Your First Ruby Script

Step-By-Step

Martin Miliauskas · @mmiliauskas
# Table of Contents

- Introduction 4
- Warming Up 5
- Step 1. Basic Spider 14
- Step 2. Getting Smarter 17
- Step 3. Adding Guard 19
- Step 4. Scraping Abstract 22
- Step 5. More Scraping 24
- Step 6. Refactoring with OOP 26
- Step 7. Accessing Attributes 30
- Step 8. Verbose CLI 33
- Step 9. Fixing Bad Practice 36
- Step 10. Optimizing Keys 39
- Step 11. Adding Filters for Links 41
- Step 12. Custom Filters 45
- Step 13. Namespaces 48
- Step 14. Decoupling 50
- Finish 54
Introduction

This book’s intention is to teach you Ruby programming language by writing a real world script, step-by-step. Every step will introduce a new Ruby concept written in a concise, all you need to know manner. By the end of this book you will be able to read and write intermediate level Ruby scripts.

What we will build?
During the course of this book, we will build a spider script capable of scraping Wikipedia.org pages.

What Ruby concepts will we cover?
• Datatypes and Objects
• Methods
• Classes and Modules
• Code Blocks
• Dealing with External Libraries (aka gems)

What you need to have?
• Knowledge of at least one programming language;
• Ruby installed. The following guide should suffice.
Warming Up

A very useful tool which comes bundled with Ruby is **IRB**, an interactive command-line interpreter which lets you execute and see the results of your Ruby code immediately.

In this chapter we will test Ruby concepts and play with its syntax using exactly this tool. So without further ado, let’s start IRB by executing *irb* command in the terminal:

```
projects$ irb --simple-prompt
```

```
Everything is an object

In Ruby everything is an object, even literals:

```ruby
>> "alice".capitalize
=> "Alice"
>> 5.next
=> 6
>> false.class
=> FalseClass
>> [5, 12, 4].sort
=> [4, 5, 12]
```

Math

Basics

```ruby
>> 3 + 2
=> 5
>> 5 * 2
=> 10
```

Power

```ruby
>> 2 ** 8
=> 256
```

Division

If all numbers in the expression are integers (objects of Fixnum), the result is an integer as well:

```ruby
>> 8 / 3
=> 2
```

You can think of the above expression as 5 div 6 (if you are familiar with Turbo Pascal).

In case you need more precise result, simply turn one of the numbers into a floating point number:

```ruby
>> 8 / 3.0
=> 2.66666666666667
>> 8 / 3.to_f
=> 2.66666666666667
```
Data structures

Two main types of collections that are used in Ruby are: **arrays** and **hashes** (associative arrays). Arrays and hashes are **mutable**.

A common way to define an array is using ` [...]` literal, and `{ ... }` to define a hash:

```ruby
>>> [ 4, 5, [21, 'hello'] ]
=> [ 4, 5, [21, "hello"] ]
>>> {'a'=>"Alice", 2 => 'John' }
=> {"a"=>"Alice", 2=>"John"}
```

We will play more with these data structures in further chapters.

Methods

In order to have a re-usable code, we need to store it inside the **method**. Method definition starts with `def` and ends with `end`:

```ruby
>>> def add a, b
>>> a + b
>>> end
=> nil
>>> add 1, 5
=> 6
```

As you can see, brackets around parameters in a method definition and method call are **optional**. `return` statement is also **optional**, and by default the result of the last expression is returned.

Chaining methods

Another cool thing that Ruby allows is **method chaining**. You can chain methods as long as the object returned by the previous method call has the subsequent method, i.e.:

```ruby
>>> add(5,6).next.next
=> 13
```

`add(5,6)` returns an object of `Fixnum`, which has `next` method, which in turn also returns an object of type `Fixnum`.
Code blocks

Code blocks are like anonymous functions, which can be attached to a method call. For example, you can attach a code block to `Array#each` method, which then passes every element from the array to it:

```ruby
>> [2, 15, 42].each { |x| puts x}
2
15
42
=> [2, 15, 42]
```

Code between {...} is used for defining a code block, the parameters of which are defined inside of the |...|. Code blocks which span over multiple lines can also be defined using `def ... end`:

```ruby
def ... end:
>> [2, 15, 42].each do |x|
?> x = x * 2
>> puts x
>> end
4
30
84
=> [2, 15, 42]
```

Iterating

A common use of code blocks is to iterate over arrays and hashes:

```ruby
>> [2, 15, 42].each_with_index do |x, index|
?> puts "#{index}=>#{x}"
>> end
0=>2
1=>15
2=>42
=> [2, 15, 42]
```

```ruby
>> u = { 'a' => 'Alice', 'j' => 'John' }
=> { "a" => "Alice", "j" => "John" }
>> u.each_pair do |key, value|
?> puts "#{key}=>#{value}"
>> end
a=>{value}
j=>{value}
=> { "a" => "Alice", "j" => "John" }
```
Simple *counter loops* can be achieved using methods provided by integers:

```ruby
>>> 3.times { |counter| puts counter }
0
1
2
=> 3
>>> 1.upto(3) { |counter| puts counter }
1
2
3
=> 1
>>> 0.downto(-3) { |counter| puts counter }
0
-1
-2
-3
=> 0
```

**Flow control**

In order to break or skip iteration, use `break` and `next` respectively, i.e.:

```ruby
>>> [2, 15, 42, 68, 111].each do |x|
  ?> if x == 15
  >>  next
  ?> if x > 50
  >>  break
  ?> end
  >>  puts x
  ?> end
2
42
=> nil
```

**Variables**

Global variables are available anywhere in your script and are prefixed with a dollar ($) sign:

```ruby
>>> $foo = "bar"
=> "bar"
```
Local
Local variables start with a lowercase letter or an underscore. Default notation is to use snake_case for your variable names:

```ruby
>> foo_bar = 10
=> 10
```

Local variables are only available in their scope: if declared in a method, or code block (since Ruby 1.9) - they won't be available outside.

```ruby
>> 1.times do |x|
  y = x + 10
  puts y
  puts x
>> end
10
0
=> 1
>> puts y
NameError: undefined local variable or method y' for main:Object ...
>> puts x
NameError: undefined local variable or method x' for main:Object ...
```

However, variables defined outside the code block are available inside it:

```ruby
>> foo = "bar"
=> "bar"
>> 1.times do
  puts foo
>> end
bar
```

Constants
Variables where the first letter is uppercase are treated as constants. By convention, constants have all letters in the uppercase and use underscore to separate words:

```ruby
>> FOO_BAR = 10
=> 10
```

The interesting thing about constants is that they are not final; you can assign a new value to same constant. Ruby will give you a warning but will change the value anyway:
Ruby treats its developers as an adults, thus it does not impose restrictions that can be found in other languages (i.e. Java or Python). Constants case is a good example of this; you are allowed to change their value, but Ruby trusts you to do it only when it is really necessary.

**Strings**

Strings usually initialized by wrapping them with double (""") or single (‘’) quotes:

```ruby
>> greet = "Hello"
=> "Hello"
>> to = 'World'
=> "World"
```

Using double quotes allows it to perform **string interpolation** - placing Ruby code into a string:

```ruby
>> "1 + 2 = #{1+2}"
=> "1 + 2 = 3"
```

It is common technique to use interpolation to concatenate strings:

```ruby
>> "#{greet}, #{to}!"
=> "Hello, World!"
```

**Outputting**

Two common ways to output something from the Ruby script, is by using `puts` or `p`.

**puts**

`puts` calls `obj.to_s` and outputs return value to output stream. `to_s` method returns string representation of an object `value`:

```ruby
>> puts ['hello', 'world']
hello
world
```
p
p is same as puts obj.inspect. inspect returns human readable version of an object, thus it is useful for debugging:

>> p ['hello', 'world']
['hello', 'world']

Classes
The final exercise in our warm up is defining a Class:

>> class Greeter
>> def initialize
>>   @greet = ''
>> end
>>
?>> def say_hello
?>>   @greet = 'Hello'
?>>   self
?>> end
>>
?>> def to name
?>>   puts "#{@greet}, #{name}!
?>> end
>> end
=> nil
>> g = Greeter.new
=> #<Greeter:0x007f892a101390 @greet="">
>> g.say_hello.to "Alice"
Hello, Alice!
=> nil

1. Class definition starts with class keyword and ends with end;
2. Greeter – the name of the class must always start with a capital letter;
3. initialize - a constructor method;
4. By prefixing a variable with `@` you turn it into an **instance variable**, which is available anywhere in the class scope;

5. `self` - same as `$this` in PHP or `this` in Java - represents instance of the current class;

6. `Greeter.new` - creates a new instance of `Greeter` class;

7. `g.say_hello.to` - `Greeter.say_hello` can be chained with `Greeter.to` method, because `Greeter.say_hello` returns `self`.

**Conclusion**

In this short chapter we had a crash course on Ruby syntax. We played with: different data types; defined variables, methods, classes; and iterated with code blocks - good start! But it was just the beginning of the fun. In the next chapters we will build an actual **spider script**, step-by-step. With every change of our code we will learn new Ruby concepts.
Step 1. Basic Spider

We are going to begin by writing a short script which would output source code of the *V for Vendetta* movie Wikipedia page:

**Full code**

```ruby
require 'open-uri'

url = 'http://en.wikipedia.org/wiki/V_for_vendetta'
source = open(url).read
puts source
```

Place above code into `spider.rb` file and execute it with `ruby` command, as shown below:

![Command output](image)

Things to note about the code that we wrote:

1. `require 'open-uri'` - loads standard Ruby library called `OpenURI`, which helps deal with http, https and ftp;
2. `open(url).read` - `open(url)` gets content from a remote file, which address is `url`, stores it inside the temporary file and returns `File` object representing it. `read` returns the contents of the `File` object as a string.
External libraries

Libraries in the Ruby world are called gems and are managed by Rubygems package manager.

Installing a gem

$ gem install rspec

Listing all installed gems

$ gem list

Using gem in your script

In order to use gems in your script, you need to require them first, i.e.:

require 'rspec'

Where are my gems stored?

The directory where Ruby gems are installed is in a list of Ruby’s search paths, thus you don’t need to specify an absolute path to your installed gems.

If you are eager to know where your gems are stored, execute gem env command, and look for GEM PATHS:

$ gem env

...  
- RUBYGEMS PLATFORMS:
  - ruby
  - x86_64-darwin-11
- GEM PATHS:
  - /Users/mmiliauskass/.rvm/gems/ruby-1.9.2-p320
  - /Users/mmiliauskass/.rvm/gems/ruby-1.9.2-p320@global
Did you like first two chapters?
Download full version here.