## 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 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 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 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 test, 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 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

## 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 understanding 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 straight forward 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

## Debugging Using PRY

As you develop and debug your programs, often times you'll find it useful to test ideas in a "sandbox" environment where you can get quick feedback on the state of your code. We'll use a Ruby **REPL** to this end. To review, **REPL** stands for **R**ead, **E**valuate, **P**rint, **L**oop.

At App Academy, our Ruby REPL of choice is [Pry](http://www.pryrepl.org/). Pry is an open source project developed as an alternative to IRB, the standard Ruby REPL. Pry is a much more powerful tool that allows for more efficient and effective debugging. Because of this, we'll prefer Pry over the standard REPL.

### Installing Pry

If you haven't installed Pry already, you can run the following commands in your command line to get set up:

* gem install pry - this will install the main pry gem
* gem install pry-doc - this will allow us to view ruby documentation in Pry, a nice bonus!

### Using Pry

You can begin a pry session by simply using the command pry in your command line. You'll be greeted by a seemingly unassuming REPL:

$ pry

[1] pry(main)>

Like any REPL, you can type Ruby code line by line to test ideas on the fly. Every line you write will have its evaluation displayed with =>:

[1] pry(main)> greens = ["kale", "spinach", "arugula"]

=> ["kale", "spinach", "arugula"]

[2] pry(main)> greens << "broccoli"

=> ["kale", "spinach", "arugula", "broccoli"]

[3] pry(main)> greens.length

=> 4

You could even copy and paste multiple lines of code into a pry session. That's nothing groundbreaking, but what advanced tools does pry offer us?

#### ls

We can use ls to list methods that we can use in a given context. For example let's say we wanted to know what methods we have available to use on strings by entering ls String in Pry:

[1] pry(main)> ls String

String.methods: try\_convert

String#methods:

% capitalize delete! end\_with? ljust rpartition strip tr!

\* capitalize! delete\_prefix eql? lstrip rstrip strip! tr\_s

+ casecmp delete\_prefix! force\_encoding lstrip! rstrip! sub tr\_s!

+@ casecmp? delete\_suffix freeze match scan sub! undump

-@ center delete\_suffix! getbyte match? scrub succ unicode\_normalize

<< chars downcase grapheme\_clusters next scrub! succ! unicode\_normalize!

<=> chomp downcase! gsub next! setbyte sum unicode\_normalized?

== chomp! dump gsub! oct shell\_split swapcase unpack

=== chop each\_byte hash ord shellescape swapcase! unpack1

=~ chop! each\_char hex partition shellsplit to\_c upcase

[] chr each\_codepoint include? prepend size to\_f upcase!

[]= clear each\_grapheme\_cluster index pretty\_print slice to\_i upto

ascii\_only? codepoints each\_line insert replace slice! to\_r valid\_encoding?

b concat empty? inspect reverse split to\_s

bytes count encode intern reverse! squeeze to\_str

bytesize crypt encode! length rindex squeeze! to\_sym

byteslice delete encoding lines rjust start\_with? tr

Whoa! There are tons of methods that we haven't even learned. Don't be intimidated, documentation is your friend. From this point you can search the [Ruby Docs](https://ruby-doc.org/) in your web browser to gain more insight into the functionality of a method you are unfamiliar with. You can even use Pry to get a summary of a method's documentation. Let's see that next.

#### show-doc

We can use show-doc to show a documentation summary for a given method. For example, if we want to learn more about the end\_with? method from the example above, we can use the pry command show-doc String#end\_with?. We'll need to follow this # notation closely as the output from ls suggests.

[2] pry(main)> show-doc String#end\_with?

From: string.c (C Method):

Owner: String

Visibility: public

Signature: end\_with?(\*arg1)

Number of lines: 7

Returns true if str ends with one of the suffixes given.

"hello".end\_with?("ello") #=> true

# returns true if one of the suffixes matches.

"hello".end\_with?("heaven", "ello") #=> true

"hello".end\_with?("heaven", "paradise") #=> false

Some of the initial verbiage won't be particularly useful to us at this point, but looking at the latter synopsis and examples provided, we can get a really good idea of how one could use this method. If you still can't gather the usage of a method, then you should view the full documentation article in the [Ruby Docs](https://ruby-doc.org/) in your web browser.

#### load

This is not a Pry specific command, but you'll find it useful nonetheless when you want to bring an entire files worth of Ruby code into your Pry session. If we begin a Pry session in the same folder where we have a .rb file. We can load the entire file into Pry and begin playing with it.

Let's say our command line is located in my\_ruby\_code/ and there is a code.rb file within that we want to load into pry. Our folder structure is set up like so:

my\_ruby\_code/

└── code.rb

Then we can load the file into pry:

~/Desktop/my\_ruby\_code$ ls

code.rb

~/Desktop/my\_ruby\_code$ cat code.rb

def is\_prime?(num)

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

return false if num % factor == 0

end

num > 1

end

~/Desktop/my\_ruby\_code$ pry

[1] pry(main)> load "code.rb"

=> true

[2] pry(main)> is\_prime?(16)

=> false

[3] pry(main)> is\_prime?(7)

=> true

[4] pry(main)>

#### show-source

If we want to see the code that implements a method, we call this the source code, we can use the show-source method. Let's say we previously copy and pasted a definition for is\_prime? into our Pry session, then we could view the original source using show-source is\_prime?:

[5] pry(main)> show-source is\_prime?

From: (pry) @ line 1:

Owner: Object

Visibility: public

Number of lines: 7

def is\_prime?(num)

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

return false if num % factor == 0

end

num > 1

end

You'll find the few commands above the most useful as you code, so you should focus on this handful of commands for now. However, there is much more we can do with Pry; feel free to read about it in the [Pry GitHub](https://github.com/pry/pry/blob/master/README.md).

## Using Byebug

"Debugging is twice as hard as writing the code in the first place.

Therefore, if you write the code as cleverly as possible,

you are, by definition, not smart enough to debug it."

- Brian W. Kernighan

In Ruby versions 2.0 and greater, we can use Byebug as a debugging tool. Byebug lets us do many cool things. We can pause execution, step through our code one line at a time, take a look inside key variables and methods. To install it you can run the following command in your terminal:

gem install byebug

### Byebug Cheatsheet

* before running your file
  + require "byebug" - add to the top of your file to gain access to the gem
  + debugger - place this line at a point in your file where you want to begin debugger mode
* while in debugger mode
  + l <start line>-<end line> - list the line numbers in the specified range
    - example: l 3-20
  + step or s - step into the method call on the current line, if possible
  + next or n - move to the next line of executed code
  + break <line num> or b <line num> - place a breakpoint at the specified line number, this will pause execution again
  + continue or c - resume normal execution of the code until a breakpoint
  + display <variable> - automatically show the current value of a variable

### Running Byebug

After installing byebug, you'll have to add a few lines of code to your .rb file to begin using it. We'll first need to require "byebug" to gain access to the gem. Then, we can add a debugger line to pause execution of our code and hop into debugging mode. Let's set up some first\_n\_primes code to use the debugger. There are no bugs in the following code to be found, instead we'll focus on learning the mechanics of byebug. We'll also number the lines as byebug would:

# code.rb

1: require "byebug" #

2:

3: def is\_prime?(number)

4: (2...number).each do |factor|

5: return false if number % factor == 0

6: end

7:

8: number > 1

9: end

10:

11: def first\_n\_primes(num\_primes)

12: primes = []

13: num = 2

14: debugger #

15: while primes.length < num\_primes

16: primes << num if is\_prime?(num)

17: num += 1

18: end

19: primes

20: end

21:

22: p first\_n\_primes(11)

You'll want to reference this initial numbering if ever you get lost in the big picture as you follow this reading.

Now that we have those two byebug lines in the file we want to debug, we can execute this file with the usual ruby code.rb. Execution of the code will take place as normal, until we run the debugger line, at which point we pause the runtime at that line:

# [10, 19] in /Users/appacademy/Desktop/lecture/code.rb

10:

11: def first\_n\_primes(num\_primes)

12: primes = []

13: num = 2

14: debugger

=> 15: while primes.length < num\_primes

16: primes << num if is\_prime?(num)

17: num += 1

18: end

19: primes

(byebug)

From here we can check the current value of a variable by simply referencing its name

# ...

(byebug) primes

[]

(byebug) num

2

### display

We can use the display command to automatically set up tracking for variables:

# [10, 19] in /Users/appacademy/Desktop/lecture/code.rb

10:

11: def first\_n\_primes(num\_primes)

12: primes = []

13: num = 2

14: debugger

=> 15: while primes.length < num\_primes

16: primes << num if is\_prime?(num)

17: num += 1

18: end

19: primes

(byebug) display primes

1: primes = []

(byebug) display num

2: num = 2

### next

We can execute subsequent code using the next command. Be aware that using next doesn't always mean advancing to the next sequential line number. In other words if we are on line 17, the next executed line may not be line 18. Instead we move to the next line according to normal control flow; so we obey all of the behavior of loop iterations, conditional branches, etc.., **except we won't step into any other method calls**. Let's use next a few times:

(byebug) next

1: primes = []

2: num = 2

# [15, 18] in /Users/appacademy/Desktop/lecture/code.rb

15: while primes.length < num\_primes

=> 16: primes << num if is\_prime?(num)

17: num += 1

18: end

(byebug) next

1: primes = [2]

2: num = 2

# [15, 18] in /Users/appacademy/Desktop/lecture/code.rb

15: while primes.length < num\_primes

16: primes << num if is\_prime?(num)

=> 17: num += 1

18: end

(byebug) next

1: primes = [2]

2: num = 3

# [15, 18] in /Users/appacademy/Desktop/lecture/code.rb

15: while primes.length < num\_primes

=> 16: primes << num if is\_prime?(num)

17: num += 1

18: end

(byebug) next

1: primes = [2, 3]

2: num = 3

# [15, 18] in /Users/appacademy/Desktop/lecture/code.rb

15: while primes.length < num\_primes

16: primes << num if is\_prime?(num)

=> 17: num += 1

18: end

(byebug) next

1: primes = [2, 3]

2: num = 4

# [15, 18] in /Users/appacademy/Desktop/lecture/code.rb

15: while primes.length < num\_primes

=> 16: primes << num if is\_prime?(num)

17: num += 1

18: end

### step

You may have noticed that using next will not bring us into the evaluation of is\_prime?(num) (line 16). Once execution is paused on a line containing a method call, we can use the step command to step into that method. Picking up on the iteration where we left off, let's step into line 16's call.

(byebug) step

1: primes = (undefined)

2: num = (undefined)

# [1, 10] in /Users/appacademy/Desktop/lecture/code.rb

1: require "byebug"

2:

3: def is\_prime?(number)

=> 4: (2...number).each do |factor|

5: return false if number % factor == 0

6: end

7:

8: number > 1

9: end

10:

(byebug) display number

3: number = 4

(byebug) display factor

4: factor = (undefined)

Since we are stepping into a different method call, our previously tracked variables of primes and num are undefined in this new context. Because of this we set up tracking on new variables that are relevant as we debug is\_prime?, mainly number and factor. From here, you can use next to walk through the code like before.

### break and continue

Let's say we are done with the bulk of our debugging and we want to fast forward to a much later point of the execution. Instead of mashing the next command barbarically (bugs test the best of us), you can use break <line num> to set a future breakpoint in the code. Setting a breakpoint in the code will mark a line that we want to pause and reenter debugging mode on. We can then use the continue to resume normal execution until we hit the breakpoint! Let's set up a breakpoint on the return statement for first\_n\_primes (line 19) and then continue:

(byebug) break 19

# Created breakpoint 1 at /Users/appacademy/Desktop/lecture/code.rb:19

(byebug) continue

# Stopped by breakpoint 1 at /Users/appacademy/Desktop/lecture/code.rb:19

1: primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]

2: num = 32

3: number = (undefined)

4: factor = (undefined)

# [13, 22] in /Users/appacademy/Desktop/lecture/code.rb

13: num = 2

14: debugger

15: while primes.length < num\_primes

16: primes << num if is\_prime?(num)

17: num += 1

18: end

=> 19: primes

20: end

21:

22: p first\_n\_primes(11)

Amazing! We are now at the point of execution where we finished the while loop, and our primes array contains the first 11 prime numbers. Just as intended.

### Wrapping Up

When you use byebug out in the wild, you'll want to really analyze how your variables are changing over time and what logic is executed. Bugs are always a product of a disconnect between what we want to happen in the code and what is actually happening in the code. Because of this, don't make assumptions based how you want the code to run, use byebug to truly witness how it runs.

## Common Error Types

As you've been progressing through this course, you've likely run into few different error types by mistake. Let's take a look at these different errors and understand what caused them. If we are able to identify what exactly is causing an issue, we can act accordingly and ensure bug-free code. In this reading, the errors we refer to are those that prevent our code from completing execution. That is, Ruby will terminate execution of the program immediately when any of these errors are hit.

### Reading an Error Message

Let's say we had this broken ruby code in a code.rb file:

def say\_hello

puts "hi!"

The code above is obviously incorrect because of a missing end. When we execute this code we'll receive an error message:

$ ruby code.rb

code.rb:2: syntax error, unexpected end-of-input, expecting keyword\_end

puts "hi!"

A ruby error message begins with the file name that we tried to execute (code.rb) as well as a line-number (2) for the error. Given this information, we ought to verify that we ran the correct file (it happens to the best of us :)) and scan line 2 for any obvious errors. It may be the case that the suggested error line is nowhere near the actual cause of the error, especially when we are dealing with a SyntaxError, so don't tunnel vision into checking that single line.

For example, let's say we had a different kind of typo in the code:

def check\_num(num) # 1

if num > 0 # 2

p "positive" # 3

else if num < 0 # 4

p "negative" # 5

else # 6

p "zero" # 7

end # 8

end # 9

The Ruby syntax coloring above makes this mistake obvious, but peek at the error message we would get when executing this code:

$ ruby code.rb

code.rb:9: syntax error, unexpected end-of-input, expecting keyword\_end

Ruby is complaining about the line 9, but the real mistake is on line 4. The suggested line number isn't useful here, but the type of error, syntax error, will give us an idea of what kind of mistake we are looking for.

Let's break down common error types next. The following list is by no means comprehensive, but you will run into these errors most often.

### SyntaxError

A SyntaxError results from incorrectly structuring our code. In proper English we must follow certain grammar rules. In the same way proper Ruby must follow particular syntax rules. Because we can violate the "grammar" of Ruby in many different ways, there is no hard and fast rule to spot a SyntaxError. In general, you should be scanning your code for structural mistakes:

# SyntaxError because we need to have a value to assign

my\_variable =

### NameError

A NameError results from referring to a variable or method name that has not been defined:

my\_variable = 42

p my\_variabel # NameError because this name has not been defined previously

### NoMethodError

A NoMethodError results from referring to an undefined method. It is considered a special instance of a NameError:

def say\_hello(name)

p "hi " + name

end

hello("world") # NoMethodError because `hello` is not a defined method

### ArgumentError

A ArgumentError results from incorrectly specifying arguments for a method call. Typically this means passing too little or too many arguments:

def say\_hello(first\_name, last\_name)

p "hi " + first\_name + " " + last\_name

end

say\_hello("world") # ArgumentError because we gave 1 arg, but `say\_hello` expects 2 args

### TypeError

A TypeError results from performing an operation with incompatible data types:

5 + "spaghetti" # TypeError because we cannot add a Integer and a String toge

## Blocks

Let's take a closer look at main feature of the Ruby language: **blocks**. Until this point you have been utilizing blocks with enumerable methods like each, map, and times. A block is a chunk of code that is passed into a method to be executed. We explored blocks in previous sections, but now we'll want to take a peek under the hood and use blocks in our methods. First, let's do a quick refresher.

### Block Basics

There are two ways to pass a block into a method.

We can use {...} brace syntax for blocks that only contain a single line of code:

[1,2,3].map { |num| -(num \* 2) } # => [-2, -4, -6]

Or, we can use do...end syntax for multiline blocks:

[1,2,3].map do |num|

doubled = num \* 2

-doubled

end # => [-2, -4, -6]

Brace {...} blocks and do...end blocks are functionally equivalent, we just prefer do...end for blocks that contain many lines.

Blocks can accept parameters if we name them between pipes (|param\_1, param\_2, etc.|).

We know that when we pass a block into map, map will execute the block, passing in every element of the array one by one. It will take each evaluation of the block and make that value an element of the new array that the map method will return.

Blocks are somewhat similar to methods in that both can contain lines of code as well as take in parameters. However, an important distinction to make is that the return keyword pertains to methods, not blocks. Let's take a look at a common pitfall:

# Correct:

def double\_eles(arr)

arr.map do |ele|

ele \* 2

end

end

double\_eles([1,2,3]) #=> [2,4,6]

# Incorrect:

def double\_eles(arr)

arr.map do |ele|

return ele \* 2

end

end

double\_eles([1,2,3]) #=> 2

Looking at the incorrect implementation of double\_eles, we use the return keyword within the block. If we use return we will be returning out of the entire double\_eles method on the first iteration of map. In other words, using return in a block will not make the block evaluate to the return value. The return will force the outer method to evaluate to the return value. This is a huge reason why we should treat blocks and methods as somewhat distinct concepts.

### Using Methods as Blocks

It is very, very common to have blocks that take an argument and call a single method. For example:

["a", "b", "c"].map { |str| str.upcase } #=> ["A", "B", "C"]

[1, 2, 5].select { |num| num.odd? } #=> [1, 5]

Ruby allows us to use cleaner syntax when we have simple blocks that follow the above pattern. Let's refactor the above example to use this shortcut:

["a", "b", "c"].map(&:upcase) #=> ["A", "B", "C"]

[1, 2, 5].select(&:odd?) #=> [1, 5]

You may find this syntax quite strange. Let's notice a few things about how we are using map. We call map and pass in a single argument, &:upcase. :upcase is a symbol referring to the String#upcase method. We use & to convert this "method" into an object that we can pass into map. In Ruby, we cannot directly pass a method into another method, so we need to use the & operator. In the next lecture we'll explore the intricacies of &.

For now, here's a hard and fast rule you can use to optimize some blocks. If you are calling a method like map, passing a block that has this general structure:

array.map { |block\_param| block\_param.method }

Then you can rewrite it as:

array.map(&:method)

When employing this trick, be aware of what &:method you are using. The method you choose should be compatible with your data:

["a", "b", "c"].map(&:upcase) # => ["A", "B", "C"]

[1, 2, 3].map(&:upcase) # NoMethodError: undefined method 'upcase' for Integer

The second map is invalid because we can't use upcase on numbers!

## Procs

Now that we have reviewed the basics of using methods that accept blocks, how can we write our own custom methods that utilize blocks? We'll need knowledge of **Procs**. Procs and blocks go hand in hand. A proc is an object that contains a block. We need procs because they allow us to save blocks to variables so we can manipulate them in our code.

### Creating a Proc

We cannot pass a block into a method, but we can turn the block into a proc and pass that proc into a method to be executed. Think of a proc as a wrapper around a block! Let's take a look at how we can create a proc manually. We'll need to use Proc.new and give it the block to wrap up:

doubler = Proc.new { |num| num \* 2 }

p doubler # #<Proc:0x00007f9a8b36b0c8>

Printing the proc gives a somewhat cryptic output,"<Proc:0x00007f9a8b36b0c8>". This is the visual representation of the proc object, but it's not too important. Since a proc is a normal ruby object, we are free to save the proc to a variable, doubler. This is an important fact about procs. In comparison, we cannot save a plain block to a variable:

# incorrect

doubler = { |num| num \* 2 } # SyntaxError: unexpected '}'

### Calling a Proc

Now that we have a proc, let's execute the code it contains. To do this, we need to use the Proc#call method:

doubler = Proc.new { |num| num \* 2 }

p doubler.call(5) # => 10

p doubler.call(7) # => 14

When calling the proc, we can pass in any arguments the block expects. We can also call the proc as many times as we please! The Proc#call method will evaluate to the last line of code executed within the block. Let's take a look at this with a multiline block:

is\_even = Proc.new do |num|

if num % 2 == 0

true

else

false

end

end

p is\_even.call(12) # => true

When we do is\_even.call(12), the result is true because the last executed expression in the block is true.

### Passing a Proc to a Method

Since a proc has an advantage of being an object, we can pass this proc object into a method. Let's say we had this method:

def add\_and\_proc(num\_1, num\_2, prc)

sum = num\_1 + num\_2

p prc.call(sum)

end

The add\_and\_proc method will take in two numbers and a proc. It will call the proc, giving it the sum of the two numbers, and finally print the result of the proc. Let's see it in action. To use this method, we'll also need a proc to pass in:

def add\_and\_proc(num\_1, num\_2, prc)

sum = num\_1 + num\_2

p prc.call(sum)

end

doubler = Proc.new { |num| num \* 2 }

add\_and\_proc(1, 4, doubler) # => 10

square = Proc.new { |num| num \* num }

add\_and\_proc(3, 6, square) # => 81

negate = Proc.new { |num| -1 \* num }

add\_and\_proc(3, 6, negate) # => -9

Notice that we can pass different blocks/procs into the method to really vary its behavior. Now our add\_and\_proc method is pretty versatile. The only knock against this code is that we have to repeatedly wrap each block in a proc using Proc.new. Fret not! Ruby affords us a way to automatically convert a block into a proc when passed into method. Let's compare the two ways, side by side:

# Version 1: manual, proc accepting method

def add\_and\_proc(num\_1, num\_2, prc)

sum = num\_1 + num\_2

p prc.call(sum)

end

doubler = Proc.new { |num| num \* 2 }

add\_and\_proc(1, 4, doubler) # => 10

# Version 2: automatic conversion from block to proc

def add\_and\_proc(num\_1, num\_2, &prc)

sum = num\_1 + num\_2

p prc.call(sum)

end

add\_and\_proc(1, 4) { |num| num \* 2 } # => 10

Take a moment to compare the two methods and how we call them. In version 2, it seems that we only pass two number arguments to the method, but the definition lists 3 arguments. This is because the third argument, prc, will refer to the block we pass! By using the & operator on the third parameter, ruby knows to automatically convert the block into proc for us.

Because of the &prc parameter we must always pass a block into add\_and\_proc, we can no longer pass in a proc because a conversion from block to proc must take place.

def add\_and\_proc(num\_1, num\_2, &prc)

sum = num\_1 + num\_2

p prc.call(sum)

end

doubler = Proc.new { |num| num \* 2 }

add\_and\_proc(1, 4, doubler) # ArgumentError: wrong number of arguments (given 3, expected 2)

Here are two general tips that you can use to reason out whether a method expects a proc or a block.

def my\_method(prc)

# ...

end

# By looking at the parameter `prc` we know that my\_method must be passed a proc:

my\_proc = Proc.new { "I'm a block" }

my\_method(my\_proc)

def my\_method(&prc)

# ...

end

# By looking at the parameter `&prc` we know that my\_method must be passed a block,

# because & denotes conversion from block to proc here:

my\_method { "I'm a block" }

### Using &

We already saw how & can be used to convert a block into a proc. But it can also be used for the opposite, that is, convert a proc into a block. We know, we know, that sounds hopelessly confusing. Let's show it off using our last example. We already established that this code can only accept blocks now, in this context &prc is converting a block to a proc. If we try to pass our method the doubler proc, we will get an error. This is because doubler is a proc, not a block!

def add\_and\_proc(num\_1, num\_2, &prc)

sum = num\_1 + num\_2

p prc.call(sum)

end

doubler = Proc.new { |num| num \* 2 }

add\_and\_proc(1, 4, doubler) # Error

However, we can use & again to convert a proc to a block. In other words, if doubler is a proc, then &doubler is a block:

def add\_and\_proc(num\_1, num\_2, &prc)

sum = num\_1 + num\_2

p prc.call(sum)

end

doubler = Proc.new { |num| num \* 2 }

add\_and\_proc(1, 4, &doubler) # => 10

Since & either turns a block into a proc or proc into a block, here's a rule you can use to identify what is happening. It all depends on context: when we see & in the parameters for a method definition we know it will convert a block to a proc and when we see & in the arguments for a method call we know it will convert a proc to a block. Another give away is that doubler is most certainly already a proc since we used Proc.new, so &doubler converts that proc into a block.

### Another Example

The dual function of & is the biggest point of confusion for blocks and procs so let's step through another example in familiar territory. We know that map is a built-in method that must be given a block:

[1,2,3].map { |num| num \* 2 } # => [2, 4, 6]

However, if we have a proc and want to use it with map, we can use & to convert it to a block:

doubler = Proc.new { |num| num \* 2 }

[1,2,3].map(&doubler) # => [2, 4, 6]

You can download the files used in this lecture to try RSpec for yourself:

[Download Demo](https://s3-us-west-1.amazonaws.com/aao-alpha/assets/topics/testing/rspec_demo.zip) [Download Demo Solution](https://s3-us-west-1.amazonaws.com/aao-alpha/assets/topics/testing/rspec_demo_solution.zip)

In the video, you'll notice that Alvin is using a different text editor (Atom) and a separate Terminal window. You'll be able to follow along just the same by using VSCode and it's integrated terminal (ctrl + ~). He likes to rebel sometimes, but has since converted fully to using VSCode :).