3 Ruby: Reading RSpec Tests

Ensuring Your Code Behaves Properly

> This checkpoint shows how to **read** RSpec tests. We'll learn much more about RSpec and how to **write** tests later.

A method can return an object without error but still not do what it was meant to. You have to **test** your code to ensure proper behavior. For example, implement an incorrect `add` method by typing:

```
def add(a,b)
  a-b
end

p add(5,4)
```

> Since you're not currently on an exercise page, enter the code examples using **repl.it**.

The `add` method will return a valid number, but not the expected result. Programmers need to test their code to make sure that it does what it was meant to do.

You could run a simple test by using a comparison operator. If the comparison returns `true` you'd know that the method accomplished its mission.

```ruby
def hello
  "hello world"
end

p hello == "hello world"
```

This is a very specific test though, and not scalable for code enhancements. There are much more flexible and dynamic ways to test code using test frameworks. One of the most popular frameworks in the Ruby on Rails community, and Bloc's choice, is **RSpec**.

RSpec in a Nutshell

RSpec, like most testing frameworks, runs your code in particular conditions and with particular arguments. It sets *expectations* for the outcome of this process, and the test passes if those expectations are met.

> This basic concept is important to understand. A spec is just running your code and testing what happens.

Let's explore a method test and how it might lead us to write a method. For this example, consider `greet`, which will take two arguments, a person's first and last name, and then return a personalized "hello" directed to (the individual with) that name. As a concrete example, if we call `greet` with the arguments `("Annie", "Oakley")`, it will return `"Hello Annie Oakley."`.

Let's say we've already written this method. We could "test" our method by using pure Ruby to assert that the return of our method equals what we want it to:

```ruby
greeting = greet("Annie", "Oakley")
greeting == "Hello Annie Oakley."
#=> If our method behaves as specified, this would return true. If not, it would ret
```

In RSpec, the core of the test looks very similar:

```ruby
    greeting = greet("Annie", "Oakley")
    expect(greeting).to eq("Hello Annie Oakley.")
```

In RSpec, rather than returning `true` or `false`, this test will "pass" or "fail" depending on

whether the expectation for the method is met. RSpec syntax, however, is a *little* bit more complicated than the above. Consider the following example:

```
describe "greet" do
  it "says hello to someone" do
    greeting = greet("Annie", "Oakley")
    expect(greeting).to eq("Hello Annie Oakley.")
  end
end
```

While the "meat" of the test remains unchanged, we've now wrapped these lines in `describe` and `it` sections.

> The strings following `describe` and `it` are for human consumption. You can put whatever you want there — though it should actually describe the method and its behavior. Changing the contents of those strings will *not* change the test behavior.

Because the strings immediately following `describe` and `it` are for readers, not the computer, we've used these strings to explain what we're testing. Following a typical RSpec pattern, we've made the text to the right of `describe` (`"greet"`) a description of the method we will be testing. The string `"says hello to someone"` following `it` is similarly explanatory, and gives a basic human-readable description of the behavior we intend to test.

Inside the `describe` and `it`, we write the actual test. It runs our `greet` method with two arguments and sets an expectation about what that method will return. Specifically, we use RSpec's `expect`, `to`, and `eq` methods to assert that we expect the return value of calling `greet` with the arguments "Annie" and "Oakley" to equal "Hello Annie Oakley." If that expectation is met, the test will pass. If not, it will fail.

Let's walk through this test again. You don't need to know just yet *how* it's working, but you should be able to understand *what* it's doing:

- We declare what we're testing (the `greet` method), in between `describe` and `do`.
- We declare our expected behavior in English (for humans), in between the `it` and the `do`.
- We write the actual *test*, assigning the result of calling `greet("Annie", "Oakley")` to a `greeting` variable, then asserting that we expect the greeting variable to equal "Hello Annie Oakley." If that expectation is fulfilled, the test will pass; otherwise, it will fail.
- We "close" the `describe` and `it` "blocks" with the keyword `end`.

> We'll explain blocks (and the `do` and `end` syntax) later. For now, just note that each `do` here is "closed" with an `end`.

To make this test pass, you would implement the `greet` method as it's written below:

```
def greet(first, last)
  "Hello #{first} #{last}."
end
```

> Why didn't we use `p` here? The `p` method would print the value of the string; we just want to return it. RSpec assesses the return value, stored in the `greeting` variable, rather than watching the "side effect" of printing.

Test-Driven Development

When tests are written to specify future code's behavior, it's called "Test-Driven Development", or TDD. Let's try some TDD one more time before launching into the exercises. Read this test:

```
describe "multiply" do
  it "multiplies its two arguments" do
    product = multiply(5, 3)
    expect(product).to eq(15)
  end
end
```

> How would you write this `multiply` method to pass the above test? How many arguments would it take? What would it do with those arguments?

This test sets expectations about the `multiply` method. Specifically, it asserts that calling `multiply` with the arguments `3` and `5` will return the value `15`. Let's take the easiest path towards fulfilling this spec by writing the below method:

```
def multiply(x, y)
  15
end
```

> Why are we again returning the value, rather than using `p` to print it?

But something is wrong. We've ignored the arguments passed in, instead "hardcoding" the value to be returned. This narrow implementation passes the above spec, but will break the instant we test it with different values. If our test gives the `multiply` method different numbers, as it does below, the above implementation will fail:

```
describe "multiply" do
  it "multiplies its two arguments" do
    product = multiply(4, 3)
    expect(product).to eq(12)
  end
end
```

> How do we make our implementation more flexible?

As you can see above, narrow implementations of methods can pass single specific tests. Because of this, the specs in the coming exercises will test your code from a variety of angles. Read the specs after the exercise description, to make sure you understand exactly what is expected of your code, and what is needed to pass. These specs will **run your code** with a variety of arguments, making sure it behaves as asserted.

Exercises

~~A Tested Method~~

Solution Walk-Throughs

Do not watch these videos until after you've attempted your own solutions. If you struggle to complete this exercise, submit your best effort to your mentor *before watching a solution walk-through*. Submit your answer even if some tests don't pass, as it's important for your mentor to understand your thought process. Discuss difficult exercises with your mentor.

There are many possible solutions to the exercises. A walk-through is meant to provide insight into the thought process of *one possible solution* per exercise. If your solution is different yet passes all the tests, do not change your solution to match what is seen in the video. Instead, discuss your thought process with your mentor.

A Tested Method Solution

assignment completed