Open in new tab

Ruby Testing

Lesson Duration: 30 minutes

Learning Objectives

- Know what a test is
- Understand why we test our code
- Understand Arrange, Act, Assert
- Be able to use MiniTest to test Ruby functions

What is a Test?

A test is exactly what it sounds like - something that verifies your code is working. Let's say you write a function:

```
def five()
  return 5
end
```

How do we know that this function is working properly? Sure, we can look at it, and see that it's working, but that only works while the logic is simple. Another thing we can do is actually run the code. Like this:

```
p(five()) # 5
```

Now, whenever we want, we can run the code, and check that the correct thing is printed out. This is great.

But could we do better? This is software development, after all. Wouldn't it be ideal if we could write code that ran our code and checked it worked properly? That would be mind-blowingly cool. And that's exactly what we mean when we talk about testing our code.

By the end of this lesson we'll be able to write code that automatically verifies that our functions are working correctly.

Why Should We Test?

When we write our code we want to make sure it works as we expected. Maybe this sounds obvious. So let's make an ever bigger claim. We can craft better code if it's tested.

We haven't really touched on the craft of writing code yet. When we talk about code craft we mean things like the ability to easily make changes to code without worrying about introducing bugs, as well as the ability for other people to work on our code easily.

Having tests we can run ensures that if we break code by changing it we know instantly. Therefore the quality of the code is instantly improved, even if the code itself is unchanged. It also means that someone new to our code can look at the tests to very quickly work out how the code works.

There are many approaches to testing our code but the principles are the same. What we want to do is write code which expects some specific thing to happen. These "expectations" are referred to as assertions. As our tests run, each assertion will either pass or fail, and the result will be printed to the terminal.

Most programming languages have test suites and different ways of testing. The testing library we will use in Ruby is called MiniTest.

How Do We Test?

> Hand out start point

To setup our tests we create a folder to hold our tests and a separate file:

```
# terminal
mkdir specs
touch specs/my_functions_spec.rb
```

This `my_functions_spec.rb` is going to be run by us manually to ensure our code works as expected!

We can install additional classes not available by default using the `gem` command line tool. It can go to the internet and grab other Ruby classes written by other people. This is incredibly powerful as it means we don't have to, e.g., write our own testing framework. Let's grab MiniTest now.

```
gem install minitest
```

We can also install `minitest-reporters`, this will swap out the `Minitest` runner to the custom one used by `minitest-reporters` and use the correct reporters for the console. If you would like to write your own reporter, just include `Minitest::Reporters` and override the methods you'd like.

```
gem install minitest-reporters
```

Now we need to require the gems we installed. For *Reporters*, we are going to use a report (this just adds some extra behavoiur/functionality to `Reporter` gem) called `SpecReporter`, which output will show full detailed feedback for the test file. Similarly to `Resporter` gem we need to invoke it first in order to be able to use it. We will also need the file we are going to test.

```
# my_functions_spec.rb

require( 'minitest/autorun' )
require('minitest/reporters')
Minitest::Reporters.use! Minitest::Reporters::SpecReporter.new

require_relative( '../my_functions' )
```

Then we do some test setup.

```
# my_functions_spec.rb

class FunctionsTest < MiniTest::Test

end
```

This makes a class called `FunctionsTest` that is based on (includes all the functionality of) the `Test` class in the MiniTest library we just installed. We'll find out more about classes in the coming weeks.

Next we write our actual test. We write this in a very similar way to writing a function with `def`. In MiniTest, all test names MUST begin with the word `test` for them to be recognised as tests and run.

```
# my_functions_spec.rb

class FunctionsTest < MiniTest::Test

  def test_greet

  end

end
```

Arrange, Act, Assert

A very common way to structure a test is using "Arrange, Act, Assert", where every test you write is composed from the following 3 parts:

- **Arrange** - do any set up necessary for the test (for really simple tests, there won't be any)
- **Act** - call the *function under test* (one call, never more)
- **Assert** - check that the right thing happened, using at least one "assert" method (this might be checking the return value of a function or checking it has modified something else)

MiniTest gives us a function, `assert_equal` which takes two arguments: the value you expected to see, and the value you actually got.

We can write a test for the `greet` function we wrote earlier:

```
# my_functions_spec.rb

def test_greet
  # arrange
  # nothing to do here...

  # act
  # we're testing the greet method, so we'd better call that
  # store the result so we can check it was right in the next step
  result = greet('David', 'morning')

  # assert
  # given the parameters we passed in, we should expect the function
  # to return 'Good morning, David'
  assert_equal('Good morning, David', result)
end
```

Note that we didn't have any arranging to do - when we have more complex code using classes later, there will more likely be stuff in this section.

Also, as can be seen in our assert, we are testing the return value of our greet method. We know that this method will return something, so we want to test this output

We can run our test file in the terminal:

```
# terminal
ruby specs/my_functions_spec.rb
```

We can see there has been one test run, with one assertion, and no failures.

> Show a few more simple tests to drill in the format.
>
> [Task:] Add a test for the add function that we made.

```
# my_functions_spec.rb

def test_add
  result = add(2, 3)
  assert_equal(5, result)
end
```

Good/bad Method Names

It's always useful to use function names that say what the function does - use snake case, and if we are testing the same function multiple times we can use two dashes then specify the context for that test.

> [Task:] Add a test for the add function that tests if adding a negative number works.

```
# my_functions_spec.rb

def test_add__negative
  result = add(4, -2)
  assert_equal(2, result)
end
```

Published with GitHub Pages