Default Arguments

Why Optional Arguments?

We should constantly strive for our code to be dynamic and flexible. As programmers, we are lazy (which is a virtue). Consequently, we want the code we write to be re-usable.

If we define a method, `#greeting`, like this:

```ruby
def greeting
  puts "Hello, Ruby programmer!"
end
```

We have to re-define or re-write that method every time we'd like to use it to greet someone else who might not be a Ruby programmer. Since that's way too much work for us, we'll define our method to take in an argument of someone's name:

```ruby
def greeting(name)
  puts "Hello, #{name}"
end
```

Now our method is flexible and dynamic. It can be used again and again to greet different people.

But what if we don't know the name of the person we are trying to greet? We can make this method even more flexible by making the `name` argument optional. We do this by using optional, or default, arguments.

Default Arguments

In order to define a method that optionally takes in an argument, we define our method to take in an argument with a **default value**. By defining our method with default arguments, we make it possible to call the method with optional arguments, i.e. with or without arguments.

```ruby
#                   assigning a default value
def greeting(name = "Ruby programmer")
  puts "Hello, #{name}"
end
```

In our argument list, `(name = "Ruby programmer")`, we simply assign the argument `name` a default value of `"Ruby programmer"`. By doing so, we are really saying:

If the method is invoked without any arguments, i.e. like this: `greeting`, Ruby will assume the value of the `name` variable inside the method to be `"Ruby programmer"`.

However, if the method is invoked with an argument, `greeting("Sophie")`, Ruby will assign the variable `name` to the string `"Sophie"` inside the method.

```ruby
greeting
# > Hello, Ruby programmer!

greeting("Sophie")
# > Hello, Sophie!
```

With default arguments, our once simple machine becomes profoundly useful and abstract.

Adding Default Arguments

Default arguments are easy to add, you simply assign them a default value with `=` ("equals") in the argument list. There's no limit to the number of arguments that you can make default.

```ruby
def greeting(name="Ruby programmer", language="Ruby")
  puts "Hello, #{name}. We heard you are a great #{language} programmer."
end
```

Let's take a look at the different ways we can call this method:

```ruby
greeting
# > Hello, Ruby programmer. We heard you are a great Ruby programmer.

greeting("Sophie")
# > Hello, Sophie. We heard you are a great Ruby programmer.

greeting("Steven", "Elixir")
# > Hello, Steven. We heard you are a great Elixir programmer.
```

Using Default Argument and Required Arguments

It is possible to define a method that takes in both required and default arguments. To do so, however, we must place the default argument at the end of the argument list in the method definition.

Take a look:

```ruby
def greeting(name, language="Ruby")
  puts "Hello, #{name}. We heard you are a great #{language} programmer."
end
```

Let's call our `#greeting` method with and without an explicit `language` argument:

```ruby
greeting("Sophie", "Ember.js")
# > Hello, Sophie. We heard you are a great Ember.js programmer.

greeting("Dan")
# > Hello, Dan. We heard you are a great Ruby programmer.
```

It works! Why must we place the default argument at the end of the argument list?

Let's take a look at what would happen if we didn't:

```ruby
def greeting(language="Ruby", name)
  puts "Hello, #{name}. We heard you are a great #{language} programmer."
end
```

Now, what happens when we try to call our method without an explicit `language` argument?

```ruby
greeting("Sophie")
```

You might expect it to break. Or you might expect it to think that the `language` variable is being set equal to `"Sophie"` in this method call.

Neither of those things will happen. The method will work as we intended because Ruby is smart and has a few tricks up its sleeve to help determine what method arguments are being used where in a method's body.

However, defining the default argument first is confusing. We can understand this from our very reasonable expectations that the above method invocation would break. For this reason, it is conventional to place any default arguments at the end of an argument list when defining a method that takes in both required and default arguments.

Conclusion

Method arguments, both required and optional, make methods powerfully abstract and dynamic machines that are easy to build yet very flexible and adaptable to different situations and requirements. Get used to defining methods with required and default arguments and calling them correctly.