Functions in Ruby

If you're more familiar with a language like Java, you might be more used to calling a method that is outside of a Class a *function*. In Ruby, we call *both* of these things methods. This is because, technically, a “function” method in Ruby is still part of a Class (that class is either `Method` or `UnboundMethod`). For the purposes of this tutorial, we will be discussing methods that behave like functions do in languages like Java: that is, they are not attached to a *user*-defined Class.

Declaring a Method

The syntax for declaring a method in Ruby is as follows:

```
def methodName(parameter1, parameter2)
      whatever the method does here
end
```

`def` tells the compiler where the method begins. `end` likewise indicates that the method is completed. Parameters are separated by commas. If there are no parameters, simply leave that area blank, like so:

```
def methodName
      whatever the method does here
end
```

This syntax is similar to how functions are declared in Python, with the exception of `end`.

Method Parameters

As noted above, a method does not have to have parameters. If it does, however, there are a few important things to keep in mind:

1. We cannot type check parameters. This means that we cannot explicitly say what variable type is passed to a method. For example, in Java, we might write a method like so:

   ```
   public myMethod (int x) { some code here; }
   ```

   In the example above, we can see that methods declared in Java allow parameters to have their type defined: `x` is defined to be type int. If an Integer is not passed as the parameter of myMethod, an error will be thrown. In Ruby, however, this is not the case. For example, in the method below, `x` could be any type (String, Integer, Double):

   ```
   def myMethod (x)
         some code here
   ```

```
end
```

This means we have to be careful with how we write our method code. This *also* means that you can put in any combination of data types, unless the code within your method checks for type. For example, your method could contain an if statement that ends the method if the data type inputted is a String.

2. We can set default parameters, so that our method *always* does something. For example, we can write the following method:

```
def say(words=”Hi”)
      puts words
end
```

We can then call the method in two ways:

```
say() #prints “Hi”
say(“Hello”) #prints “Hello”
```

3. Ruby does not require parenthesis to be used when calling methods. Using the method above as an example, both ways of calling the method `says` are appropriate:

```
says(“Hello”)
says “Hello”
```

Method Order

It is important to remember that a method cannot be called before it is defined. The order that Ruby code runs won’t be able to hop around, look for the method definition, and return to your code. So you cannot, for example, do this:

```
myMethod “SomeParameter”

def myMethod (parameter)
      some code here
end
```

You can, however, call methods that have not yet been defined inside of other methods. For example you can do this:

```
def myMethod1
      myMethod2
end

def myMethod2
      puts “We did it”
end
```

```
myMethod1
```

This is because even though `myMethod1` calls `myMethod2`, we are not calling `myMethod1` until after our methods have been defined.

There are more rules for method order regarding methods inside of Classes, but remember, we are only focusing on methods outside of user-created Classes right now.

Recursion

Ruby does support recursive methods! A recursive method is a method that calls itself. A great example of this is a method that calculates factorials.

A factorial multiplies every number starting with 1, to a target number. So if we are trying to find factorial 4, we would multiply 1 * 2 * 3 * 4. We could do this with a loop, but we can use fewer lines of code by using a recursive method. Here is a method that calculates factorials:

```
def factorial(n)
      return 1 if n <= 1
      n * factorial(n-1)
end
```

What this code actually does is it takes in parameter `n` and then multiplies `n` by `factorial(n-1)`. So it is calling the method, `factorial`, inside of itself! It does this until `n = 1`. Then, it follows the line of calls back up the stack and calculates the final result. Let's do an example.

```
factorial 3
```

If we run this, what happens is:

`factorial 3` calls `factorial 2. Factorial 2` calls `factorial 1.` Since `n` is equal to 1, we now move back up the call stack. `factorial 1` computes `n * factorial(n-1); n` is currently equal to 1, so we get 1, and send that to `factorial 2. factorial 2`, which has an `n` value of 2, computes `n * factorial(n-1)` again with *its* `n` value; it gets a result of 2. `factorial 2` now sends that result to `factorial 3,` which *also* computes `n * factorial(n-1)` with its `n` value of 3: it gets a result of 6. Now it returns 6, because it is back at the original call of `factorial 3.`

You can see this in action on the accompanying file, methods.rb.

Return

So far, we have looked at examples of methods that print output using `puts`. What if we want it to return a value that can be saved into a variable? We can do that using the statement `return`. Here is an example:

```
def return8
      return 8
end

value = return8
```

This very simple function takes no parameter, and returns the integer 8. We then stored the result of the method in the variable, `value`. If we were to print the output of `value`, we would get 8! You can see this in action on the accompanying file, methods.rb.

However, we don't actually *have* to use the statement `return` at all; by default, a Ruby method will return the value that resulted from the last evaluated statement. This is how the above `factorial` method returns 6, and not 1. Here is another example:

```
def return8
      4 + 4
end
```

`return` is still useful, however, since it lets us specify what we want to return, and allows us to return the function early, such as in `factorial`.

One thing Ruby *cannot* do is return multiple values. Every method returns exactly one thing. If we write this:

```
def return8
      return 8
      return 5
end
```

Our returned value will always be 8; it won't even look at `return 5`. If we wanted to return *both* 8 and 5, we would have to either print them out, or store them into an array and return the array.

Pass-By-Reference or Pass-By-Value

Languages tend to be either pass-by-reference or pass-by-value type. This refers to how parameters and arguments in methods are handled. Pass-by-reference means that the arguments of a method are references to the variables that were passed into that method. Changing an argument changes the original variable. In pass-by-value a copy of the value passed to the method is made, meaning there are two variables with the same value; changing the argument therefore does not change the original variable.

Ruby is a complex case, because of the way it handles variables. All values in Ruby are actually ObjectsThis means that the variables themselves are references to those Objects. For example:

```
x = 5
```

x is a reference to an Object with the value of 5. This makes it sound like Ruby would be pass-by-reference… but it is closer to pass-by-value.

Let's take a look at the code below.

```
def testPassType(argument)
      argument = "Hey"
end

testValue = "NotHey"

testPassType testValue
puts testValue
```

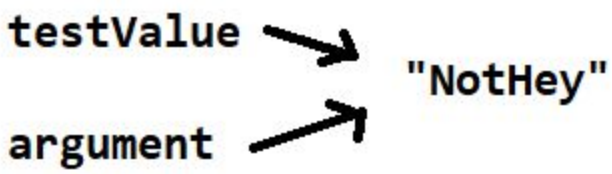
The method above takes an argument, and then reassigns it's value to "Hey". We create a String, `testValue` with the value "NotHey" and pass it to `testPassType`. Then we print `testValue`, to see if it's value changed.

If you run this code, however, the output will be "NotHey". However, if you print this instead:

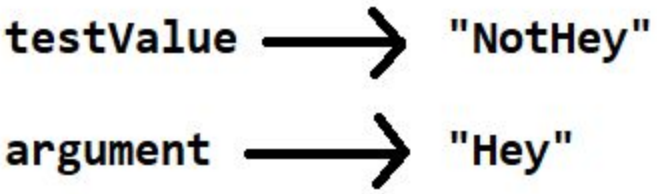
```
puts testPassType testValue
```

Then you will get "Hey".

This is because `argument` is a local variable inside the method. We are creating a new reference to the same Object that `testValue` refers to (which has the value of "NotHey") called `argument`.



However, when we say `argument = "Hey"` we are reassigning the value `argument` "holds" or references. So now `argument` and `testValue` refer to different Objects,



Since `argument` only exists within the scope of the method, if we do not 1) print it or 2) save it to another variable, it will go away once the method finishes. We can't call `argument` outside of the method, but we can say `puts testPassType testValue` *or* `newTestValue = testPassType testValue`. `testValue` never stops referring to the Object with the value of "NotHey".

Because `argument` and `testValue` are "holding" independent Objects or values, many call Ruby a pass-by-value language. However, there is also an argument to be made that Ruby is "pass-by-reference-value", because the variables we send to methods are not holding the values themselves but are instead *references* to Objects holding a value.

For more information on Ruby being pass-by-value or pass-by-reference, check out the following thread, RubyScope.

This, as well as examples of declaring and using methods, can see this in action on the accompanying file, methods.rb.

Sources

digitalocean.com
https://www.digitalocean.com/community/tutorials/how-to-work-with-strings-in-ruby Accessed October 8, 2020.
educative.com https://www.educative.io/edpresso/pass-by-value-vs-pass-by-reference Accessed October 27, 2020.
launchschool.com https://launchschool.com/blog/object-passing-in-ruby Accessed October 27, 2020.
mixandgo.com https://mixandgo.com/learn/is-ruby-pass-by-reference-or-pass-by-value Accessed October 8, 2020.
rubyguides.com https://www.rubyguides.com/2018/01/ruby-string-methods/ Accessed October 8, 2020.
rubyguides.com https://www.rubyguides.com/2015/08/ruby-recursion-and-memoization/ Accessed October 8, 2020.
ruby-for-beginners.rubymonstas.org
http://ruby-for-beginners.rubymonstas.org/writing_methods/return_values.html Accessed October 8, 2020.
ruby-lang.org https://docs.ruby-lang.org/en/2.0.0/Array.html Accessed October 8, 2020.

stackoverflow.com
https://stackoverflow.com/questions/14575581/can-i-tell-a-ruby-method-to-expect-a-specific-parameter-type Accessed October 8, 2020.
stackoverflow.com
https://stackoverflow.com/questions/64562581/why-can-you-change-the-value-of-a-local-variable-in-ruby-in-a-function-using-ano Accessed October 27, 2020.
tutorialspoint.com https://www.tutorialspoint.com/ruby/ruby_methods.htm Accessed October 8, 2020.