**Introduction to RSpec Testing**

**Why do we use automated testing?**

Currently, we manually test our code. For example, we create our own test cases by printing and checking the output of our functions. This can be tedious, repetitive, and **worst of all, it is a method vulnerable to both false positives and false negatives.** The larger your code base is, the more chances there are for both of these to occur. Because of this, relying on manual testing alone is not a sustainable solution when you start working on larger code projects written by multiple people.

Enter automated testing.

**What is automated testing?**

When using automated testing, developers will code *test suites*, a collection of test cases that are intended to show that a program demonstrates some specified set of behaviours. There are many libraries dedicated to doing this, with the most popular one for Ruby being **RSpec**.

Though we spend more time upfront writing and updating code for our test suite, the advantage is that we can instantly and easily test that code at any time from then on, in a way that will be simpler and more robust than doing it manually.

**Foundations and testing**

Testing is a huge topic that we'll more fully introduce during the main course, but as we saw in the last section, the core concept is simple: the tests sets up some expectation of behavior and then checks that your code meets that expectation. **For now, our goal is to be able to interpret tests and understand the basics of RSpec, not necessarily write tests in RSpec.** The upcoming projects in this course utilize automated testing via **RSpec**, so the ability to read tests will be invaluable!

**Anatomy of an RSpec project**

To use RSpec, we'll need to structure our project files in a certain way. We separate our implementation code files from the testing files using a /lib and /spec folder respectively. Another word for a "test" is a "spec" (short for specification, since the tests specify how our code should behave). Let's say we had two methods that we wanted to have tests for, add and prime?, then we can structure our project like so:

/example\_project

├── lib

│   ├── add.rb

│   └── prime.rb

└── spec

├── add\_spec.rb

└── prime\_spec.rb

To use RSpec, we **must** follow this structure. We need folders with the literal names lib and spec as direct children of the example\_project folder. The test files inside of the /spec folder must end with \_specin their names.

**How to Read Specs**

You may initially find reading spec files as intimidating because you are interpreting another programmer's code and you don't understand exactly how it works. That's okay! The beauty of RSpec is that you don't need to know every detail of how it works, just look for the big picture idea. RSpec reads like english. To reiterate, our goal right now is to *read* RSpec, not necessarily *write* it.

Let's take a look at the contents of /spec/add\_spec.rb to see how we test the add method. The behavior outlined in the specs will dictate how we ought to design the method in /lib/add.rb.

# /spec/add\_spec.rb

require "add" # this line will include code from "/lib/add.rb"

describe "add method" do

it "should accept two numbers as arguments" do

expect { add(2, 3) }.to\_not raise\_error

end

it "should return the sum of the two numbers" do

expect(add(2, 3)).to eq(5)

expect(add(10, 12)).to eq(22)

end

end

Reading this code, you should get the feel of how the add method will be tested. Here's the semantic interpretation of the code:

* The description of the add method outlines 2 criteria:
  + it should accept two numbers as arguments
  + it should return the sum of the two numbers

By simply looking at the describe and it lines, we get a clear idea of how add should behave. However, if you need more clarity on the meaning, we can look inside of the expects inside of each it block. Let's hone in on the first it block:

it "should accept two numbers as arguments" do

expect { add(2, 3) }.to\_not raise\_error

end

Again, this code reads like english. Here's the somewhat obvious interpretation: *The expectation is that when we call the add method and pass it two number arguments like 2 and 3, it should not get an error.*

Let's focus on the the second it block now:

it "should return the sum of the two numbers" do

expect(add(2, 3)).to eq(5)

expect(add(10, 12)).to eq(22)

end

Like you probably guessed, eq is short for "equal". With that in mind, here's the interpretation of the first expect: *The expectation is that if we pass the add method 2 and 3 as arguments, it should return the sum, which is equal to 5.*

Now that we understand the spec for the add method, let's implement it in /lib/add.rb:

# /lib/add.rb

def add(num\_1, num\_2)

num\_1 + num\_2

end

Nice! Now we know how to read basic RSpec. describe, it, and expect are the fundamental keywords we should focus on, but that's not to say that there aren't other RSpec terms we'll run into in the future. Don't worry, all of these terms are pretty self explanatory. For example, try to interpret the spec we would use for the prime? method:

# /spec/prime\_spec.rb

require "prime"

describe "prime? method" do

it "should accept a number as an argument" do

expect { prime?(7) }.to\_not raise\_error

end

context "when the number is prime" do

it "should return true" do

expect(prime?(7)).to eq(true)

expect(prime?(11)).to eq(true)

expect(prime?(13)).to eq(true)

end

end

context "when the number is not prime" do

it "should return false" do

expect(prime?(4)).to eq(false)

expect(prime?(9)).to eq(false)

expect(prime?(20)).to eq(false)

expect(prime?(1)).to eq(false)

end

end

end

Above we use context additional blocks to outline different scenarios that our code is expected to satisfy. A straightforward interpretation to the first context is: *When the number is prime, it should return true*.

**Wrapping Up**

Here are the core RSpec terms you'll see in every spec file:

* describe names the method being tested
* it expresses the expected behavior of the method being tested
* expect shows how that behavior is tested