Getting Started

Create a git repository on GitHub, called "ruby_fundamentals2" or something similar. Clone it onto your own computer. This assignment will walk you through creating several Ruby programs which you should add to your git repository. Make sure to copy the question(paste it as comments) to the file you answer .

Methods

Built-in Methods

Ruby comes with a library of predefined methods. In yesterday's assignment we used the built-in methods `puts` and `gets`.

```
puts "What's your name?"
name = gets # gets is called here, just by writing its name
puts "Hello #{name}"
```

You will also sometimes see methods be called by using parentheses. This is sometimes necessary, but in Ruby, we only use them if we need to. For example, the following two lines of code are identical:

```
puts "Hello world"
puts("Hello world")
```

However, the first example is preferred. You will see examples using both styles throughout this assignment.

Some other built-in Ruby methods include `.class`, `.to_i`, and `.to_s`. Unlike `puts` and `gets`, these methods must be performed by specific objects. In programming lingo we say they must be called on specific objects. Let's see that in action:

```
3.to_s # "3"
```

In this example we "called" the `.to_s` ("to string") method "on" the number `3`, and in doing so instructed that number to convert itself into a string (`"3"`).

```
"5.0".to_i # 5
```

In this example we called the `.to_i` ("to integer") method on the string `"5.0"`, and in doing so instructed that string to convert itself into an integer (`5`).

```
"Programming".class # String
```

In this example we called the `.class` method on the string `"Programming"`, which instructed that string to tell us its data type (String).

```
12.class # Integer (or Fixnum if you're on a version of Ruby older than 2.4)
```

In this example we called the `.class` method on the number `12`.

Defining Methods

You can define your own methods by using the keyword `def` followed by the method name. The method body is enclosed by this definition on the top and the word `end` on the bottom. In Ruby the convention is to keep your method names all lower case and to use underscores (_) to separate words. Let's see an example:

```
def my_first_method
  return 1 + 1 # Code to be executed
end
```

To execute or call a method in irb, we simply write the following:

```
irb(main):001:0> my_first_method
=> 2
```

or in a Ruby file:

```
my_first_method # returns 2
```

Return values

Every method returns a value. You can specify a return value using the `return` keyword, otherwise the value of the last line that the method executes will be the value returned by the method. The return value is handed back ("returned") to the caller (ie. the place in your code

where you called the method) when the method has done its work. A method must always return exactly one value.

Using the `return` keyword is referred to as an "explicit return". An "implicit return" refers to when the last line of the method is taken as the return value.

```
def explicit_return_method
 "The interpreter reads over me, but does nothing"
 return 25
  "The interpreter does not read me, because the return keyword above forces the interpreter to exit the method"
end
```

The `explicit_return_method` method returns the value `25` because of the use of the `return` keyword.

```
def implicit_return_method
  1 + 1
  25
end
```

The `implicit_return_method` method returns the value `25` because that is the value of the last line of code in the method.

Note: If you have a `puts`, `print` or `p` statement as the last line in your method its return value will be `nil`, which is Ruby's way of representing a non-value.

Parameters

Method parameters (also called arguments) are specified between parentheses following the method name. A method can accept any number of parameters (or none). This is a way for the caller of a method to pass it the information it needs in order to do its job.

```
def reverse_sign(num)
  return -1 * num # Code to be executed
end
```

Here we defined a `reverse_sign` method that accepts one parameter called `num`, which needs to be a number.

```
reverse_sign(56) # -56
```

Here we called the `reverse_sign` method and passed it the number `56` as a parameter/argument.

Variable Scope

Scope defines where in a program a variable is accessible. Ruby has four types of variable scope: local, global, instance and class. At this point we are only concerned with local variables. We'll talk more about the other three types in the next few days.

To illustrate the concept of scope:

```
def plus_one(num_2)
 return num_2 + 1
end

plus_one(num_1) # undefined local variable or method 'num_1' for
main:Object
```

Notice that we're trying to pass the variable `num_1` as a parameter to the `plus_one` method. This produces an error because we never defined the variable. Let's look at the same example with "num_1" now defined:

```
num_1 = 20

def plus_one(num_2)
 return num_2 + 1
end

plus_one(num_1) # 21
plus_one(num_2) # undefined local variable or method 'num_2' for
main:Object
```

It should be clear that `num_1` is different than the variable `num_2`, which only exists as a placeholder within the definition of `plus_one`. When we call `plus_one(num_1)` it is the equivalent of calling `plus_one(20)`, because that is the value that `num_1` points to. When we call the `plus_one(num_1)` method the variable `num_2` gets temporarily assigned to the value that was passed in as a parameter (`20`), but after the method executes that one time, `num_2` no longer has a value. This means that trying to call `plus_one(num_2)` still causes an

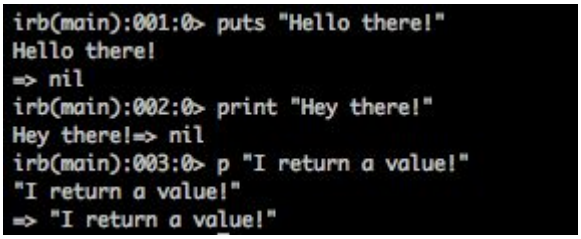
error because `num_2` exists *only within the scope of* `plus_one`. We can't use it outside of that method.

The best way to understand variable scope is to play around with it as much as you can. Never be afraid to make a new Ruby program to test your knowledge about something 'obvious'. Programmers do this all the time!

`puts` / `print` / `p`

As a beginner, it can be quite confusing as to where to put `puts` and `print` statements. You should avoid using them on the last line of a method because that will make the method return `nil` (Ruby's non-value).

See for yourself with irb:



The reason for this is `puts` and `print` are both methods that themselves have the return type `nil`

Instead you can use the `p` method, which acts like `puts` but returns back the string you give it instead of `nil`.

Exercise 1

Define a method called `double` that accepts an argument called `my_number` and returns that number multiplied by 2.

Try calling it multiple times and pass in different numbers each time.

Exercise 2

Define a method called `negative?` that accepts a number as an argument and returns a boolean (true/false) indicating whether that number is negative or not.

Try calling it multiple times, passing in different numbers each time.

Exercise 3

Define a method called `is_even?` that accepts a number as an argument and returns a boolean (true/false) indicating whether that number is even or not (HINT: use the `%` operator).

Try calling it with different numbers.

Exercise 4

Define a method that accepts a string as an argument and returns false if the word is less than 8 characters long (or true otherwise).

Exercise 5

In the far future, everyone spells their names backwards. Create a method called `greet_backwards` that greets people using their reversed names. For example:

Calling `puts(greet_backwards('Amanda'))` should output:

Hello, adnamA! Welcome home.

Call that method for four different people: "Bob", "Shirly", "Sue", and "Andy".

Notice how much extra code we would have needed if didn't make use of a method!

Finally, modify your `greet_backwards` method to say the person's name twice. For our 'Amanda' example, you should get:

Hello, adnamAadnamA! Welcome home.

Notice that we would have had to modify four lines of code if we didn't have the `greet_backwards` method! This is just one reason methods are useful.

Exercise 6

Create a method that converts Fahrenheit temperatures to Celsius in a file called exercise6.rb.

Start with prompting the user for a temperature in Fahrenheit. Then call your method and pass in the user input as a parameter.

Your method should:

- have one parameter: the temperature in Fahrenheit
- do the conversion with this formula: C = (F - 32) x 5/9
- ensure that the parameter you pass in is a number by converting it with `to_i`

Output the result in a full sentence using string interpolation.

Don't forget to commit your progress as you go along. Once you're done, commit one last time and push it to github.

TIP: Don't start your variable names with capital letters, otherwise Ruby will think you want to make a constant instead of a variable. Making a constant is a way of giving a name to a value that won't ever be reassigned.

Exercise 7

Let's create a method `wrap_text` that wraps text in symbols of our choice. For example:

```
wrap_text('hello', '===')
```

should return:

```
===hello===
```

Now that this method works, how can we use it (without modifying the method) to generate the following string?

```
---===###new message###======-
```

Note that `wrap_text` needs to *return* a value rather than print one.

Hints:

- You'll have to call the same method multiple times.
- Try breaking down the problem into smaller pieces that you know `wrap_text` can solve.

Exercise 8

Read the following Ruby code that does not follow the principle of "don't repeat yourself". Rewrite it to use methods instead of repeating code. Consider what your arguments and return values should be.

```
puts "How far did person 1 run (in metres)?"
distance1 = gets.to_f
puts "How long (in minutes) did person 1 run take to run #{distance1}
metres?"
mins1 = gets.to_f


puts "How far did person 2 run (in metres)?"
distance2 = gets.to_f
puts "How long (in minutes) did person 2 take to run #{distance2}
metres?"
mins2 = gets.to_f


puts "How far did person 3 run (in metres)?"
distance3 = gets.to_f
puts "How long (in minutes) did person 3 take to run #{distance3}
metres?"
mins3 = gets.to_f

secs1 = mins1 * 60
speed1 = distance1/secs1
secs2 = mins2 * 60
speed2 = distance2/secs2
```

```
secs3 = mins3 * 60
speed3 = distance3/secs3

if speed3 > speed2 && speed3 > speed1
  puts "Person 3 was the fastest at #{speed3} m/s"
elsif speed2 > speed3 && speed2 > speed1
  puts "Person 2 was the fastest at #{speed2} m/s"
elsif speed1 > speed3 && speed1 > speed2
  puts "Person 1 was the fastest at #{speed1} m/s"
elsif speed1 == speed2 && speed2 == speed3
  puts "Everyone tied at #{speed1} m/s"
else
  puts "Well done everyone!"
end
```