Ruby First Note Chapter

Garen Ikezian

Inspired from RubyMonk. I've loved the website since then and I thought it could be a good idea to share my notes publicly. Enjoy!

It is a work in progress

1.0 Strings Basics

Strings are objects in Ruby. Unlike Python, they're mutable (like PHP).

1.1 Double Quotes and Single Quotes

You can have it `''` or `""`

```
'This is a string'
"This is also a string"
```

The difference between `""` and `''` is that `""` can interpret escape characters (like `\n`) while `''` do not.

1.2 Determine the Length

To determine the length of a string, we use the `length` method:

```
irb(main):001:0> sentence = "this is a sentence"
=> "this is a sentence"
irb(main):002:0> sentence.length
=> 18
```

Note that the parentheses `()` is optional when you call a method.

1.3 String Interpolation

Like Python using f-string, Ruby also has string interpolation. We use `#{}` as a placeholder for our variables to represent something.

```
def say_name(name)
    puts "Your name is #{name}"
end

#If name is John, the function will print:
#"Your name is John"
```

1.4 Sub-string functions

There are: - `include?` - `start_with?` or `starts_with?` - `end_with?` or `ends_with?`

To determine if the specified sub-string is included in a string, we use the `include?` method. The `?` is to show that `include` returns a boolean value.

```
#Note how the strings are case-sensitive
irb(main):001:0> "Apples, oranges, pears".include? "apples"
=> false

irb(main):002:0> "Apples, oranges, pears".include? "oranges"
=> true
```

```ruby
#These are also case-sensitive

# Prints: True
puts "Ruby is a beautiful language".start_with?("Ruby")

# Prints: True
puts "I can't work with any other language but Ruby".end_with? "Ruby"
```

1.5 Finding the index of a Character

We use the `index` method determine the index of a character in the string. It returns us the index wth 0 being inclusive.

```ruby
puts "I am a Rubyist".index("R")
#Prints: 7
```

1.6 Manipulating Strings

There are different ways to manipulate a string in Ruby:

If we wish to capitlize every letter in our string, we use the `upcase` function. Otherwise, we use the `downcase` function to lowercase every letter. There's also the swapping letters option with `swapcase`.

```ruby
puts 'This is LOWERCASE'.downcase
# Prints: this is lowercase

puts 'oh my god'.upcase
# Prints: 'OH MY GOD'

puts "ThiS iS A vErY ComPlEx SenTeNcE".swapcase
#Prints: tHIs Is a VeRy cOMpLeX sENtEnCe
```

1.7 Difference Between `puts`, `p`, and `print`

- `p` prints out the object's "raw" value. It is useful for debugging purposes especially when you want to see escape characters in the console.
- `puts` will treat the argument as a string (also adds a newline character). If the passing argument is a array, it will display its elements one-per-line by calling the `each` method as each element gets converted with the `to_s` method.
- `print` also treats the argument as a string (like `puts`, with the newline). But when the passing argument is an array, it will NOT display its elements one-per-line.

1.1 Strings Advanced

The `split` function splits words (where the delimeter is a white-space character by default) and forms them into an array.

1.1.0 The `split` function

```ruby
typeVar = "Hello, are you here?".split
puts typeVar << "\n" #mutating the string (not appending like in +)
puts typeVar.class
```

1.1.1 Concatenating

The `+` operator does not modify the original string. Instead, it creates a new string 'JohnField'. Take note if you're dealing with large scale string manipulations.

Just like `<<`, the `concat` method does not involve creating a new string.

`<<` is an alias of `concat`.

```ruby
puts 'John' + 'Field' #This is slow concatenation

puts "Elvis".concat("Presley")
```

1.1.2 Replacing a substring

There are different methods to replace a substring. The methods are:

- `sub`: Replaces the first occurrence of the specified substring (1st arg) to a new one (2nd arg)
- `gsub`: (Think of "global substitution") Replaces all occurrences of the specified substring

They both return a modified string.

We use the `sub` method to replace a the first occurrence of a specified substring (first arg) to a new one (second argument).

```
last_name = "Sparrow"
puts "I am Jack Sparrow.".sub("Sparrow", "Black")

puts "Society & liberty".gsub("y", "é")
```

You can also use a regex (regular expression) to replace substrings.

```
#Here we're replaceing every capital to 0
puts 'RubyMonk Is Pretty Brilliant'.gsub(/[RMIPB]/,'0')
```

1.1.3 Finding a substring using RegEx

With the `match` method, we can find a substring based on our passing regEx argument. We use regEx if we can't find the exact substring (kinda like querying a database)

```
'RubyMonk Is Pretty Brilliant'.match(/ ./, 9)
```

1.2 Boolean and Logical Expressions

1.2.0 Boolean and Logical Operators

Boolean expressions in Ruby are just like in other programming languages. `==`, `!=`, `<=`, `>=`, `>`, `<`, `||`, `&&`, `(`, and `)`.

1.2.1 The if-else construct

Since Python relies on identation, you need a `:` to make scopes viable. In Ruby, you don't need `:` as it follows a Pascal-like scoping tradition.

```
def check_age(age)
    if age < 12
        puts "You're preteen. You are #{age}"
    elsif age < 19
        puts "You're a teenager. You are #{age}"
end

name = gets.chomp #chomp is to ignore the newline character
check_age(age)
```

Ruby also has an `unless` keyword to check for a negative condition. It is synonymous to `if !x`.

```
age = 18
unless age >= 16
    puts "You cannot drive as your are under 16 years old"
else
    puts "You can drive! Get ready drivin'!"
end
```

1.2.2 Ternary operator

Just like any other programming languages, the ternary follows the same syntax.

```
expression ? expression is true : expression is false
```

```
10 == 0 ? true : false
```

1.2.3 Different ways to express `true` and `false`

Unlike C where `1` means true and `0` means false, Ruby treats both as true. The only objects that are false are `false` and `nil`.

See how 0 is treated as true in Ruby:

```
if 0
    puts "0 is true in Ruby"
end
```

1.3 Loops Introduction

1.3.0 Infinite loops

It is self-explanatory. It is pretty different. This is what it never-ending loop looks like in Ruby.

```
loop do
    puts "this line will be executed infinitely"
end
```

It also has the `break` keyword .

```
#The following will not compile as the me object does not exist
#It's only here to demonstrate a point
loop do
    me.run
    if me.tired?
        break
    end
end
```

The for loop in Ruby can also syntactically different. It is structured very much like in Japanese. If you wish to say "I drink twice" in Japanese, you would say "twice I drink".

In Ruby, it is "`N` times I `statement`"

```
5.times do
    "I'm jumping"
end

#If you want to specify the number of times, we add ||
5.times do |i|
    "I'm jumping #{i} times"
end
```

```
def ring(n)
   i = 0
   n.times do
      p "ringing #{n} time(s)"
      i += 1
       if i >= 6  # use >= instead of <
          break
       end
   end
end
times = 0
ring(times)
```

But of course, you couldn't go wrong without typing `for` as long as there's something iterable to loop for (same as in Python).

```
array = [1,2,3,4]
for i in array
    puts "At index #{i}"
end
```

We can also do a for each loop. The following is the same as above but it is just done differently.

```ruby
array = [1,2,3,4]
array.each do |i|
    puts "At index #{i}"
end
```

1.4 Arrays

1.4.0 Indices

Arrays (like Python) are represented with square brackets (`[]`). `Array.new` is an alias of `[]`.

Arrays in Ruby can be a combination of different types (strings, integers, floats ,etc) (just like Python but not like C).

In any C-based languages, we look up the value of an array as a variable by their index like `arr[i]`. In Ruby, we can take the array and type two more enclosed square brackets with an index inside like this:

```ruby
#This will print 3
puts [1,2,3,4][2]
```

Pretty interesting huh?

Just like in Python, we can also start backwards from right to left. We use a negative number as an index to look for our value in the other direction. This is called "reverse index lookup".

```ruby
puts [1,2,3,4][-0]
puts [1,2,3,4][-1]
```

1.4.1 Growing the Array

Unlike C, the size of the array is not fixed, We use `<<` to append any datatype to our array. We can only use the insetion operator `<<` when we want to add a single element in the array.

```ruby
puts [3,1,4,1,5] << "Pi is irrational dude"
```

We can use the `push` method to add more than one elements in the array

```ruby
puts [8,2,5].push("Some random string", 0, 'Yeah')
```

1.4.2 Changing the Array

There are different way to change the contents of the array, one that is based on a condition and one without.

We use `Array#map` and/or `Array#collect` (they're the same) to iterate through every element and commit the action specified in `{}`.

Just as much as there is a ternary operator for an if else block, there is a `map` method for arrays as a shorthand for a for loop

```ruby
puts "With map"
puts [1,2,3,4].map {|i| i + 1 }

puts "With collect"
puts [2,4,6,8].collect{|i| i/2}
```

Meahwhile the `Array#select` needs a condition (boolean expression), not a statement.

`Array#select` is kind of like a shorthand for a while loop while `Array#map`/`Array#collect` is like a shorthand for a for loop.

```ruby
names = ['rock', 'paper', 'scissors', 'lizard', 'spock']
puts names.select { |name| name.length > 5}
```

1.4.3 Deleting the elements

We use `Array#delete` to delete every occurrence of a given argument.

```ruby
arr = [1,3,4,6,1]
p arr
puts "Deleting all elements with value 1"
deleted_number = arr.delete 1
puts "Deleted: #{deleted_number}"
```

```
    puts
    p arr
```

If we wish to make a condition before deletion, we use `Array#delete_if`

```
    p [1,5,10,2,6,5].delete_if{ |i| i % 5 != 0 }
```

1.5 Hashes

1.5.0 Syntax and Fetching Values

More Info: https://ruby-doc.org/3.2.2/Hash.html

There are two ways to create a hash. One is by using a hash rocket `=>` and the other is by using a JSON-style colon `:`.

With a traditional syntax, you can explicitly state the key as a symbol with the colon `:` before it or using double quotes `""` instead. A symbol in Ruby is kind of like a const string and a reference in one. Instead of having multiple strings in a hash, using symbols would be more optimal.

```
#This is invalid syntax. In a traditional syntax, the datatype for the
#key must be specified as a symbol with : or a string "".
=begin
    my_friend = {
        name => "John",
        age => 19
    }

    p(my_friend[name])
=end

#This is valid (recommended)
my_friend = {
    :name => "John",
    :age => 19
}

#This is also valid
my_friend2 = {
    "name" => "Alex",
    "age" => 20
}

puts("Printing my_friend and my_friend2...")
p(my_friend[:name])
p(my_friend2["name"])

puts()

student_ages = {
    "Jack": 30,
    "Jill": 22,
    "Bob": 18
}

 #Accessing the property of a hash is different in ruby
 #In Ruby we use [] just like an array
 puts("Printing student_ages...")
 p()
 p(student_ages[:Jack])
 #This will return nil. It needs to be a symbol regardless of the syntax
 p(student_ages["Jack"])
```

Fetching a value is indexing an array in Ruby. We use square brackets `[]` to retrieve a value from our hash exemplified above.

1.5.1 Modifying a Hash

It is no different from changing an element of an array by its index.

```
menu = {
    "Rice" => 10.5,
    "Chicken" => 25.99,
    "Tea" => 5.5,
    "Bread" => 1.5,
    "Pepsi" => 2.5
}

puts("menu before")
p(menu)
menu["Pepsi"] = 3.5
puts()
p(menu)
```

1.5.2 Iterating over a Hash

If you wish to iterate over both the keys and their values, the `each` method can do the trick as it returns both the keys and their values.

```
restaurant_menu = {
    "Ramen" => 3,
    "Dal Makhani" => 4,
    "Coffee" => 2
}

#One application is to increase the values of the hash
#restaurant_menu
restaurant_menu.each do | item, price |
  get_price = restaurant_menu[item]
  new_price = get_price * 0.1 + get_price
  restaurant_menu[item] = new_price
end

restaurant_menu.each do | item, price |
    puts "#{item}: $#{price}"
end
```

If we want to extract only the keys, use call the `keys` method. For values, we call `values`.

```
person = {
    :name => "John",
    :age => 19
}

#printing values and keys
p(person.keys)
p(person.values)
```

1.5.3 Another way to create a Hash

You can use a hash by calling `Hash[]` link

```
#An initialized hash object
fruits = Hash[:apple, 2, :orange, 3]

#An empty hash
empty_hash = Hash.new

p fruits
p empty_hash
```

1.6 Class

1.6.0 How are they built?

You cannot have an object-oriented programming language without classes. In Ruby, classes can be built as such:

```
class Triangle
    #def initialize(length, breadth)
    #   @length = length
    #   @breadth = breadth
    #end

    def perimeter
        @length + @width + @hypotenuse
    end
end

Triangle.perimeter
```

Variables with `@` become the instance variable of the class. This is what can help distinguish objects with the same datatype.

Calling a method without instance variables initialized will fail the code.

We need a constructor which is represented like a method called `initializer`.

```
class Triangle

    def initializer(length, width, hypotenuse)
        @length = length
        @width = width
        @hypotenuse = hypotenuse
    end

    def perimeter
        length + width
    end
end

triangle = Triangle.initializer(2,1,4)
peri = triangle.perimeter
p peri
```

Note that the perimeter method has no `return`. This is because (like Python), if no return is specified, the last line will automatically be treated as a return value.

1.6.1 Grouping Objects

We can look the class of any object by calling the `class` method.

```
    p 1.class
    p "".class
    p [].class
```

Then there is the method `is_a?` made to be conditional like so:

```
    #Is 1 an Integer type?
    p 1.is_a?(Integer)
    #Is 1 a String type?
    p 1.is_a?(String)
```

Classes are deemed as inferior to objects. But in Ruby, they are just synonyms to objects. Calling the class method repeatedly demonstrates this.

```
p 1.class.class.class.class
```

1.6.2 Everything has a class. Including nil (NilClass).

```
def do_nothing
end

puts do_nothing.class
```

This prints out NilClass because there must always be an object one way or another. NilClass is an object representing the absence of an object.

1.7 Methods

1.7.0 The Basics

At a basic level, methods in Ruby are just like in Python. You can call methods inside another methods as usual.

```
def add(x,y)
    return x+y
end

def minus(x,y)
    return x-y
end

#Just like in Python. We can pass in default value to a parameter
def divide(x,y=1)
    return y != 0 ? x/y : nil
end

def multiply(x,y)
    product = 0
    temp = 0
        y.times do
            puts "#{temp}, #{product}"
            product = add(product,x)
            temp += 1
        end
    return product
end

x=8
y=2
minus(x,y)
divide(x,y)
p multiply(x,y)
```

1.7.1 Objectifying Methods

As the subtitle suggests, in Ruby, the `method` method can hold the reference of the method specified. That way, we can create an object that can do one particular task only.

```
#prints the reference to the method next
p 1.method("next")

oneUpper = 9.method("next")
#prints Method
p oneUpper.class
#prints 10
p oneUpper.call
```

1.7.2 The splat operator * and the inject() method

Parameters in Ruby can be interpreted as a list. We use a unary *splat operator* * that can enable any number of parameters.

But how will we iterate such a list?

By using the `inject()` method, we can iterate a list of passed arguments. It's just another workaround for `each()`

Notice how the value `0` of the `sum` variable for the `each` section is the same as that to the argument of inject (`.inject(0)`).

Also note the `sum` variable in the two blocks. In the inject method, the sum is inside the block parameter (`||`) while the `each` method has the `sum` outside of the block parameter. **0** and **sum** are here to illustrate this.

```
#With each
sum = **0**
(1..10).each{ |i| **sum** = sum + i }
p.sum

#With inject
p(1..10).inject(**0**){ |**sum**, i| sum + i }
```

```
i = 1
numbers_array = []

#It takes a list of numbers and adds them up one by one
def add(*numbers)
    numbers.inject(0){|sum, number| sum + number}
end

#We can either do this
#puts add(1)
#puts add(1,2,3)
#puts add(1,2,3,4)

#Or make it more convenient
5.times do
    numbers_array = numbers_array.push i
    p numbers_array
    #* is to mean, "this array will be iterated"
    p add(*numbers_array)
    i = i.next
    puts ""
end
```

1.7.3 Passing a hash to a function (as a parameter)

In Ruby, passing a hash to a function needs to be explicit in the function signature. We use the curly `{}` braces to make the parameter's type identifiable.

```
#We're passing options (this is what the method sees)...
=begin
    options = {
        :reciprocal = true,
        :precision = 2,
        :round = true
    }
=end

#to this method
def divide(num1, num2, options = {})
    quotient = num1 / num2
    quotient = 1/quotient if options[:reciprocal]
    quotient = quotient.round(options[:precision]) if options[:round]
    return quotient
```

```
end

puts divide(1.0, 2.0)
puts divide(2.0, 4.0, reciprocal: true)
puts divide(22.0, 7.0, precision: 2, round: true)
```

1.8 Lambda, Procs, and Blocks

Now this is the confusing part, so hear me out.

If there's a part that you don't understand. It's not the end of the world. You can skip some parts and call it a day. But I highly advise to play around with the code regardless.

1.8.0 Blocks

Blocks in Ruby are anonymous functions without a name. They can be passed into methods and are usually temporary. They are represented with `do..each` for multi-line blocks and `{}` for single-line blocks.

The syntax for blocks is as follows:

```
object.method{ *block* }

..OR..

object.method do
     *block*
end

..OR..

def method
     **yield**
end
method { *block* }
```

Blocks can also accept an argument(s)

```
#single-line {}
[1,2,3].each { |num| puts num }

#multi-line do..each
['a','b','c'].each do |num|
    puts num
end

#Where num1 is the number from the first array
#num2 is the number from the second array
[[2,4],[1,3]].each { | num1, num2 | puts num1, num2  }
```

One interesting thing about blocks is the `yield` keyword. `yield` enables blocks in Ruby methods.

```
def some_method
    yield
end

some_method { puts "That's me" }
```

Here is a way to pass an argument to a block inside a method.

```
def n_squared
    puts yield(2)
end

n_squared { |n| n*n }
```

Implicit vs Explicit blocks Blocks can be implicit or explicit.

Explicit blocks have a name while implicit blocks don't (we only use `yield`).

However, note that when you make a block explicit, they become a proc. we use the `&` operator to treat the block as if it was actually a proc. There can be any name beside `&`.

```
def print_hello(&block)
    block.call
end

print_hello { puts "hello" }

def print_hi(&b)
    b.call
end
print_hi { puts "hi" }
```

1.8.1 So, what are Procs?

Procs are closely related to blocks. They are a more versatile and **explicit** version of blocks that also encapsulate blocks of code. Unlike (implicit) blocks, they can be stored in variables and they always have a name. They provide the ability to be invoked independently.

```
#Creating proc with the Proc class constructor
process = Proc.new { puts "From process" }

#Creating proc with Kernel#proc method (shorthand of Proc.new)
another_process = proc { puts "From another_process" }

#Explicit blocks are procs
def method_to_call_proc(&process)
    process.call

    puts process.is_a?(Proc)
end

method_to_call_proc { puts "From method_to_call_proc: This is proc"}
```

They have a return statement that exits their *surrounding* method.

For example:

```
def foo
    process_print = proc { puts "puts: ok bar"}
    process_return = proc { return "return: happy bar" }
    process_print.call
    process_return.call
    return "sad bar" #never reaches sad bar
end

puts foo()
```

Notice how the proc treats return differently. Unlike methods/lambdas, `return` returns from `foo` instead of its own block. That's how `return: happy bar` is printed to the console

Procs are also less strict on the types and the number of arguments passed.

```
point = proc { |x, y| "x=#{x}, y=#{y}" }

#passing two int args
puts point.call(1,2)
#x=1, y=2
```

```ruby
#passing three int args
puts point.call(1,2,9)
#x=1, y=2

#passing one int arg
#y is nil
puts point.call(1)
#x=1, y=

puts "Passing an array:"
#The array disintegrates into two ints
puts point.call([1,2])
#x=1, y=2

puts point.call([1,5,3])
#x=1, y=2
```

Procs come in two flavours. There are lambda procs and there are non-lambda procs.

```ruby
#Proving that lambdas are procs
isProcLambda = lambda { puts "Am I lambda?" }
puts isProcLambda.is_a?(Proc) #true
```

We've touched on non-lambda procs. Let's move on to lambdas.

1.8.2 Lambda

Lambdas (like other programming languages) are also methods without a name and is one flavour of proc. Lambdas have their own scope, and the value of the last expression serves as the return value for the lambda itself. Like procs, we can use double pipes `||` syntax to pass arguments to a lambda.

```ruby
fruitSays = lambda{"Don't eat me"}

sayName = lambda do
    return "John Smith"
end

#The passing parameter is "fruit"
fruitsPicked = lambda do |fruit|
    if fruit == "apple"
        return "apple"
    else
        return "orange"
    end
end


#calling single-line lambda
puts "The fruit said: \""+fruitSays.call+'"'

#calling multi-line without arg and multi-line with arg
puts sayName.call() + " picked an " + fruitsPicked.call("fruit")

square = lambda { |x| x+(x * 0.13) }
puts "John paid $"+square.call(2).to_s()
```

Unlike regular procs, lambdas have their own scope. They behave just like methods. They are like "mini-methods" in Ruby.

```ruby
def foo
    process = lambda{ return "happy bar" }
    process.call
    return "very happy bar" #reaches "very happy bar" since it's a lambda
end
```

```
    puts foo() #Prints "very happy bar"
    #Doesn't print "happy bar" because it only returns from lambda
    #(just like regular methods)
```

Extra

Here is some code to play around with. It is tough to hang to it at first but I advise to get your hands dirty on this.

What you'll notice is that the method ends until all its blocks are executed.

```
    def print_once
        yield
        puts "From print_once"
    end

    print_once {puts "From print_once"}

#Output
=begin
    From print_once
    From print_once#
=end


    def print_thrice
        yield
        puts "From print_once thrice"
    end

    print_thrice { puts "From print_thrice once"}
    print_thrice { puts "From print_thrice twice"} #yield can "see" this

#Output
=begin
    From print_thrice once
    From print_once thrice
    From print_thrice twice
    From print_once thrice
=end
    def print_thrice
        yield
        yield
        yield
    end

    print_thrice { puts "From print_thrice once"}
    print_thrice { puts "From print_thrice twice"}
    print_thrice { puts "From print_thrice thrice"}

#Output
=begin
    From print_thrice once
    From print_thrice once
    From print_thrice once
    From print_thrice twice
    From print_thrice twice
    From print_thrice twice
    From print_thrice thrice
    From print_thrice thrice
    From print_thrice thrice
```

```
=end

def one_two_three
  yield 1
  #yield 2
  #yield 3
end

one_two_three { |number| puts number * 10 }

#Output
#10
```

Here, you can better see how blocks become an object inside an object. At least the following demonstrates that:

```
class MyObject
  def call
    puts "This is an instance of MyObject"
  end
end

#block = proc { puts "This is a proc object" }
another_block = Proc.new { puts "This is another proc object" }
block = Proc.new{ puts "This is a proc object"}

myobj = MyObject.new

#the method foo, has to accept an argument
def foo(obj)
  puts "Entering foo."
  obj.call
  puts "Leaving foo."
end

#Executing here...
# call foo with our "block" object (notice how we're passing an argument)
foo(block)
#foo(another_block)

# call foo again but with a different proc object
foo(proc { puts "This is our second proc!" })

# call foo but using our instance of MyObject
foo(myobj)

=begin
    Entering foo.
    This is a proc object
    Leaving foo.
    Entering foo.
    This is our second proc!
    Leaving foo.
    Entering foo.
    This is an instance of MyObject
    Leaving foo.
=end
```

Summary

Blocks in ruby are either implicit or explicit (proc)

Procs are either lambda or non-lambda.

1. implicit blocks are not procs
2. explicit blocks are procs
3. implicit blocks are objects
4. explicit blocks are objects

```
def is_block_proc_implicit
    puts "Are blocks procs (implicit): "
    puts yield.is_a?(Proc) #returns false
end
is_block_proc_implicit { }

puts "--------------------------------"
def is_block_proc_explicit(&block)
    puts "Are blocks procs (explicit) "
    puts block.is_a?(Proc) #returns true
end
is_block_proc_explicit {  }

puts "--------------------------------"
def is_block_object_implicit
    puts "Are blocks objects (implicit) "
    puts yield.is_a?(Object) #returns true
end
puts is_block_object_implicit {  }

puts "--------------------------------"
def is_block_object_explicit(&block)
    puts "Are blocks objects (explicit) "
    puts block.call.is_a?(Object) #returns true
end
puts is_block_object_explicit {  }
```

5. Lambda is a proc
6. Lambda is an object

```
def is_lambda()
    is_lambda = lambda {}
    puts is_lambda.is_a?(Proc)
    puts is_lambda.is_a?(Object)
end

puts is_lambda()
```