Ruby - Final Project

Ruby was created by Yukihiro Matsumoto in 1993 who wanted to make a new language that combined the features from several of his favorite languages such as Perl, Ada, and Lisp. The main goal was to make Ruby a scripting language that was truly object oriented and easy to use. Ruby was publicly released on December 21, 1995 and has rose in popularity ever since, with it now being one of the top programming languages in the world. It has become even more prevalent with the release of the Ruby on Rails framework in 2005. While mainly used now in this capacity, it is a general purpose language capable of solving most problems.

Yukihiro Matsumoto created Ruby with the philosophy that people should program with humans in mind. In an Interview with Bill Venners, Matsumoto said, “Don't underestimate the human factor. Even though we are in front of computers, they are media. We are working for human, with human.” (The philosophy of Ruby, Bill Venners). With this philosophy in mind, Matsumoto decided that the language itself must be simple, easy to understand, and easy to use, so that programmers could think about their problems more naturally, or in a people-first mindset. As a result of this, Ruby is fundamentally an easy language to pick up and learn quickly, so it has built up a large community very quickly. This community rapidly helped develop the language via frameworks such as Rails, Sinatra, and many more. The use of rails in web development skyrocketed the popularity of the language and caused Ruby to be classified as one of the top 5 languages to learn within the last 5 years. Ruby on Rails is now used as the backbone of many of the top websites used today. AirBnB, github, and shopify use Ruby currently and twitter used Ruby on Rails when it first started. The language is not only easy to use and understand, but it is also powerful enough to run all of these high traffic websites seamlessly.

Ruby is an Interpreted language. When you run a Ruby script, everything is parsed and executed. Matsumoto himself wrote what is arguably the most popular interpreter for Ruby, “Matz’s Ruby Interpreter” or MRI for short. However, Ruby has the ability to also be compiled. Some of the most popular compilers for Ruby are called JRuby and Rubinius which both function in largely the same way. They work in a very similar way to Java and the Java Virtual Machine. Rubinius, for example, takes each line of Ruby code and compiles it into Rubinius Bytecode which is then run through the Rubinius Virtual Machine which translates the bytecode into native machine language. Rubinius is comprised of two sections. A ruby compiler, which is written in Ruby itself, to take Ruby code and turn it into bytecode. This is complimented by the Rubinius Virtual Machine which uses a compiler that was written in C++ to take the bytecode and turn it into native machine code. This enables Ruby to be run anywhere. JRuby functions very similarly but does not show the power and flexibility of Ruby as much as Rubinius does. The first part of JRuby is written in Java and turns Ruby into bytecode. Then a JRuby Virtual Machine, written in C, turns this bytecode into native machine code. Nonetheless, these two tools are compilers that were written for what is largely referred to as an interpreted language, and prove that Ruby can function as a compiled language as well as an interpreted language.

The Semantics behind Ruby are pretty simple; everything is an object. Every value, every class, even things that most languages define as primitives, are all objects in Ruby, and as a result can have methods called on them, which leads to some pretty powerful programming practices and a very intuitive way of designing your projects when using ruby. But Ruby is also described as a “multi-paradigm” language because it also supports procedural and functional programming.

The Syntax of Ruby is designed to be simple, yet similar to other programming languages. This has the dual benefits of being familiar and easy to pick up by current industry professionals in the IT field, but also newcomers who may just be starting to tackle object oriented programming.

Print statements in Ruby are very similar to other languages. The print keyword is used to print a string, which follows the keyword. The puts keyword prints a string, but with a newline at the end. For example, the simple Hello World program would be written  
  
print “Hello World”  
puts “Hello World”  
  
 The math operators are the same as other programming languages: addition +, subtraction -, multiplication \*, division /, modulus %. One that is unique to Ruby is the power operator, which takes a number to the power of another number. For example 2\*\*3 would be 2 cubed.  
 Variables are easy to declare in ruby! That is because Ruby will automatically declare a type to a variable, so all you have to do is give a variable a name and a value. For example, if you wanted to make a variable that holds a name:  
  
name = “Ruby”  
  
Now, let’s say we wanted to print out the value of that variable. To do so, we set up the print statement as usual, but we format the variable like this: #{variable}. So, to print out the name variable:  
  
print “I love #{name}”  
puts “I love #{name}”  
  
Comments are a must have in any program, and they are easy in Ruby. Use the # symbol for a single line comment, =begin to start a multiline comment, and =end to end it.  
  
# Single line comment  
=begin  
this is a   
multiline comment  
=end  
  
If statements use the if, elsif for the if else commonly used in other languages, and else. Parentheses are not needed after an if statement, but can be used. The end keyword ends the if block.  
  
If x > y  
 puts “Greater”  
elsif (x < y)  
 puts “Less”

else  
 puts “Same”  
end  
  
To store input into a variable, the keyword gets is used in the variable assignment.  
  
name = gets  
  
The logical operators used for boolean comparisons are and/&&, or/||, not/!. These can be used in if statements, assignments, loops...etc. On the topic of loops, there are 3 main types of loops in Ruby: until, while, and for. Until loops run until the given parameters are met, While loops run while the given parameters are met, and for loops run over a given range of numbers. The next keyword skips the rest of the loop and starts the next iteration of the loop. The end keyword ends the loop  
  
until a > 10 while a < 10 for i 1...5  
 puts “#{a}” puts “#{a}” next if i%2 == 0  
 a += 2 a += 2 puts i  
end end end  
  
Methods are declared much like ML and Python, with no return type needed to be specified. The last line of the function will be what is returned. The def keyword is used to define the beginning of the function, and is then followed by the name of the function. Then, the parameters that are being passed to the function are defined, inside or outside parentheses, separated by commas. The end keyword is used to end the method definition.  
  
def hello(first, last)  
 puts “Hello #{first} #{last}”  
end  
  
Arrays work differently in Ruby than in languages in Java or C. Arrays in Ruby do not have to hold strictly one type of data. The arrays can hold multiple types of data, or even arrays. Here is an example of an array definition:  
  
a = [1, “Will”, 3.14, [3, 4]]  
  
The indexes start at 0, and to access the data at a given indice, put the indice in brackets next to the name of the array: a[2] would return 3.14. The array class also comes with many built in methods, such as reverse, which reverses the order of the array.  
  
a.reverse → [[3,4], 3.14, “Will”, 1]  
  
Files are manipulated very similarly to most other languages:   
  
file = File.open(“filename”, “w+”)  
file = File.new(“test.txt”, “w+”)  
file.puts(“text”)  
file.close  
  
Open opens up a file, new creates a new file, puts works like print, but instead of printing out, it writes the string to the file, and close closes the file.  
  
Classes operate very similarly to what you may be familiar. The class keyword starts the class definition. The attr\_accessor is used to automatically make getters and setters for the following variable name, stated with a colon in front of it. Methods are declared inside the class just like they normally would be. The end keyword is used to end the class.

Ruby is unique in that it supports methods, blocks, procs, and lambdas. These four functions are the cornerstones of Ruby’s functional style of programming. They are extremely powerful, flexible, and elegant. Ruby methods are similar to the methods in most popular programming languages. It is able to accept arguments, return values, and has the basic structure of what you would see in C++ or Java. Ruby blocks are snippets of code that can be created to be executed later. Blocks are passed to methods that yield them within the **do** and **end** keywords. Blocks are only called by its corresponding method, therefore, they are useless when on left their own. The block is very helpful when you are looking to reuse code inside of a method and for format purposes. An example of a method using a block,

**def** **my\_method**  
 puts "1"  
 **yield**  
 puts "3"  
 **end**

my\_method **do**  
 puts "2"  
 **End**

This block simply prints out “2” when the keyword yield is called by the method. Blocks also have the ability to receive parameters, return data, and call methods, similarly to the functionality behind methods. The key difference, and whats makes blocks unique, is their ability to only be accessed by one specific method.

The proc and lambda are very similar to the block function we just discussed. A “proc” is an instance of the Proc class, which holds a code block to be executed, and can be stored in a variable. Similar to the proc, A lambda is a way to define a block & its parameters with some special syntax. I like to think of procs and lambdas as containers for blocks. These containers give us the ability to pass blocks to the parameters of other methods. This is advantageous because it allows for the use of higher order functions and programming in a functional style. They also give us the ability to use multiple blocks inside a method since we are able to pass multiple of these containers in as arguments. Here is an example of initializing both a lambda and a proc,

times\_2 = Proc.new { |num| num \* 2 }

times\_2.call(3) # returns 6

times\_2 = lambda { |num| num \* 2 }

times\_2.call(3) # returns 6

In this case, both the lambda and proc store a block that multiplies the parameter, num, by two. So what's the difference between procs and lambdas? There’s not much. Lambdas are essentially procs with a few small adjustments. Lambdas are more like “regular” methods in two ways: they enforce the number of arguments passed when they’re called and they use “normal” returns. When calling a lambda that expects an argument without one, or if you pass an argument to a lambda that doesn’t expect it, Ruby will throw an error, however, this does not occur in procs. Also, a lambda treats the return keyword the same way a method does. When calling a proc, the program yields control to the code block in the proc. So, if the proc returns, the current scope returns. If a proc is called inside a function and calls return, the function immediately returns as well. Because of these factors, lambdas have taken over as the go-to container for storing blocks and working with higher order functions.

With the support of these four different functions in ruby, the programmer is able to program in a functional style. Ruby also supports currying, which is a cornerstone functional programming technique, along with immutable states. Therefore, is ruby considered a functional programming language? Although ruby has the tools to perform many functional programming techniques, it is not considered a fully functional programming language. It only allows the programmer to code in a functional style if they choose to do so. For example, fully functional programming languages like Haskell curry their functions by default. To curry a method in ruby you have to first get a reference to that method, then turn that method into a proc, and finally tell it to work with partial application. This is not nearly as elegant for fully functional languages. Also, ruby considers nearly everything in its language as an object. Functional languages, at their core, focus their structure around the function, not the object. Ruby fails to do this in nearly every aspect. However, the power of ruby being able to support functional ideas while keeping the familiar object oriented look is truly remarkable.

**Resources**

<https://www.artima.com/intv/ruby4.html>

<http://patshaughnessy.net/2012/2/15/is-ruby-interpreted-or-compiled>

<https://blog.appsignal.com/2018/09/04/ruby-magic-closures-in-ruby-blocks-procs-and-lambdas.html>

<https://medium.com/@baweaver/functional-programming-in-ruby-state-5e55d40b4e67>