Notes on Ruby

Written by
Mahmoud Saeed
<msaeed.ma [at] gmail [dot] com>

w w w . d i g i t a l k e r n e l . i n f o

Index

1 Introduction

Ruby is a cross-platform, interpreted, scripting, object-oriented, high-level, free of charge, and open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.

Cross-platform:

It runs on several platforms; UNIX, Linux, Windows, Mac OS, ..etc.

Interpreted:

Programs written in Ruby are not translated or “compiles” into the machine code but instead they are indirectly executed (or said to be interpreted) by another software program known as an interpreter.

Scripting:

Can be used to make scripts to control the operation of a normally-interactive program, giving it a sequence of work to do all in one batch.

Object-oriented:

It uses the Object-oriented programming (OOP) paradigm that uses objects which are data structures consisting of data-fields and methods together with their interactions.

Free of Charge & Open Source:

It's licensed under the GNU General Public License (GPL). So, its source code is available for free and you can copy, modify, and distribute it.

2 Standard Types

2.1 Strings

We use the puts() and gets() methods to deal with strings in Ruby.

puts()

The put string, *puts* , method is used to print out a string then it adds a linefeed at the end.
The string is delimited by two single or double quotes.

```
puts( "Hello, Ruby!" )
puts( 'Hello, Ruby!' )
```

Note:

The brackets enclosing the strings after *puts* are optional.

```
puts "Hello, Ruby!"
puts 'Hello, Ruby!'
```

The *print* method gives you the same function, nevertheless it doesn't add a linefeed at the end.

gets()

The get string, *gets* , method is used to read a string from the user. This string can be assigned to a variable.

```
name = gets()
```

Note:

In Ruby, you don't have to pre-declare this variable nor to specify its type. Ruby instead do this for you and infers its type. You also have the choice to use semicolons as statement separators.

```
puts( "Hello" );
```

Note:

Single-line comments is placed after a hash character. But comments which consist of multiple lines are placed between *=begin* and *=end*.

```
# This is a single-line comment

=begin
    This is a comment that
    consists of
    multiple lines
=end
```

2.1.1 Strings and Embedded Evaluation

Simply, you can embed evaluations - such as variables, methods, mathematical expressions, and even bits of program code - which return values into the string itself. But in this case you have to use the double quotes to delimit the string.

This is done by placing the evaluation between two curly braces preceded by a hash character *#{}*.

```
name = gets()
print "Hello, #{name}"
puts "One plus Three = #{1+3}"
# Produces: One plus Three = 4
```

2.2 Numbers

Just assign the numbers to your variables and Ruby will specify their types.

```
myFloat = 1.505
myInt = 112
myNegativeNum = -125
```

You can also convert one variable to another type by applying the following methods to it:

```
flt.to_i
# Convert the float flt to an integer

int.to_f
# Convert the integer int to a float

int.to_s
# Convert the integer int to a string
```

2.3 Ranges

Ranges are widely used and Ruby made it easy to deal with them.

Let's list some possible ranges:

- 0 to 10
- 0000 to 9999
- a to z
- aa to zz

and so on.

One other useful method is *to_a* which converts something into an array. We'll use it to represent our ranges.

```
print (1..9).to_a
```

This line of code will convert the range *1..9* into an array, then will print out its contents, as follows:

```
123456789
```

To improve the representation of this output, we may use the *inspect* method which is defined for all Ruby's objects. It returns a string containing a human-readable representation of the object.

```
print (1..9).to_a.inspect
# Produces: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

As the ring is converted to an array, and arrays are represented in square brackets delimit their elements.

The *p* method is a shortcut for the *inspect* method.

```
p (1..9).to_a
# Produces: [1, 2, 3, 4, 5, 6, 7, 8, 9]

p ('a'..'c').to_a
# Produces: ["a", "b", "c"]

p ('aa'..'ae').to_a
# Produces: ["aa", "ab", "ac", "ad", "ae"]
```

You can apply some methods on ranges. Let's consider a range variable named *myRange* that represents the integers from 1 to 9.

```
myRange = 1..9
```

We'll apply some methods on it.

The two methods *min* and *max* returns the minimum and the maximum elements of the range.

```
print myRange.min
# Produces: 1

print myRange.max
# Produces: 9
```

The *include* method checks if the element is found in the range. If it's found it returns true, and false if not.

```
print myRange.include?(9)
# Produces: true
```

The *reject* method rejects some elements of the range according to an expression.

```
p myRange.reject { |x| x < 3 }
# Produces: [3, 4, 5, 6, 7, 8, 9]
```

The *each* method calls the range once for each element and passes this element as a parameter which is declared between two pipe characters | |.

The *each* method is also used with the collection types such as arrays, strings, and hashes.

```
myRange.each { |x| print x }
# Produces: 123456789
```

To apply multiple lines of code, you have to use the following form:

```
myRange.each do |x|
print " #{x} "
# You can add multiple lines here
end
```

2.4 Regular Expressions

Regular expressions, also known as regex or regexp, are methods for matching strings of texts and also modifying them.

In Ruby, regular expressions are delimited by two forward slashes. Let's try to match the string 'ruby' and test the result using the =~ operator method:

```
puts /ruby/ =~ 'ruby'
```

The match is made and it will return 0 which is an integer represents the character position in the string. If no match is made, 'nil' is returned.

```
puts /r/ =~ 'my ruby'
# Produces: 3

puts /x/ =~ 'my ruby'
# Produces: nil

puts /python/ =~ 'my py'
# Produces: nil
# 'cause the string 'python' is not found
# in 'my py', but part of it
```

So, let's try the following:

```
puts /[python]/ =~ 'my py'
# Produces: 1
```

When we included our string 'python' in square brackets , the match will be made with any one of those characters in the string. So, the second character 'y' is matched in the position number 1.

The character ^ is used to make a match at the beginning of a string, while the $ character is used to make a match at the end.

```
puts /^py/ =~ 'python'
# Produces: 0

puts /^py/ =~ 'my python'
# Produces: nil

puts /on$/ =~ 'python'
# Produces: 4

puts /th$/ =~ 'my python'
# Produces: nil
```

3 Methods, Classes, and Objects

3.1 Methods

A method Is a sequence of command or programming code to do a certain function. It may has parameters and return values.

```
def myMethod(num1, num2)
  return num1+num2
end

print myMethod(5, 6)
# Produces: 11
```

3.2 Local and Global Variables

Local variables begin with a *lowercase character* and they only exist within the scope they are defined in it.

Global variables begin with the *dollar sign character* *$* and they exist anywhere in the program. So, any new assignment to them affects the value of them elsewhere in the program.

```
$globalVar = "I'm global"
localVar = "I'm local"

puts $globalVar
puts localVar

def m1

    puts $globalVar
    # Prints: I'm global

    puts localVar
    # Error, localVar is undefined inside m1
end
```

3.3 Classes and Objects

Inside a class you can create its own methods and variables, then create objects from that class.

Note:
Class name must begin with an uppercase letter.

Let's trace the following code to understand how classes work in Ruby:

```
class Job
    def initialize(title, salary)
        @jobTitle = title
        @jobSalary = salary
    end

    def changeSalary(newSalary)
        @jobSalary = newSalary
    end

    def printSalary
        print @jobSalary
    end
end

engineer = Job.new("Engineer", 6000)
doctor = Job.new("Doctor", 5000)

engineer.printSalary      # prints 6000
doctor.printSalary        # prints 5000

engineer.changeSalary(6500)
doctor.changeSalary(5500)

engineer.printSalary      # prints 6500
doctor.printSalary        # prints 5500
```

We defined a new class *Job*. Then defined three methods inside it: *initialize*, *changeSalary*, and *printSalary*.

when a method named 'initialize' is defined inside a class it's automatically called when a new object (like engineer & doctor) is created using the *new* method. It's called the constructor of the class.

So, we defined the *initialize* method which takes two arguments (title & salary), then it assigns their values to other two variables named *@jobTitle* & @jobSalary.

Note:

Variables begin with the @ character are 'instance variables' because they belong to individual objects and their values differ from one object to another.

Then, we defined a method named *changeSalary* which take one argument (newSalary) and assigns its value to the instance variable *@jobSalary*.

The final method *printSalary* prints the value of the instance variable *@jobSalary*.

We created two objects (engineer & doctor) from the class *Job* using the *new* method. And specified the two parameters *title* and *salary* for those objects of the class.

Then, we called the *printSalary* method once for each object. This method prints out the value of the instance variable *@jobSalary* for each object.

Finally, we called the second method *changeSalary* and specified the value of its parameter to change the value of salary for each object (engineer & doctor). Then called the method *printSalary* again after changing the value of each salary to print out the new salaries.

4 Arrays and Hashes

4.1 Arrays

- Arrays are objects and may deal with several methods
- Delimited by square brackets
- Indexed from 0

Here's a simple example to illustrate how to create an array:

```
myArray = ['mahmoud', 'abraam', 'shereef']
puts myArray[0]   # prints: mahmoud
puts myArray[1]   # prints: abraam
puts myArray[2]   # prints: shereef
puts myArray[3]   # prints: nil
```

myArray[3] is not found. Hence, 'nil' is returned.

It's allowed to create an array that contains different data types and even expressions.

```
myMix = ['string', 1, 1.5, 2+3]
puts myMix
```

Since arrays are objects from the *Array* class, you can also create arrays using the *new* method and specify the number of its elements.

```
p myArr1 = Array.new
# Produces: []
p myArr2 = Array.new(3)
# Produces: [nil, nil, nil]
p myArr3 = Array.new(3, "ruby")
# Produces: ["ruby","ruby","ruby"]
```

4.1.1 Multidimensional Arrays

Simply:

```
myArray = [ ['1', '2', '3'],
            ['4', '6', '6'],
            ['7', '8', '9']  ]
```

Or you can create one dimensional array then add other arrays to each of its elements.

```
myArray = Array.new(2)
myArray[0] = Array.new(2, "a")
myArray[1] = Array.new(2, "b")

p myArray
# Produces: [["a", "a"], ["b", "b"]]
```

4.2 Hashes

In arrays we index our data by specific numbers (0, 1, 2, ...). The hash class allows you to create indexes of any type you wish (Integers, Strings, ...). This is the different between arrays and hashes.

Let's define a new hash named myLanguage with two indexes (“mahmoud” and “abraam”):

```
myLanguage = Hash.new
myLanguage["mahmoud"] = "Ruby"
myLanguage["abraam"] = "C#"

print myLanguage["mahmoud"]
# Produces: Ruby

p myLanguage
# {"mahmoud"=>"ruby", "abraam"=>"C#"}
```

In the previous example we defined a new hash named myLanguage, then entered two elements (“Ruby” & “C#”) and indexed them by two strings (“mahmoud” & “abraam”) rather than numbers.

“mahmod” & “abraam” are the keys of the hash and “ruby” & “C#” are their values.

Using arrays the previous code would be something like the following:

```
myLanguage = Array.new
myLanguage[0] = "Ruby"
myLanguage[1] = "C#"

print myLanguage[0]
# Produces: Ruby

p myLanguage
# Produces:
# ["ruby", "C#"]
```

If no value is assigned to a key, a 'nil' will be returned when you call this key. But you may assign a default value for the hash.

```
myLanguage = Hash.new("not assigned")
myOS = Hash.new

print myLanguage["shereef"]
# Produces: not assigned

print myOS["shereef"]
# Produces: nil
```

5 Loops and Iterators

5.1 For Loops

For loops in Ruby differ than those of many other programming languages. Instead of giving the for loop a starting and an ending value, you give the for loop a list of items and it iterates over them and assigns their values to a loop variable in turn.

Here's a simple example illustrates a for loop that iterates over an array and prints out each item in the array.

```
for i in ['a', 'b', 'c'] do
    print i
end
```

Note:

Using the each method we can do the same job:

```
['a', 'b', 'c'].each do |i|
    print i
end
```

Note:

You may use the range to create the normal for loop used in many other programming languages.

```
for i in (0..10) do
    print i
end
```

5.1.1 Multiple Iterator Arguments

For example, in the case of multidimensional arrays:

```
for (x,y) in [['a', 'b'], ['c', 'd']] do
    print x, y
end
# Produces: abcd
```

5.2 While Loops

You may use one of the following ways:

```
while true
    print "true"
end
```

Or:

```
print "true" while true
```

Here's a simple example:

```
i = 0
while (i < 5) do
  print "#{i} "
  i = i +1
end
# Produces: 0 1 2 3 4
```

5.2.1 While Modifiers

The while modifier lets you write the code between *begin* and *end* before the test condition to ensure that the code will run at least once.

```
i = 0
begin
    print "Hey"
    puts "Heeeey!"
    i = i + 1
end while i < 5
```

5.3 Until Loops

The “until loop” will run as long as the condition is false. It can be thought of as a “while not” loop.

```
i = 10
until i == 5
    print "Yeah!"
    i = i - 1
end
# Produces: Yeah!Yeah!Yeah!Yeah!Yeah!
# The until loop will stop when the condition
# becomes true (i == 5)
```

5.4 Loop

A block of code enclosed by curly braces will loop forever until a break keyword is executed inside the block.

```
i = 0
loop {
    if (i == 5)
        break
    else
        print "Yeah!"
  end
    i = i + 1
}
# Produces: Yeah!Yeah!Yeah!Yeah!Yeah!
```

5.5 Integers as Iterators

Since integer are objects (like many other things in Ruby), there are some methods can be applied to integers.

The following examples will illustrate some of them:

times:

```
3.times do
  print "WOW! "
end
# Produces: WOW! WOW! WOW!
```

upto:

```
0.upto(9) do |x|
     print x, " "
end
# Produces: 0 1 2 3 4 5 6 7 8 9
```

downto:

```
9.downto(0) do |x|
     print x, " "
end
# Produces: 9 8 7 6 5 4 3 2 1 0
```

step:

```
0.step(12, 3) { |x| print x, " "  }
# Produces: 0 3 6 9 12

0.step(12, 3) do |x|
     print x, " "
 end
# Produces: 0 3 6 9 12
```

Iterating over Arrays using the *each* method

```
[ 00, 01, 10, 11 ].each do |val|
print val, " "
end
# Produces: 00 01 10 11
```

6 Conditional Statements

6.1 If Statement

```
if num < 0 then
    puts "negative"
elsif num > 0 then
    puts "positive"
else
    puts "zero"
end
```

The *then* keyword is optional in the previous case. But it's necessary if the if statement is written on a single line.

```
if (num == 10) then puts "ten" end
```

You may also use a colon character instead:

```
if (num == 10) : puts "ten" end
```

There are two methods for testing Boolean conditions:
Using words: and, or, not
Using symbols: &&, ||, !

```
if lang == "Ruby" || lang == "Python"
print "Scripting Language"
```

There's a short-form for if..then..else:
<condition>? <if true do> : <if false do>

```
lang == "Ruby"? puts("R") : puts("Not R")
```

The case equality operator (===) returns true if the value is found in the range and returns false if not.

We may use it as a condition for an if statement:

```
myRange = ('a'..'h')
if myRange === 'k'
    print "k is found"
else
    print "k is not found"
end
# Produces: k is not found
```

6.2 Unless Statement

The *unless* statement can be thought of as an “if not” statement. It will execute the code if the condition is false.

```
lang = "python"
unless lang == "ruby"
    print "It's not Ruby"
else
    if lang == "ruby"
        print "It's Ruby!"
    end
end
# Produces:It's not Ruby
```

6.3 If and Until Modifiers

```
lang = "ruby"
begin
    puts "Ruby!"
end if lang == "ruby"
# Produces: Ruby

begin
    print "It's Ruby!"
end unless not lang == "ruby"
# Produces: It's Ruby!
```

6.4 Case Statement

Here's an example with several possible cases:

```
case(x)
  when 0 : puts "It's Mahmoud"
  when 1 : puts "It's Shereef"
  when 2, 3 :
    puts "It's either Kareem or Marc"
  when 4, 5 :
    puts "It's Abraam" if x == 4
    puts "It's Amin" if x == 5
    puts "It's one guy who uses C#"
  when (6..16) :
    puts "It's someone out there"
  else
    puts "It's an alien!"
end
```