**Introduction to Testing**

As a programmer, your job is to write code that is robust and bug-free. So far throughout the course we have *tested* our functions for correctness by calling them and printing out the results. Testing is important because it allows us to verify our code. Later in this topic, we'll explore **RSPEC** to automate tests, but for now let's simply focus on what it means to be test driven.

**What is TDD?**

**TDD (Test Driven Development)** is a strategy to develop programs where the requirements for the program are turned into test cases. Changes to the program are made until the program passes the test cases. Here is a high level overview of a basic TDD workflow to create a method:

1. Write a new test
2. Run all tests & check for fail

* The new test should fail. If it passes, it should be rewritten

1. Make changes to the method to satisfy the tests
2. Run all tests & check for pass

* if any tests fail, go to back step 3
* if all tests pass, but more test coverage is needed, go to step 1

Once we complete all 4 steps, we have completed 1 *iteration* of TDD.

In the course thus far, method problems usually come with examples of how the method should behave. These are examples of simple test cases.

**TDD workflow with**prime?

Time to explore a TDD workflow for developing our classic prime? implementation:

Write a method `prime?(n)` that takes in a number and returns a

boolean indicating whether or not the number is a prime.

A prime number is a number only divisible by two numbers, 1 and itself.

Bear in mind that the point of TDD is to write code methodically and test thoroughly. One iteration of the TDD cycle will work to add one "feature" to our method. We will keep making iterations until our prime? has complete coverage of all scenarios we can give it. Let's go in-depth through the first TDD iteration of prime?. We begin with an empty method definition. Surely this will fail any test cases:

def prime?(num)

end

**Iteration 1: Write a new test**

Let's add our first test. The test below will check if the method is correctly able to identify numbers that are not prime. Notice that the desired output is only about a single requirement for the method. There are multiple example cases, but they all test the same fact: "the method **should** return false when the number is not prime"

# TDD Iteration 1: return false if the number is not prime

# new test

prime?(6) # => false

prime?(8) # => false

prime?(9) # => false

**Iteration 1: Run All Tests and Check for New Failure**

Now that we have added a test, we should run the test to verify that it fails. This step seems trivial but it is very important. If we have followed a true TDD cycle thus far, the test will almost certainly fail as we have not yet implemented code to support the test. However, we cannot assume. **We need to know that the test can fail, otherwise it may not be testing anything at all.** Imagine the scenario where we unknowingly wrote a broken test that always passes. If we don't have the expectation that the test *should* fail as soon as we add it, we will unwittingly add the broken test to our test suite. The bad test will give us a false sense of security and this can be very deadly. Because false positives can be costly, let's take a moment to verify that our new test fails.

# TDD Iteration 1: return false if the number is not prime

def prime?(num)

end

# desired output

prime?(6) # => false

prime?(8) # => false

prime?(9) # => false

# current output

prime?(6) # => nil (FAIL)

prime?(8) # => nil (FAIL)

prime?(9) # => nil (FAIL)

Wooo! Look at all of those fails. This is what we want. *TDD itself is driven by failure.* This seems counterintuitive, but this is what makes TDD so thorough! A test is only valuable if it has more than 1 outcome. Now we know that the test has a possible failure outcome and we can proceed.

**Iteration 1: Make changes to the method**

Now it's time to code! Our goal is to change the method so that it satisfies the failing test. Now we run into another key principle of TDD. **We should only make changes to the method that will support the test. We should not add any extra functionality.** At this stage, we want to write a minimal amount of logic that will allow us to pass the test. This will help keep our code simple and avoid over-engineering. Even if we anticipate further functionality requirements, we should save implementing it until we reach another iteration of the TDD cycle.

So let's be methodical and write code to support the test at hand. That is, we need to implement the logic to check if a number is not prime.

# TDD Iteration 1: return false if the number is not prime

def prime?(num)

(2...num).each do |i|

return false if num % i == 0

end

end

**Iteration 1: Run All Tests**

Now it's time to see if the changes have met the requirements. At this point, **we should run all tests, even those from previous TDD iterations.** It is important to run all tests to verify that the changes we made did not break any existing functionality. If any tests fail, we should go back to the last stage and debug. If all tests pass we can continue.

# TDD Iteration 1: return false if the number is not prime

def prime?(num)

(2...num).each do |i|

return false if num % i == 0

end

end

# desired output

prime?(6) # => false

prime?(8) # => false

prime?(9) # => false

# current output

prime?(6) # => false (PASS)

prime?(8) # => false (PASS)

prime?(9) # => false (PASS)

Nice! We are passing all of the tests. At this point we can consider adding more tests to the method. Although our current implementation of prime? is passing all tests, it lacks complete coverage. Can you think of any scenarios where our current prime? will not work? Let's do additional iterations of TDD to fix these!

**Iteration 2: Write a New Test**

We need more coverage to make sure our prime? can handle prime numbers correctly.

# TDD Iteration 2: return true if the number is prime

prime?(2) # => true

prime?(7) # => true

prime?(11) # => true

**Iteration 2: Run All Tests and Check for New Failure**

# TDD Iteration 2: return true if the number is prime

def prime?(num)

(2...num).each do |i|

return false if num % i == 0

end

end

# desired output

prime?(2) # => true

prime?(7) # => true

prime?(11) # => true

# current output

prime?(2) # => 2...2 (FAIL)

prime?(7) # => 2...7 (FAIL)

prime?(11) # => 2...11 (FAIL)

**Iteration 2: Make changes to the method**

# TDD Iteration 2: return true if the number is prime

def prime?(num)

(2...num).each do |i|

return false if num % i == 0

end

true

end

**Iteration 2: Run All Tests and Check for Pass**

# TDD Iteration 2: return true if the number is prime

def prime?(num)

(2...num).each do |i|

return false if num % i == 0

end

true

end

# desired output

prime?(2) # => true

prime?(7) # => true

prime?(11) # => true

# current output

prime?(2) # => true (PASS)

prime?(7) # => true (PASS)

prime?(11) # => true (PASS)

**Iteration 3: Write a new Test**

We need more coverage to make sure our prime? can handle numbers less than 2 properly.

# TDD Iteration 3: return false if the number is less than 2, since 2 is the smallest prime

prime?(1) # => false

prime?(0) # => false

prime?(-42) # => false

**Iteration 3: Run All Tests and Check for Failure**

# TDD Iteration 3: return false if the number is less than 2, since 2 is the smallest prime

def prime?(num)

(2...num).each do |i|

return false if num % i == 0

end

true

end

# desired output

prime?(1) # => false

prime?(0) # => false

prime?(-42) # => false

# current output

prime?(1) # => true (FAIL)

prime?(0) # => true (FAIL)

prime?(-42) # => true (FAIL)

**Iteration 3: Make Changes to the Method**

def prime?(num)

return false if num < 2

(2...num).each do |i|

return false if num % i == 0

end

true

end

**Iteration 3: Run All Tests and Check for Pass**

def prime?(num)

return false if num < 2

(2...num).each do |i|

return false if num % i == 0

end

true

end

# desired output

prime?(1) # => false

prime?(0) # => false

prime?(-42) # => false

# current output

prime?(1) # => false

prime?(0) # => false

prime?(-42) # => false

We have completed 3 TDD iterations on prime? and we are done! Note that we did not show the previous tests on every iteration to keep things clean in our illustrations. However, you **should** run both the previous and new tests whenever testing the method. This ensures that any new code you write does not break any previous functionality.

**When Should We Use TDD?**

TDD is a popular development strategy to employ in the professional world. From our walkthrough of test-driven prime?, you can definitely see how methodical the process is. Adding one feature and one test at a time is indeed painstaking and tedious at times. If you are chomping at the bit to complete a method (like when you are facing a deadline) you will probably diverge from the TDD process and not follow it as strictly. However, you *should* follow it as best you can to be thorough!

You don't have to employ a true TDD workflow at this point in your programming careers, however you should at least have it in mind. The big picture idea to take away from this lesson is that you should think about how a method should behave before writing it. **If you are designing a method, you should think of example method calls for yourself.** If you understand how the method *should* behave, then you have a clear goal in mind and the code you write should bring you closer to this goal.

When writing an example call about how a method should behave, we note its input (arguments) and output (return value). Designing an example call is like plotting start and end points on a map. After we have established the start and end, we can then choose the turns to take in the hope of reaching the end point.

Next we'll learn about **RSPEC**, a tool we can use to automate our testing and speed up the development process!