Open in new tab

Ruby Functions

Lesson Duration: 60 minutes

Learning Objectives

- Know what a function is
- Understand passing arguments into functions
- Be able to write and test your own functions

What Is a Function?

A 'function' is a reusable chunk of code that can be called (invoked) by name to perform a specific task. We can think of it as a little machine that takes in some information and returns something.

Let's create a little code file to play with.

```
# terminal
touch my_functions.rb

# my_functions.rb
def greet()
   return "Hey"
end

p greet()
```

What Do Functions Do?

We use them to encapsulate meaningful pieces of code into small, isolated constructs.

We've been using some functions already like `.length`. These are special kinds of functions which are associated with an object ("methods"). Try not to worry about this too much - we will cover it later in the week.

In Ruby, functions *always* return a value - whether we want them to or not. By default, the result of the last evaluated expression is returned, but you can also use the `return` keyword to explicitly return a value. We will do this in the beginning because it makes it clearer.

It is also important to remember that although Ruby functions always return a value, we might not always need to know what this is or need to use it. Sometimes we just want a function to complete the action we've told it to and then finish.

Why Do We Use Functions?

- It's much easier to find and fix bugs if you've organised your program well.
- Separation of concerns makes our code less redundant
- We can 'abstract' our programs into individual parts.
- We can 'encapsulate' data so that no other part of the program can interfere with it

Anatomy of a Function

Defined with `def` then function name followed by (), end with `end`

- name can't begin with a number
- name *must* begin with a lower-case character
- special characters are allowed and imply functionality:
    - ? for predicate; i.e. the function will return `true` or `false` and nothing else, e.g. `3.even?` will return `false`.
    - ! or 'bang'; used when a function will change the object is it called on. Don't worry about this, we will probably never have to use it!
    - = for setters, which we will learn about next week.

Scope

A function has it's own internal world and doesn't share it's variables that another function has. We call this little world the *scope* of the function.

```
def greet()
  words = "Hey"
  return words
end

def greet_two()
  return words
end

p greet_two()
```

This will throw an error as the `greet_two()` function doesn't know about words.

We call variables which are solely defined in a function "local variables". They exist for the life of the function, then are lost. They cannot be seen by any other function.

Passing Variables to Functions

We can give functions the information they need using arguments which we pass into the function. The behaviour of our function will now depend on what we pass in. This is a very powerful concept.

```
def greet(name)
  return "Hey " + name
end

p greet('Bob')
# => "Hey Bob"

def myFunction( parameter )
  return parameter
end

p myFunction( 'argument' )
# => 'argument'
p myFunction( 12345 )
# => 12345
```

When we define what we need in our functions world, we call these *parameters*. When we provide what a function has asked for, we call these *arguments*.

Multiple parameters are separated with a comma:

```
def greet(name, time_of_day)
  return "Good " + time_of_day + ", " + name
end

p greet('Bob', 'morning')
# => "Good morning, Bob"
```

This way of 'adding' strings together (technically called string *concatenation*) is getting a little messy, but there is a way to return a pretty string and write prettier code at the same time!.

String *interpolation* allows us to use tags `#{}` inside a string. Anything in these tags will be evaluated as code, i.e it will look for the variable. So we can write our full return as one string, and use interpolation where we want to use a variable:

```
def greet(name, time_of_day)
  return "Good #{time_of_day}, #{name}"
end

p greet('Bob', 'morning')
# => "Good morning, Bob"
```

> [Task:] Improve the greet method to always have a capital at the start of their name.

We can improve the greet method by making sure the `name` parameter is always capitalised in the return string:

```
def greet(name, time_of_day)
  return "Good #{time_of_day}, #{name.capitalize()}"
end

p greet('bob', 'morning')
# => "Good morning, Bob"
```

Ruby objects, such as String and Integer come with many excellent useful methods.

> [:MINI LAB]
>
> 1. Create an add function
>
> - this will take two parameters
> - call the first parameter first_number and the second second_number
> - use the return keyword
> - Invoke the function `add( 2, 3 )`
>
> 1. Create a population_density function
>
> - this will take two parameters
> - call the first parameter population and the second area
> - calculate the population density (e.g. `population / area`) and return it
> - invoke the function using the population and area of Mauritius:
>   - population: 5373000
>   - area: 77933

Solution:

```
def add(first_number, second_number)
  return first_number + second_number
end

p add(2, 3)

def population_density(population, area)
  return population / area
end

p population_density(5373000, 77933)
```

Published with GitHub Pages