|  |
| --- |
|  |
| Ruby Programming Language |
| **MTH 135: Structure and Comparison of Programming Languages** |
| **Dr. Starkey** |
| **Cindy Oliver** |
| **12/9/2013** |

History of Ruby

Ruby has had a short yet very interesting history. Creator Yukihiro “Matz” Masumoto has had a lot of accomplishments with this programming language. Masumoto wanted Ruby to be of careful balance as well as to be as simple in appearance but complex and natural in the

inside. He created Ruby by gathering parts from his favorite languages and then forming them into one. The languages he copied from are Perl, Smalltalk, Eiffel, Ada, and Lisp. He took the best of these programs and combined them together to create what is now Ruby. According to Gunner Technology, a web developing company, “Ruby is a general purpose object oriented programming language that is written in C.” Everything is an object in Ruby, any information and code can be given their own properties and actions. Ruby’s object-oriented approach is commonly demonstrated by a bit of code which applies an action to a number.

The idea first came into mind on February 1993, when Masumoto first thought of creating a new program language. Masumoto started to develop Ruby on February 24, 1993 and the first “hello world” program in Ruby ran in the summer of that same year. In December of 1994, the first alpha version was released. His reason for creating a new programming language was because he wanted “a scripting language that was more powerful than Perl, and more object oriented than Python.” During an online chat with one of his friends, they were discussing potential names for the new programming language that was soon going to be created. From the various names they came up with it they narrowed it down to only two: Coral and Ruby. He then decided to go with Ruby because of the precious gemstone. It was short name for a beautiful and highly valued stone, so the name was picked up, and it eventually became the official name of the language.

In December of 1995, Ruby version 0.95 was officially released to Japanese domestic newsgroups. This version, however, didn’t stay this way for long because three other versions later came within the next two days. Then in December of 1996, Ruby 1.0 was officially released to Japanese domestic newsgroups again. It wasn’t until Ruby Version 1.3 in 1999 that it became popular in English speaking countries. At the time, Ruby had its own English mailing called Ruby-Talk, which was a very important turning point for Ruby because it became a growing interest outside of Japan. Six years later, in 2005, Ruby on Rails was created which was basically a web application framework that was entirely written in Ruby. That same year, Apple then decided to ship it with the new Mac OS X operating system, Leopard. Interest in Ruby boomed when it started to transport with the Leopard operating system. In 2011, the latest stable version of Ruby was released; version 1.9.3 is so far the latest version Ruby that was released around the world.

When Ruby was first created online groups and newsgroups were made for the purpose of being able to talk about Ruby and discuss ways to improve this programming language. Ruby has had these online groups, newsgroups and mailing lists primarily to get more feedback from users. Ruby is more known in Japan since its first public release in 1995, there was no English documentation until 1997. There was no publicizing of Ruby in English until the ruby-talk mailing list was started in late 1998. Which is why it is still new to non-Japanese people, in the next three years chances are that will change because the community is growing and English books like *Ruby in a Nutshell* have been published.

Syntax and Semantics

Syntax and semantics in Ruby are quite similar to the ones from Python, Perl and Smalltalk. Ruby is an object-oriented program; every value is an object, even the types that other languages designate as primitives such as integers, booleans, and “null.” Object-oriented programming calls properties by the name instance variables and actions are known as methods. Ruby’s pure object-oriented approach is most commonly demonstrated by a bit of code which applies an action to a number, for example:

irb<main>:001:0> 3. Times do print “Ruby Rocks!” end

Ruby Rocks! Ruby Rocks! Ruby Rocks!

irb<main>:002:0> print “Mount St. Mary’s College Rules”

Mount St. Mary’s College Rules=> nil

In a Ruby program, every command will always begin with “irb<main>” which means it is an interactive program. After the irb<main>, follows the line number, which will always be one for the first line. When there is a new command it will always change to the number line it is supposed to be. After the “001 or 002” follows an additional number, in this case, it is a 0. It is called the depth; however, I will explain when it is used in a different example. The first command in this example is asking for Ruby to print out “Ruby Rocks!” three times. Once end is written into the command the phrase is printed out. The second line is asking Ruby to print out “Mount St. Mary’s College Rules.” It is different from the first command because it is only being printed out once. When it is printed out “=> nil” is appears at the end of it, which means that there are no errors, everything is okay.

In many languages, numbers and other primitive types are not objects. Ruby follows the influence of the Smalltalk language by giving methods and instance variables to all of its types. This eases one’s use of Ruby, since rules applying to objects apply to all of Ruby. However, the difference Ruby has from Python and Perl is that Ruby keeps all of its instance variables completely private to the class and only exposes them through accessor methods.

Ruby is seen as a flexible language, since it allows its users to freely alter its parts; essential parts of ruby can be removed or redefined. For example, addition is performed with the plus (+) operator. But, if you’d rather use the readable word plus, you could add such a method to Ruby’s builtin numeric class, for example:

class Numeric

def plus(x)

self.+(x)

end

end

y = 5.plus 6

# y is now equal to 11

While Ruby often uses very limited punctuation and usually prefers English keyword, some punctuation is used to decorate Ruby. Ruby needs no variable decorations. It uses simple naming conventions to denote the scope of variables.

* Var could be a local variable.
* @var is an instance variable.
* $var is a global variable.

When it comes to constants in Ruby they always start with capital letters; you can change the values of constants, but Ruby will give you a warning. Ruby also includes a standard set of looping and branching constructs: if, while, and case. Ruby keeps an if statement straight as long as the conditional (if…) and the associated code block are on separate lines, for example:

if a < 5then puts “#{a} less than 5” end

if a < 5 then puts “#{a} less than 5” else puts “#{a} greater than 5” end

\*\*\*

The if statement is also an expression; its value is the last line of the block executed. Then the line above could also be written like this:

Puts (if a < 5 then “#{a} less than 5” else “#{a} greater than 5” end)

Variables in Ruby can contain data of any type. You can use variables in Ruby programs without any declarations. Variable name itself denotes its scope (local, global, instance, etc.) A local variable name consists of a lowercase letter (or an underscore) followed by name characters. An instance variable (declared within an object always “belongs to” whatever object self refers to) name starts with an “at” sign followed by a name, for example: @sig, @\_, @Counter. A class variable (declared within a class) name starts with two “at” signs (“@@”) followed by a name, for example: @@sign, @@\_, @@Counter. A class variable is shared among all objects of a class. Only one copy of a particular class variable exists for a given class. Class variables used to the top level are defined in object and behave like global variables. Class variables are rarely used in Ruby programs. Global variables start with a dollar sign (“$”) followed by a name of characters. A global variable name can be formed using “$” followed by any single character. Ruby defines a number of global variables that include other punctuation characters, such as $\_ and $-K.

In Ruby, writing Math functions can be a bit more complicated. For example:

irb<main>:001:0> 2+10

=> 12

irb<main>:002:0> 34-23

=> 11

irb<main>:003:0> 12\*2

=> 24

irb<main>:004:0> 12/4

=> 3

irb<main>:005:0> 13/4

=> 3

irb<main>:006:0> 13.0/4.0

=> 3.25

In this example, the programmer is asking Ruby to do simple Math functions like adding, subtracting, multiplying, and diving. In the first command, the programmer is asking for Ruby to add. It is very simple as you can see because all you have to write in the command are the numbers you want to add and the add symbol. After hitting enter, it will automatically add the two numbers and that is all. The second line is subtracting a set of numbers; just like in addition it will also do the same just like in the first command line. The only time theirs is a different way to write a math function is with division. As you can see in this example a simple division problem that will give you a whole number (or an int) has no specific way you need to write it but when it comes to diving numbers it will give you a whole number instead of a remainder or a decimal. It does this because Ruby gives you an answer depending how you ask the question. What needs to be done is it need to be written as a float, which is a basically a number with a decimal point.

The next way shows an example of how to use variables in Ruby. Using variables in Ruby is simple just like writing a math function. For example:

irb<main>:001:0> x = 20

= > 20

irb<main>:002:0> x + 10

=> 30

irb<main>:003:0> y = 11

=> 11

irb<main>:004:0> y + 2

=> 13

In the first command line we are telling Ruby for ‘x’ to equal to 20. Once we hit enter and now when we write ‘x’ it equals 20. Instead of writing 20, we can just use the letter ‘x’ in equations. Whenever, we use x + 10 it is saying 20 + 10. After writing the equation, Ruby then prints out the answer without any errors. The third command line is an example with the ‘y’ variable instead.

Regular expressions, though cryptic, is a powerful tool for working with text. Ruby has this feature built-in. It's used for pattern-matching and text processing. **A regular expression is simply a way of specifying a pattern of characters to be matched in a string.** In Ruby, you typically create a regular expression by writing a pattern between slash characters (/pattern/). A *regular expression literal* is a pattern between slashes or between arbitrary delimiters followed by %r. The syntax of this would be:

/pattern/

/pattern/im # option can be specified

%r!/usr/local! # general delimited regular expression

**Regular-expression examples:**

Literal Examples:

|  |  |
| --- | --- |
| *Example* | *Description* |
| /ruby/ | Match “ruby”. |
| ¥ | Matches Yen sign. Multibyte characters are supported in Ruby 1.9 and Ruby 1.8. |

Loops in Ruby are used to execute the same block of code a specified number of times. For example, a program that prints all of your new email will need to print each email from a list, not just a single email. To do this, a constructs called loops are used. A loop will repeat the [statements](http://ruby.about.com/od/glossary/g/statement.htm) inside it a number of times until some condition is met. The first type of these loops is a while loop. While loops will execute all of the statements contained within them as long as the conditional statement remains true. For example:

#!/usr/bin/env ruby  
i = 0  
while i < 10  
  i += 1  
end  
puts i

In this example, the loop continually increases the value of the variable *i*by one. As long as the conditional statement **i < 10** is true, the loop will continue executing the statement **i += 1** which adds one to the variable.

Until loops are almost identical to while loops except that they will loop as long as the conditional statement is *false*. The *while* loop will loop while the condition is true, the *until* loop will loop *until* the condition is true.

#!/usr/bin/env ruby  
i = 0  
until i == 10  
  i += 1  
end  
puts i

This example is the functional equivalent of the while loop example, except using an *until* loop, **until i == 10**. The variable is incremented by one until its value equals ten. Though the more traditional while and until loops are used in Ruby programs, closure-based loops are more common. It isn't even necessary to understand what closures are or how they work in order to use these loops; in fact they're viewed as normal loops despite being very different under the hood.

The *times* loop can be used on any variable containing a number or used on a number itself. In the following example, the first loop is run 3 times and the second loop is run however many times is input by the user. If you input 12, it would run 12 times. You'll notice that the times loop uses the dot[syntax](http://ruby.about.com/od/glossary/g/syntax.htm) (3.times do) rather than the keyword syntax used by the *while* and *until* loop. This has to do with how the times loop works under the hood but it's used in the same way a while or until loop is used.

#!/usr/bin/env ruby  
3.times do  
  puts "This will be printed 3 times"  
end   
  
print "Enter a number: "  
num = gets.chomp.to\_i  
  
num.times do  
  puts "Ruby is great!"

end

The *each* loop is perhaps the most useful of all the loops. The each loop will take a list of variables and run a block of statements for each of them. Since almost all computing tasks use lists of variables and have to do something with each of them in the list, the each loop is by far the most common loop in Ruby code. One thing to note here is the argument to the loop's block of statements. The value of the current variable the loop is looking at is assigned to the variable name in pipe characters, which is **|n|** in the example. The first time the loop runs, **then**variable will be equal to "Fred," the second time the loop runs it will be equal to "Bob" and so on.

#!/usr/bin/env ruby

# A list of names  
names = [ "Fred", "Bob", "Jim" ]  
  
names.each do|n|  
  puts "Hello #{n}"  
end

All of Ruby’s data types are objects. The classes for these data types are:

* Boolean
* Number
* String
* Array
* Hashes symbols

A Boolean values represent a yes or no. it is a flag that hold either true or false. Even though it holds true or false, you see Boolean values everywhere. Any place you only see two options, true or false, yes or no, black or white, up or down, it can be represented by a Boolean. In a number, the two most commonly used types are integers and floats. Integers are whole numbers, or numbers that do not have a decimal place or fraction. Integers can be either negative or positive numbers. Strings are one or more characters of letters, numbers, and punctuation. Most of what is presented to the user is a collection of strings with other data types scattered here and there. These characters are enclosed by either single or double quotes. Double quotes allow for string interpolation and escaped characters; characters with the back slash before it is used to represent special meaning or action, like **\n** for newline. Single quotes will keep the special characters and print them with the other characters in the string. Ann array is a collection of values of any type, unlike languages like C and C# where the values in the array have to all be of the same type. Values are stored and retrieved by their index, which is a zero based index. A zero based index starts at zero instead of one and increments from there and an index of 9 would be the 10th value in an array. All values have a unique index. You can think of this as a key value collection by thinking of the array index as the key and the value you add to the array as the value. Hashes, however, similar to arrays, store a collection of values in a key and value fashion. Unlike arrays, hashes do not assign an index for the value you add, but instead use a value you provide as the key. Finally, symbols are light weight strings that are generally used when the string value will not be printed.

Examples of Ruby Programs

**Examples of objects and classes:**

irb<main>:001:0> class Animal

irb<main>:002:1> attr\_accessor :name, :age, :trait

irb<main>:003:1> end

=> nil

irb<main>:004:0> first\_animal = Animal.new

= > #<Animal:0x26e9ce0> (stored in memory)

irb<main>:005:0> first\_animal.name = “Lucky”

=> “Lucky”

irb<main>:006:0> first\_animal.age = 1

=> 1

irb<main>:007:0> first\_animal.trait = “small”

=> “small”

irb<main>:008:0> puts first\_animal.name

Lucky

=> nil

irb<main>:009:0> puts first\_animal.trait

small

=> nil

This example shows how to create objects and classes. In this example, I used animal as the object and name, age, and trait (or attributes) as the classes. Whenever “puts” is written it means the attribute is being printed out.

**Example of an Inheritance and adding attributes:**

irb<main>:001:0> class Animal

irb<main>:002:1> attr\_accessor :name, :age, :trait

irb<main>:003:1> end

=> nil

irb<main>:004:0> class Dog < Animal

irb<main>:005:1> end

=> nil

irb<main>:006:0> class Cat < Animal

irb<main>:007:0> end

=> nil

irb<main>:008:0> class Fish < Animal

irb<main>:009:1> end

=> nil

irb<main>:010:0> class Fish < Animal

irb<main>:011:1> attr\_accessor :type

irb<main>:012:1> end

=> nil

irb<main>:013:0> fish = Fish.new

=> #<Fish:0x2602770>

irb<main>:014:0> fish.name = “Stewart”

=> “Stewart”

irb<main>:015:0> fish.age = 3

=> 3

irb<main>:016:0> fish.type = “Goldfish”

=> “Goldfish”

irb<main>:017:0>

This example shows how an inheritance is written. An inheritance means to add more traits to the objects. In this example, I am putting more traits into “Fish.” This also shows how to add more attributes. This starts on line 11. Unlike before how we inherited the attributes (name, age, and trait), the fish class is different. It will inherit everything from Animal but also the type of fish. On the 13 line, we are creating a new object, which is why we write “Fish.new.” We can finally tell Ruby what the name of the fish is, how old is and the type of fish it is. Let’s say, hypothetically speaking, this database animal program, where people can store all of their pets names, is selling well but one day some customers also want to add the name, trait, and age. This type of Ruby program can be used for this type of situation.

**Methods:**

irb<main>:001:0> class Merida

irb<main>:002:1> def talk

irb<main>:003:2> puts “Hi, my name is Merida!”

irb<main>:004:2> end

irb<main>:005:1> end

=> nil

irb<main>:006:0> bo = Merida.new

=> #<Bucky:0x2727708>

irb<main>:007:0> bo.talk

Hi, my name is Merida!

In methods, it is telling the object what to do. In this example, it is printing a text on the screen. This method has to be inside a class. In order to make a method we need to give it a name or defining (def) a method. All this method is going to do is print out “Hi, my name is Merida!” With the end, it automatically tells the class it wants to end.

**Comparison operators and Decision Making:**

irb<main>:001:0> weight = 140

=> 140

irb<main>:002:0> puts “You need to eat some cheeseburgers” if weight < 150

You need to eat some cheeseburgers

=> nil

irb<main>:003:0> weight = 600

=> 600

irb<main>:004:0> puts “You need to eat some cheeseburgers” if weight < 150

=> nil

In this example, it is showing how Ruby can make a decision. This is a program where the user inputs their weight, whether it is too big or too small we are telling the user to “eat more cheeseburgers.” The first command line is telling Ruby what the weight is, the next command line is we want Ruby to print out if the weight is less than 150.

**Expressions and Comparative operators:**

irb<main>:001:0> a = 3

=> 3

irb<main>:002:0> b = 6

=> 6

irb<main>:003:0> a+b+10

=> 19

irb<main>:004:0> d = 33

=> 33

irb<main>:005:0> f = d

=> 33

irb<main>:006:0> puts f

33

= > nil

irb<main>:007:0> g = 68

=> 68

irb<main>:008:0> g = g+12

=> 80

irb<main>:009:0> g += 12

=> 92

The first two command lines show how to write an expression, which are like equations. Aside from using variables and simple expressions, you can also assign the value of a variable into other variables. Command line 3 to 6 shows you how to write this. It figures out the value on the right hand side and assign it to the variable on the left had side. Command line 7 to 9 then shows how to use shorthand. What this will do it will figure out everything on the right hand side first. g = g+12 is equivalent to g += 12.

**Changing Data types:**

irb<main>:001:0> 13/4

=> 3

irb<main>:002:0> 13.0/14.0

=> 3.25

irb<main>:002:0> a = 13

=> 13

irb<main>:00:0> b = 4

=> 4

irb<main>:00:0> print c.to\_f/d.to\_f

3.25=> nil

In this example, we are asking Ruby to change the data types. We are changing the integer into floats. What we do is use the method called to f. Instead of writing c/d, we use print c.to\_f/d.to\_f to give us the solution.

Evaluation

Readability:

Reading a program in Ruby was what I imagined it would be, complicated. Since Ruby was written based off of different types of programming languages there should be no surprise that learning how to read a program for Ruby is complicated. Since I am one of those people who knew absolutely nothing of the base languages (Perl, Smalltalk, Eiffel, Ada, and Lisp) it was complicated to read all the Ruby language programs, at least when I first saw them. New programmers can be quickly confused and lost when they Ruby programs because they could be different from what they are used to. There is heavy use of the “power features” of the language, like dynamic classes. I often had “where does this happen” questions because it did not make much sense how the use of less code would be easier to read. Although generally less code is better, in the case of ruby, too little code can be confusing since people may not know what the statements actually mean. It helps if you know about and understand all of what happens in the background in order to understand the objects you are reading.

* Ruby is an object oriented language and the object oriented features are used throughout. Without this critical skill, you'll be struggling as a Ruby programmer.
* It helps to have knowledge of the other program languages
* One of the problems of Ruby in readability is feature multiplicity because it is having more than one way to accomplish a particular operation

Writability:

Writability of Ruby is less complicated than it is to read it because of the fact that there is less to type than in other languages. If the programmer follows the correct standard pattern the way they are designed then there should not be any complications while writing the program. However, once again, it does have some complications especially when the person has never written any type of program, in any of the base languages, ever. If a person can easily read and write a code then it will be easy for them to use it. Because in Ruby the attribute of an object can accessed as if they were any other variable it is a bit easier to assign to these variables when a programmer wants to set the value of an attribute. Despite not knowing a lot about the other programs there were still a lot of ways I was able to understand how to write a Ruby program, and then after it was much easier to know how to read the program.

* Writing small programs with Ruby is as easy as importing the modules you need
* Ruby has first-class regular expressions, which makes text processing scripts easy to write
* The primary way of running a Ruby script is from the command-line. Knowing how to navigate directories, run scripts and redirect [input and output](http://ruby.about.com/od/rubyfeatures/a/io.htm) are essential skills to Ruby programmers.

Reliability:

Ruby’s reliability was good; Ruby’s programs performed its specifications under all conditions. In the program examples I did there were no complications while writing the program and then getting what I suppose to get, I rarely got any errors once I was following the steps on how to write a simple example or an inheritance. Ruby’s reliability in type checking is not as complicated compared to other language programs. Ruby only required that an object must always have a capital letter in the beginning, while traits, on the other hand, did not require that. If in case a programmer accidentally forgets to capitalize the first letter then Ruby automatically tells the programmer the error that occurred. Knowing that there is an error right away is helpful because instead of the language letting the programmer at the end would cause a lot of problems and was a waste of time. However, what is most important in reliability is how easy it is to read and write. Ruby, if the programmer is able to understand then it will be easy to both read and write. Since it is easy to write then chances are there program will be correct more often.

* Ruby is suitable for larger software systems
* Ruby has Ruby on rails, which is an application web framework, which is the software which has five major subsystems, numerous minor pieces and a plethora of support scripts, database backends and libraries.
* Ruby allows aliasing there are two pointers set point to the same variable, which is possible in most languages.

Cost:

Like I said before if the programmer has little or no knowledge at all on how to write a program on Ruby it will highly affect the cost, or the function of many of its characteristics, because if the programmer does not know how to write a program on language chances are they will need to be trained on how to do it. At least in this case Ruby was not as complicated to learn so the cost to train programmers to use this language won’t be as much. The cost of writing a program in Ruby is not as bad as it is with other languages. If there is a good programming environment then chances are the cost for training a programmer and writing a program will be low.

* The cost of having to train programmers is very little especially if the programmer is already good at writing programs Ruby is based off of.
* The cost of writing a program in Ruby
* Ruby does not have an expensive implementation system is not expensive or runs on expensive hardware

Resources material

1. “About Ruby”

It gives me the history on Ruby, how it was created and who created it.

<https://www.ruby-lang.org/en/about/>

1. “The Ruby Programming Language”

This article gives me program examples as well syntax and semantics.

<http://www.informit.com/articles/article.aspx?p=18225>

1. “Programing Ruby: The Pragmatic Programmer’s Guide”

This is a guide to how to use Ruby, so it will let me know more about the history and give me more examples.

<http://www.ruby-doc.org/docs/ProgrammingRuby/>

1. “A beginner’s guide to Ruby”

It is the same as the top one, only it gives me more simple examples since it is only a beginner’s guide.

<http://coding.smashingmagazine.com/2012/05/24/beginners-guide-ruby/>

1. “Ruby Programming Tutorial”

Shows beginners how to write a Ruby program.

<http://www.youtube.com/user/thenewboston>