Grand Valley State University

The Wonderful World of Ruby

Farid Karadsheh

CIS 343, Section 1

Professor Ira Woodring

12/12/2018

Ruby is a remarkable programming language. There is simply so many ways one can enjoy programming in Ruby. It is no wonder that Ruby’s own official web page refers to Ruby as “a programmers best friend”. But, that is quite the bold statement, and no proud computer scientist would listen to such a statement sitting down. Anybody who wears such a title on their shirt sleeve will demand evidence, and so, evidence shall be provided. So, here we are, about to explore the wonders, features, and history of a language we call Ruby.

In computer science, software engineering, life in general, and pretty much every discipline out there, the question, “why”, always pops up. “Why this”, “Why did that happen”, “Why isn’t this working”, are just a few examples, but it is crucial to understand that it is an absolute necessity to provide a satisfying answer to the question ‘why’. A qualifying answer will cause the asker to fall back into their seat, put their palm to their chin, and say, ‘of course’. Yet, in a world teeming with programming languages that vie for our attention, it seems that few can provide a satisfying answer to the question, ‘why this programming language’. One might just say Ruby shines like a gem within a rock bed.

Yukihiro "Matz" Matsumoto, Ruby’s creator, had a very specific philosophy from the outset. Matz wanted to design a language with the human in mind. Yes, specifically the human, for all programmers are humans; thus, by designing for the human first, one can achieve a language that benefits both the developer and the software (Ruby, Wikipedia).

One of the inspirations to create Ruby was the fact that Matz disapproved of programming languages that felt like OO was an addition rather than a core principal. Ruby is a pure OO (object oriented) language, and it has been from its birth in 1993. In 1999, Matz said, “As a language manic and OO fan for 15 years, I really wanted a genuine object-oriented, easy-to-use scripting language. I looked for, but couldn't find one” (Ruby FAQ). In addition to OO, Ruby also supports paradigms such as functional programming and imperative programming.

Ruby is inspired by languages such as: Lisp, Smalltalk, Eiffel, Ada, and Perl (About Ruby). Although the *About* section never explicitly states that Ruby is influenced by Python, one can infer a connection between the two given the similar syntax and the number of occurrences that the word Python appears within the FAQ. One might describe this connection as Ruby and Python share common ancestry, yet in many ways Ruby strives to be the antithesis of Python. Continuing with the previous metaphor, one might call this a sibling rivalry.

“Like Perl, Ruby is good at text processing. Like Smalltalk, everything in Ruby is an object...” (Ruby FAQ). Every Ruby program has a ‘root object’ where methods and variables outside of classes are added (Ruby, Wikipedia, Semantics). This allows support for procedural programming, yet this should be discouraged when OO is at Ruby’s core.

Much of Ruby’s popularity is due to the web framework *Ruby on Rails* since many users will install Ruby just to use Rails (Treehouse Blog). This makes Ruby a treat for server side scripting, and even has inspired frameworks like Sails.js, Django, and Laravel (Ruby on Rails, Wikipedia).

Another Ruby web framework is Sinatra. Sinatra is much smaller and simpler than Rails, but it is quite powerful. Thanks to slow internet, it took 241 seconds to download and install Sinatra using Gem, Ruby's package manager. To get a server running, it takes less than 30 seconds (Treehouse Blog). Suffice to say, Ruby is a good choice for web development.

```
require 'sinatra'
# Simple webserver!
get "/" do
  "Hello World!"
end
```

Other popular Ruby based services are Homebrew, a package manager for Mac, and Rake, a task manager and build automation tool that is similar to Make, but allows the the full use of Ruby within Rake's DSL (Rake, Wikipedia).

Being a high level, OO, language, it isn't much of a surprise that Ruby is strong, implicit, and dynamic in it's typing system; in addition, types such as arrays, strings, hashes, and numerics (WikiBooks) are all provided right out of the box which allow users the freedom to define variables simply and elegantly. Excluding integer to float and vice versa, implicit type coercion is not present, but the ability to explicitly cast is (Code Maven).

Just like every other language out there, Ruby supports traditional arithmetic expressions. Support for string expressions are included, so concatenation and multiplication of strings are a piece of cake (Rubyist).

```
    B = "bye"
    H = "hi"
    B += H # "byehi"
    B *= 2 # "byehibyehi"
```

Operator precedence follows PEMDAS for most expressions, yet element reference does take precedence over PEMDAS since we need to have all of our values in play prior to performing a computation. Unary operators take precedence over multiplication, division, addition, subtraction. Following all of those are bit-wise, comparison, and logical operators.

As for selection constructs, Ruby supports if, if-else, if-elif-else, and the case statement. The case statement is similar to the commonly known switch statement, but does not allow fall-through. *Default* is replaced with else. Personally, I find this to be an improvement over the classic switch statement; they are often far too similar to if and if-else statements. Break statements were repetitive to write in, and they were an eyesore. The lack of fall-through can be worked around by abstracting sub-routines even further (Rubyist).

In Ruby, a programmer has many tools at their disposal when it comes iteration. Arrays and hashes have a an *each* method built in which works in tandem with in-line defined iterators. So, traversing through an array is a sinch.

```
X = ["Dogs", "Cats", "Sea Turtles: the only animal that matters"]
X.each { |e| puts e }
```

See, that was almost too easy. Yet, the style of iteration will probably seem mysterious at first to many programmers. All we are doing is allowing ‘e’ to be our current element, and the code following that is meant to be performed on our element. This could have been done with a traditional *while*, but this prevents the programmer from having to define and manage an integer based loop and makes it clear that the array is being looped over (Rubyist). *For* and *while* loops exist with a few added features. The *for* can act as a for-each, for-in-range, or as a traditional for loop. The *while* can be used as one-liner.

Functions that take no parameter can be called without parentheses. When defining a function signature, parameters are not given a type (only a name). Functions are created with the *def* and the *end* keyword; in addition, if no return is specified, then the last thing computed is returned.

```
def add(a,b)
  a+b # returns the outcome of a + b
end
puts add(10,10) # prints 20
```

This can make defining container classes and overloading operators much easier. Another example is a function that is immediately greeted with control/branching statements, e.g. a recursively defined function for calculating factorials..

Scope comes in three flavors: local, global, and instance; each of these have their own naming conventions. Global variables are prepended with a dollar sign, instance variables are prepended with an at symbol, and local variables start with a lowercase letter or an underscore. Local variables have block scope, so if a local variable is defined within an if statement, then said variable only exists within if-block (Rubyist).

Global variables can be accessed from almost anywhere. Although, there are a few caveats; for example, a global variable within a method definition will not be initialized until the method is called. Yet, this segways into instance variables. With an instance variable it is possible to mimic the effects of the static keyword that is found in languages like C++ and Java. Yet, with a bit of creativity one can mimic a static variable.

```
def printStaticVar
  @myVar
  @myVar = 0 if not @myVar
  puts @myVar +=1 # 1, 2, 3, ...
end
```

Although, this isn't quite as safe as the static keyword in other languages, so one must err on the side of caution when using this. This is due to the fact that if @myVar is redefined in another scope, then it is possible that the two will share the same reference. Typically, instance variables are intended to be used as fields for a class.

Local scope is what one might expect when approaching a new language. Typically, two variables within different blocks can share the same name, but they will not have the same memory address.

Ruby offers classes and modules. Classes are used to create objects, allow inheritance, so that we can pass around and manipulate our data more freely. On the other hand is modules, which allow the programmer to define procedures (functions) and constants in one related area. Modules also act as a namespace which greatly increase organization since an entire library of classes, procedures, and constants can be defined within a namespace. Modules can also be used as a mix-in since Ruby does not allow for multiple inheritance (Rubyist).

Some of the constructs may differ semantically, but Ruby offers exceptional exception handling that operates under our preconceived notions. Instead of *try*, we have *begin*. Instead of *catch*, we have *rescue*. Finally, we have *ensure*, which replaces *finally*. Suddenly, exception handling seems much more understandable in Ruby, yet an important question is raised (yet another pun intended). Why do the semantics for exception handling differ and are they better? Personally, I think that Ruby's choice of semantics frame what exception handling is intended for: begin a task, define ways to rescue the task if something goes wrong, but always ensure that a certain procedure always occurs (perhaps closing a file).

Our tour of Ruby has come to an end. Hopefully this inspired whoever read this to try out Ruby. Its interactive shell is great to work with. OO in Ruby is real easy to get the hang of, and short scripts don't give any trouble. So, whether you are developing the web, CLIs, or just trying to automate some tasks, Ruby has something to offer. Be sure to review the code samples above and the sample program found below!

Sample Program:

Note: I am not gifted with extraordinarily high levels of patience. The code formatting was done with a handy addon for Google Docs called CodeBlocks.

On running: Have Ruby installed on your system, open the directory in terminal, and type *ruby ./main.rb*, and voila, a calculator is born.

```ruby
#  Contents: main.rb
#    This main file contains the opening text for the program; as well as
#    a short loop that takes user input then prints the solution. The calculator
#    only calculates simple expressions that involve two operands and a single operator
#    separated by whitespace.


require "./calculator.rb"

calc = Calculator.new
puts "Type exit or use ctrl-c to exit."
puts "All expressions must have two operands and one operator, e.g."
puts "10 + 10. Spaces are required."

loop do
  begin
   puts calc.calc(calc.getInput)
  rescue
    puts "Syntax error"
  end
end
# EOF. See next page!
#  Contents: calculator.rb
#    Contains a class that gets user input and the ability to process
it.


class Calculator

  # A calculator class shouldn't handle user input, but I'll allow
it.
  def getInput
    input = gets
      exit if input == "exit\n"
  end

  # Splits the expression into an array then performs calc.
  def calc(expStr)
    expression = expStr.split(" ")
    operand1 = Float(expression[0])
    operator = expression[1]
    operand2 = Float(expression[2])

    case operator
    when "**"
      return operand1 ** operand2
    when "*"
      return operand1 * operand2
    when "/"
      return operand1 / operand2
    when "+"
      return operand1 + operand2
    when "-"
      return operand1 - operand2
    end
  end
end
# EOF. End of Sample
```

Sample Rake File:

Note: I wrote this to automate the building of a C++ executable. The folder structure is as follows:
Project Directory
- executable
- rakefile
- src/
- header/
- o/

To Build: Navigate to your project's root directory via terminal and run *rake build*.

```ruby
# SOF
flags = "-lSDL2 -lSDL2_image -lSDL2_ttf"
task :build do
  cleanObj
  arr = loadSrc
  sh "clang++ -c #{arr}"
  moveObj
  arr = loadObj
  sh "clang++ -o exe #{loadObj} #{flags}"
end

def loadSrc
  Dir.glob("./src/*").join(" ")
end

def loadObj
  Dir.glob("./o/*").join(" ")
end

def cleanObj
  if not Dir.empty?("./o/")
    sh "rm ./o/*"
  end
end

def moveObj
  sh "mv *.o ./o/"
```

```
end # EOF
```

Works Cited

"About Ruby." *Ruby-Lang*, www.ruby-lang.org/en/about/. Accessed 23 Nov. 2018.

"Rake (software)." *Wikipedia*, en.wikipedia.org/wiki/Rake_(software). Accessed 9 Dec. 2018.

"Ruby (programming language)." *Wikipedia*, en.wikipedia.org/wiki/Ruby_(programming_language). Accessed 17 Nov. 2018.

"Ruby Operator Precedence." *Technotopia*, www.techotopia.com/index.php/Ruby_Operator_Precedence. Accessed 6 Dec. 2018.

"Ruby Programming/Data types." *WikiBooks*, en.wikibooks.org/wiki/Ruby_Programming/Data_types#Ruby_Data_Types. Accessed 6 Dec. 2018.

Maeda, Shugo, et al. "The Ruby Language FAQ." *Ruby-Doc*, ruby-doc.org/docs/ruby-doc-bundle/FAQ/FAQ.html. Accessed 23 Nov. 2018.

Matsumoto, Yukihiro, et al. *Ruby User's Guide*, www.rubyist.net/~slagell/ruby/. Accessed 6 Dec. 2018.

Seifer, Jason. *Treehouse Blog*, blog.teamtreehouse.com/coolest-ruby-projects-ever. Accessed 5 Dec. 2018.

Szabo, Gabor. "Convert String to Number in Ruby." *Code Maven*, code-maven.com/convert-string-to-number-in-ruby. Accessed 6 Dec. 2018.