simple assignment operator, Assigns values from right side operands to left side operand; (a == b) is not true.
    - == checks if the value of two operands are equal or not, if yes then condition becomes true; c = a + b will assign value of a + b into c
    - === typically asks whether the thing on the right is a member or a part or a type of the thing on the left. Is used to test equality within a when clause of a *case* statement; (1...10) === 5 returns true.
  + How do you do exponents in Ruby?
    - Ex. 2\*\*3 is 8
  + What is a range?
    - A **Range** is just a continuous sequence and we represent it in a shorthand way. If we want to say 3, 4, 5, 6, 7, 8, 9, 10, 11
  + How do you create a range?
    - it's much easier to just write it the short way (3..11), meaning "all the integers beween 3 and 11, including both 3 and 11"
  + What's the difference between (1..3) and (1...3)?
    - (1..3) is 1 2 3 INCLUSIVE
    - (1…3) is 1 2 EXCLUSIVE
  + What are three ways to create a range?
    - (3..11), meaning "all the integers beween 3 and 11, including both 3 and 11". If we wrote it(3...11), it would actually exclude 11 You can also create a range using Range.new(start, finish)
* **Strings:**
  + What's the difference between single and double quotes?
    - **Double Quotes** are often interchangeable with **Single Quotes**... there's almost no difference and you're free to use either.
  + What is string interpolation?
    - If you define a variable monkey that's equal to the string "Curious George", and then you have a string that says "I took #{monkey} to the zoo", Ruby will do something called **string interpolation** and replace the #{monkey} bit with the value of monkey—that is, it will print" I took Curious George to the zoo".
  + What are escape characters?
    - **Escaping** characters just means telling the output program to not treat them specially at all (like the pound symbol, which has special meaning before the curly braces). You do so with a back slash \ before whatever you want to escape.
  + What are line breaks?
    - A line break ends the line you are currently on and resumes on the next line
  + How do you make other things into strings?
    - #to\_s is a method that will try to convert anything into a string. Note that we put a # in front of method names by convention (you don't use that symbol when you're actually calling the method).
    - > 12345.to\_s
    - => "12345"
  + How do you concatenate strings?
    - **Combining Strings** without using interpolation can be done using "concatenation", or basically just adding them together:
    - > my\_name = "Billy Bob"
    - > "hello" + " world" + ", say I, the great " + my\_name + "!"
    - # => "hello world, say I, the great Billy Bob!"
    - Instead of adding them with a plus +, you can also use the friendly shovel operator << to append to a string (just like with arrays...):
    - > "howdy " << "fella!"
    - => "howdy fella!"
  + How do you access a specific character or substring?
  + How do you split up strings into arrays?
    - Ruby has a built-in method for this called.split; it takes in a string and returns an array. If we pass it a bit of text in parentheses,.split will divide the string wherever it sees that bit of text, called a **delimiter**. For example, text.split(“,”) tells Ruby to split up the string text whenever it sees a comma.
  + How are strings and arrays similar?
    - You can use array functions on strings
  + How do you get and clean up user input on the command line?
    - When you write your Ruby programs, you'll probably want to ask for**User Input**... which is easy with #gets, which then waits for the user to type something. You'll want to store whatever the user types into a variable and be sure to trim off the extra line break (from when the user hit the enter key) using #chomp:
    - > player1 = gets
    - Erik # this was typed in manually
    - => "Erik\n" # woah, let's get rid of that \n
    - > player1 = gets.chomp
    - Erik
    - => "Erik" # better.
    - #chomp will cut off a space or newline at the END of the string (and can take an optional input so you can specify what exactly to chomp off).#strip will remove ALL spaces and newlines from both the beginning and end of the string:
    - > " dude \n".chomp
    - => " dude " # still have the extra spaces
    - > " dude \n".strip
    - => "dude" # clean as a whistle.
  + What does it mean that strings are "mutable" and why care?
  + What is a symbol?
    - You can think of a Ruby symbol as a sort of name. It's important to remember that symbols **aren't** strings:
    - Symbols pop up in a lot of places in Ruby, but they're primarily used either as hash keys or for referencing method names.
  + How is a symbol different from a string?
    - While there can be multiple different strings that all have the same value, there's only one copy of any particular symbol at a given time.
  + What is a Regular Expression (RegEx)?
    - Regular Expressions are really just a special syntax that is used to find things (and not just in Ruby, they're used all over the place).
  + How can you center or right-justify a string?
    - #ljust will let you pad your string with extra spaces (or any other character you choose) up until a specified total amount of characters, so "hi".ljust(4) => "hi " and"hi".ljust(6,"\*") => "hi\*\*\*\*". It can be handy for prettying up some of your command line outputs.
    - #rjust does the same thing but pushes your string to the right, e.g."hi".rjust(6) => " hi"
    - #center is the same idea but it centers your string:"hi".center(6,"!") => "!!hi!!"
* **Arrays:**
  + What are three ways to create an array?
    - **Creating an Array** can happen in many different ways. You can either create it empty, specify how many spaces it should have (still empty), or even **fill it with default values**:
    - > a = Array.new # 1
    - => [] # see, it's empty
    - > b = []
    - => [] # still empty
    - > c = Array[]
    - => []
    - > d = %w{ I am not a crook } # converts the string (no quotes) to an array
    - => [ "I", "am", "not", "a", "crook" ]
  + How do you prepopulate the array with default data?
    - > empty\_a = Array.new(5)
    - => [nil, nil, nil, nil, nil]
    - > full\_a = Array.new(3, "hi")
    - => ["hi", "hi", "hi"]
  + How do you access items in an array?
    - **Accessing Items** is super easy, just start from 0 like you did with strings. Just like with strings, you can start from the end of the array using negative numbers from -1 and you can even grab ranges of values at a time:
    - > arr = [1, 3, 5, 7, 2] # favorite way to declare an array
    - => [1, 3 ,5 ,7 ,2]
    - > arr[0]
    - => 1
    - > arr[-1]
    - => 2
  + How can you access a specific group of items in an array?
    - > arr[1..3]
    - => [3, 5, 7] # this returned an array!
    - > arr.slice(1..3)
    - => [3, 5, 7] # same as using [1..3]
    - > arr[1..200000]
    - => [3, 5, 7, 2] # no error... silently cuts off at the end
  + How do you modify the items in an array?
    - **Modifying Items** is as simple as accessing them is... just set them equal to a value:
    - > arr[0] = 42
    - => 42
    - > arr
    - => [42, 3, 5, 2] # changed it!
    - > arr[0..2] = 99
    - => 99
    - > arr
    - => [99, 2] # wiped out several values, oops...
  + How do you combine arrays?
    - **Adding Arrays** is also done similarly to strings, by just mashing one onto the end of the other:
    - > first = [1,2,300]
    - => [1,2,300]
    - > second = [7,8,9]
    - => [7,8,9]
    - > combined = first + second
    - => [1,2,300,7,8,9] # this is a NEW array
  + How do you find the values in one array that aren't in another?
    - **Subtracting Arrays** is a bit different... think of the minus sign as saying "take away any and all values that are duplicated in the right array from the left array". The only values remaining will be those from the left that were not included in the right side at all:
    - > [1,2,3] - [2,3,4]
    - => [1] # the 4 did nothing
    - > [2,2,2,2,2,3,4] - [2, 5, 7]
    - => [3,4] # it killed ALL the 2's
  + How do you find values in both arrays?
    - If you want to find values in **Both** arrays, check their union using the ampersand &:
    - > [1,2,3]&[2,4,5]
    - => [2]
  + What is the difference between push/pop and shift/unshift?
    - What if you only want to add or subtract one single value? That's a very common operation with arrays, and Ruby has provided four handy methods that let you either pluck away or add onto the front or back of the array. First, the more common is to add or remove stuff from the END of the array, using #push or #pop:
    - > my\_arr = [1,2,3]
    - => [1,2,3]
    - > my\_arr.push(747)
    - => [1, 2, 3, 747]
    - > my\_arr
    - => [1, 2, 3, 747] # warning: we modified my\_arr!!!
    - > my\_arr.pop
    - => 747
    - > my\_arr.pop
    - => 3
    - > my\_arr
    - => [1, 2] # warning: pop also modified my\_arr
    - What if you want to take the item off the FRONT of the array? This is less common. For that, use the similar #shift and #unshift methods:
    - > my\_arr = [1,2,3]
    - => [1,2,3]
    - > my\_arr.shift
    - => 1
    - > my\_arr
    - => [2,3] # warning: shift also modified my\_arr!
    - > my\_arr.unshift(999)
    - => [999, 2, 3]
    - > my\_arr
    - => [999, 2, 3] # warning: unshift... yep, modified my\_arr.
  + What is the shovel operator?
    - So the #push/#pop/#shift/#unshift methods should take you wherever you realistically need to go. Although there's another handy method you should be aware of, the **Shovel Operator**, aka <<. This method is almost identical to push, since it just jams whatever's to its right into the array:
    - > my\_arr = [1,2,3]
    - => [1,2,3]
    - > my\_arr << 3
    - => [1, 2, 3, 3]
    - > my\_arr << [4,5]
    - => [1, 2, 3, 4, [4, 5]] # Array within array alert!
  + How is > arr.pop different from > arr[-1]?
    - arr.pop removes the last element in the array whereas arr[-1] accesses the last element in the array
  + How is pushing or <<ing another array into your array different from just adding them together?
    - Adding is between two arrays whereas push or << puts whatever is to the right into the array
  + How do you delete items in an array?
    - **Deleting Items** from an array should be done carefully because, if you're deleting items inside a loop or something like that, it will change the index of the other items and you'll need to anticipate this or live to regret it. Delete an item at a specific index using #delete\_at, which is sort of like poping but from anywhere you want:
    - > my\_arr = [1,2,3]
    - => [1, 2, 3]
    - > my\_arr.delete\_at(1)
    - => 2
    - > my\_arr
    - => [1,3]
  + Why should you be careful deleting items in an array?
    - If you're deleting items inside a loop or something like that, it will change the index of the other items and you'll need to anticipate this or live to regret it.
  + How can you convert arrays to strings?
    - Strings are a lot like arrays... so much so that we can even **Convert an Array into a String**! Just use #join and tell it what, if anything, you want in between each element (the "separator"):
    - > ["he", "llo"].join
    - => "hello"
    - > colorful\_bugs = ["caterpillar", "butterfly", "ladybug"]
    - => ["caterpillar", "butterfly", "ladybug"]
    - > "I found a #{colorful\_bugs.join(' and a ')} in the yard!"
    - => "I found a caterpillar and a butterfly and a ladybug in the yard!"
  + How can you convert from other data types to arrays?
    - Want to know a cool way to make an array? Create it from a Range and just **Convert it to an Array**:
    - > my\_awesome\_array = (1..6).to\_a
    - => [1,2,3,4,5,6]
  + How can you figure out if an array contains a particular value?
    - See if an array **includes an item** AT ALL by using #include?, which, as you should see from the ? at the end, returns true or false:
    - > my\_arr.include?(3)
    - => true
    - > my\_arr.include?(132)
    - => false
  + How do you find the biggest item in an array?
    - #max to find the **biggest value** of an array
  + How do you find the smallest item in an array?
    - #min to find the **smallest value** of an array
  + How do you remove any duplicates from your array?
    - #uniq to **remove all duplicates** from your array
  + How do you find out how big an array is?
    - #size to find out **how big the array is**
  + How do you put an array in order?
    - #sort will clean it up again for you by putting your array **in order**.
  + What are the naming conventions for arrays?
    - Do as I say and not as I do: name your arrays with the plural form (because it has a bunch of things in it, like colorful\_bugs instead ofcolorful\_bug) and be descriptive.
  + What should you store in arrays?
    - Arrays are almost as ubiquitous as strings. You'll be working with them all the time to help store data, everything from the names of all your users to coordinates on a game board. An array is an all-purpose bucket into which you can put pretty much anything.
* **Hashes:**
  + What is a hash?
    - Hashes are sort of like JavaScript objects or Python dictionaries. If you haven't studied those languages, all you need to know that a hash is a collection of **key-value pairs**.
  + What are keys and values?
    - Keys can be anything that's hashable, such as integers, strings, or symbols, but they must be unique for the hash they belong. The values to which keys refer can be any Ruby object.
  + How is a hash similar to an Array?
    - Like arrays, they have values associated with indices, but in the case of hashes, the indices are called "keys."
  + How is a hash different from an Array?
    - A Hash is a way of storing data by a specifiable key, as opposed to an array which can only use numbers.
  + What are 3 ways to create a hash?
    - We can create hashes several ways, but two of the most popular are **hash literal notation**:
    - new\_hash = { "one" => 1 }
    - and **hash constructor notation**:
    - new\_hash = Hash.new
  + What is the hash rocket?
    - The hash syntax you've seen so far (with the=> symbol between keys and values) is sometimes nicknamed the **hash rocket** style.
* numbers = {
* :one => 1,
* :two => "two",
* :three => 3,
* }
  + - This is because the => looks like a tiny rocket!
    - However, the hash syntax has changed in Ruby 1.9. Just when you were getting comfortable!
    - The good news is that the new syntax is easier to type than the old hash rocket syntax, and if you're used to JavaScript objects or Python dictionaries, it will look very familiar:

new\_hash = { one: 1,

two: 2,

three: 3

}

* + - The two changes are:
    - You put the colon at the *end* of the symbol, not at the beginning;
    - You don't need the hash rocket anymore.
  + How do you access data in a hash?
    - Can we iterate over just keys or just values?
    - This is Ruby. Of course we can.
    - Ruby includes two hash methods, .each\_keyand .each\_value, that do exactly what you'd expect:
* my\_hash = { one: 1, two: 2, three: 3 }
* my\_hash.each\_key { |k| print k, " " }
* *# ==> one two three*
* my\_hash.each\_value { |v| print v, " " }
* *# ==> 1 2 3*
  + How do you change data in a hash?
    - puts "Enter text:"
    - text= gets.chomp
    - words = text.split(" ")
    - frequencies = Hash.new(0)
    - words.each do |word|
    - frequencies[word] += 1
    - end
  + What types of data are good to store in a hash?
    - They're basically just containers for data, like arrays, but instead of storing data based on numeric indices, you use "keys" which can be strings or symbols. This makes hashes more appropriate for storing data with a bit more depth to it.
  + What are options hashes?
    - You see hashes all the time in Rails, including as a way of passing options or parameters to a method (since they can store all kinds of different things and be variably sized), and these are often called**Options Hashes**. Methods are often defined along the lines ofdef method\_name arg1, arg2, arg3, options\_hash, allowing the user to specify any number of different parameters for that method.
  + How do you delete data from a hash?
    - **Delete** from a hash by just setting the value to nil or by calling the#delete method:
    - > favorite\_smells[:flower] = nil
    - => nil
    - > favorite\_smells
    - => {:cooking => "bacon" } # one deleted...
    - > favorite\_smells.delete(:cooking)
    - => "bacon"
    - > favorite\_smells
    - => {} # ...and the other.
  + How do you add hashes together?
    - What if we want to **add two hashes together**? Just use #merge. If there are any conflicts, the incoming hash (on the right) overrides the hash actually calling the method.
    - > favorite\_beers = { :pilsner => "Peroni" }
    - => { :pilsner => "Peroni" }
    - > favorite\_colors.merge(favorite\_beers)
    - => {"eyes"=>"blue", "hair"=>"blonde", "skin"=>"sunburned", :pilsner => "Peroni"}
  + How do you list out all the keys or values?
* restaurant\_menu = {
* "noodles" => 4,
* "soup" => 3,
* "salad" => 2
* }
* restaurant\_menu.each do |item, price|
* puts "#{item}: #{price}"
* end
  + How do you see if the hash contains a key or value?
    - If you want to know what **All the Keys** are (more common) or **All the Values** are (less common) in a hash, just use the aptly named #keysand #values methods to spit them out as an array:
    - > favorite\_colors.keys
    - => ["eyes", "hair", "skin", :pilsner]
    - > favorite\_colors.values
    - => ["blue", "blonde", "sunburned", "Peroni"]
  + What is a set?
    - A simpler kind of hash is called a **Set**, and it's just a hash where all the values are either True or False. It's useful because your computer can search more quickly through this than an array trying to store the same information due to the way it's set up behind the scenes.
* **Dates and Times:**
  + How do you get the current date and time?
    - To **Get Current Time** you just create a new Time object with no parameters or use Time.now, which is the same thing:
    - > Time.new
    - => 2013-07-10 17:04:10 -0700
    - > Time.now
    - =>2013-07-10 17:04:11 -0700
  + How do you find just the Year? Month? Hour? Second? Weekday?
    - Time gives you some handy methods to ask it questions. Almost all of them are very intuitive, so the general rule is "if you think the method should exist, it probably does":
    - > my\_time = Time.now
    - => 2013-07-10 17:04:10 -0700
    - > my\_time.year
    - => 2013
    - > my\_time.month
    - => 7
    - > my\_time.day
    - => 10
    - > my\_time.wday
    - => 0 # the day of the week, starting Sunday
    - > my\_time.hour
    - => 17
    - > my\_time.min
    - => 4
    - > my\_time.sec
    - => 10
  + How do you create a Time specifically for 12/25/2013?
    - Time also takes inputs if you want to create a specific time, from year to time zone:
    - Time.new(year, month, day, hour, min, sec, time\_zone\_offset\_from\_utc)
    - > Time.new(2012,2,14)
    - => 2012-02-14 00:00:00 -0800
  + How do you find how many days have passed between twoTime's?
    - You can add and subtract times just like they were numbers (because, remember, they basically are... just the number of seconds since 1970):
    - > vday = Time.new(2012,2,14) # Valentine's Day!
    - => 2012-02-14 00:00:00 -0800
    - > vday+3600 # 1 hour's worth of seconds
    - => 2012-02-14 01:00:00 -0800
    - > xmas = Time.new(2013,12,25)
    - => 2013-12-25 00:00:00 -0700 # Xmas!
    - > ( xmas - Time.now )/60/60/24.to\_i
    - => 167 # That's too long..
  + What's the difference between UTC and GMT and Local times?
    - I prefer to think of UTC as "Universal Time Code" because reasons. UTC is the new GMT... Greenwich Mean Time. You'll start thinking of things in terms of "how many hours away am I from England?"
  + How would you find out the time that was 100 seconds ago? 10 days ago?
    - Subtract the number of seconds from the time
* **Other Random Stuff:**
  + What is nil?
    - It's important to realize that false and nil are **not** the same thing: false means "not true," while nil is Ruby's way of saying "nothing at all."
  + How do you check if something is nil?
    - The .nil? method will return true if the object it's called on is nil, and false otherwise:
* nil\_variable = nil
* age = 26
* nil\_variable.nil? *# true*
* age.nil? *# false*
  + What's the difference between nil and blank and empty?
    - **#blank? and #empty?** are similar -- both basically check if the object has nothing in it -- but #blank? will also ignore any whitespace characters. Note that*#blank?*is a method provided by Rails and is not available in Ruby.
  + Are the following nil or empty?
    - " ", "", [], [""], {}
    - neither, empty, empty, neither, ?
  + What's the difference between puts and p and print?
    - p will give you some more information because it runs the #inspect method on the object while #puts runs the #to\_s method. **#inspect** is meant to be informative where #putsis "pretty". The difference may not be readily apparent while you're only working with simple objects like strings and arrays, but you'll notice it when you start creating your own objects and you want to see what's inside (without typing out puts my\_object.inspect).
  + What does inspect do?
  + What do +=, -=, \*= and /= do?
    - a += b is the same as a = a + b
    - a -= b is the same as a = a - b
    - a \*= b is the same as a = a \* b
    - a /= b is the same as a = a / b
    - a %= b is the same as a = a % b
    - a \*\*= b is the same as a = a \*\* b
  + What is parallel assignment?
    - **Parallel Assignment** is when you assign the values of more than one variable at a time (though it works for arrays as well!):
    - > a, b = 1, "hi"
    - => [1, "hi"] # ignore this output
    - > a
    - => 1
    - > b
    - => "hi"
    - > my\_array = [1,2,3,4]
    - => [1,2,3,4]
    - > my\_array[1], my\_array[3] = 100, 200
    - => [100,200] # ignore
    - > my\_array
    - => [1,100,3,200]
  + What's the easiest way to swap two variables?
    - > a = 10
    - > b = 20
    - > a,b = b,a
    - > a
    - => 20
    - > b
    - => 10