**Debugging**

Any program of sufficient complexity is unlikely to work the first time. You will make mistakes. Skilled and unskilled developers write a similar number of bugs. The difference is that skilled developers are able to quickly identify and fix bugs.

A rule of thumb is that it takes 10x as long to debug code as to write it. Master debugging and you master programming.

**Rule #1: Read the error**

Rule #1 is to **READ THE ERROR**.

Do NOT simply jump back to your program and start fiddling with things to see if you can get it to work. Always read the error. Oftentimes it tells you everything you need to know.

The error gives you valuable pieces of information. The first three are absolutely essential to read and understand whenever an error occurs.

* Error type
* Error message
* Line number on which the error occurred
* Chain of methods that were called leading up to it (referred to as the **stack trace**)

If you encounter an error and you are unclear about what the error type and/or message is telling you, stop and ask an instructor to explain it to you. Learning to understand errors and error messages is critical to developing your abilities as a programmer.

**Perform a Mental Stack Trace**

The stack trace below the error message can be extremely helpful, but it usually won't give you the exact information you need to fix your bug. What it does tell you is the path your program took to get to wherever the error happened.

Whenever you encounter a bug your ability to track it down will be dependent on your ability to trace the logic of your own code.

Interrogate your code actively. Why did the bug happen? What are the values of the key variables at key points in your program? What did each line evaluate to leading up to the bug?

Do not passively stare at your code or simply assume that what you think happened is what actually happened (this is what got you in trouble in the first place!). Some strategies include:

* Break your code down into more testable chunks and actively run each of those chunks to test if they're working.
* Use p statements often; use them to check what the values of variables are, that methods are called as expected, etc.
* Use the debugger.

The key with bugs and errors is to really get into the mind of the machine. You must understand what is happening in the code. To do so, you must seek out helpful feedback from the program, constantly testing your assumptions about what is happening.

You are a programmer. You hunt bugs. Hunt well.

**Write code that's testable**

Let's consider a Ruby script that is supposed to print the first 100 primes:

# primes.rb

primes = []

num = 1

while primes.count < 100

is\_prime = true

(1..num).each do |idx|

if num % idx == 0

is\_prime = false

break

end

end

if is\_prime

primes << num

end

end

puts primes

This program doesn't work; it doesn't look like it ever returns. Where's the bug (or bugs)?

The bugs could be anywhere, but we don't have the ability to isolate and test individual parts of the code. When we load up this file, it immediately starts running all the code.

Let's make this more *testable*: let's break the code into small, bite-sized methods. Small methods are easier to test, because you can test each part independently.

General hint: when you write a *script*, write all your code inside of methods. Only a very little bit of code should be written at the top level to kick things off.

# primes.rb

def prime?(num)

(1..num).each do |idx|

if num % idx == 0

return false

end

end

end

def primes(num\_primes)

ps = []

num = 1

while ps.count < num\_primes

primes << num if prime?(num)

end

end

if \_\_FILE\_\_ == $PROGRAM\_NAME

puts primes(100)

end

This code uses a common trick. We will want to be able to load our code without running it immediately. In particular, we'd like to directly call the methods and diagnose whether each is working. But before we were blocked because the program immediately started executing the script and entering an infinite loop.

The solution is the trick if \_\_FILE\_\_ == $PROGRAM\_NAME. This checks to see if the currently running program ($PROGRAM\_NAME) is the same as the current file (primes.rb). If so, then this is being invoked as a script, so we should kick things off. Otherwise, we're loading it as part of some other program (like irb or Pry), and we shouldn't do more than load the method definitions so that someone else may use them.

Great. Now we can test the prime? and primes pieces individually. If one works and the other doesn't, we can focus on the single broken method. Even if both are broken, we can fix prime? first, and then try to debug primes knowing that prime? at least works.

Also, because prime? and primes do one simple thing, we know what they're *supposed* to do: prime?(2) should be true. prime?(4) should be false. primes(3) should be [2, 3, 5].

This is better than a huge, black-box method which does a bunch of complicated stuff where it's hard to even know what the right answer should be.

**Pay technical debt**

If you encounter buggy code that is poorly decomposed into methods, **fix the design immediately**. You're going to want to fix the design eventually anyway; refactoring will probably create new bugs to fix, so you might as well deal with this bug at the same time.

More importantly, good code is the gift that keeps on giving. If this code is broken today, it's safe to assume that it will bite you in the ass with another bug a few days from now, too. And every time you come back to this code, you'll be fighting its poor design as you try to deal with it. Try to fix it now once and for all.

In the rush to complete projects, bad design is sometimes a compromise made to finish a project on-time. This is called *technical debt*. It's okay to take out debt like this, just like it's okay to take out financial debt. But the more debt you take out, the higher the payments in the form of your time.

If you find yourself struggling with a tough bug in the midst of some poorly written code, admit that your debt has caught up with you, bite the bullet and refactor.

**Don't read the source**

We haven't found out what's wrong yet. You might be tempted to first look carefully at prime? and primes, try to reason through them, and spot the bug. You may be able to do this with my simple example.

**Do not spend more than 1min doing this in real life**. Yes, many silly bugs can be spotted if you stare at the code, but many other silly bugs are difficult to spot because our eyes play tricks on us. You know how you can still read a paragraph with the spaces taken out? For the same reason, it's hard to spot silly bugs, because you know what the code is *supposed* to do.

Your bug may not be a simple bug. If it's at all non-trivial, it will be *very* hard to spot. The best way to find a bug like this is to take your code step-by-step. We'll see how to do that soon.

Yes, when debugging you should look at the source to familiarize yourself with the code. The bug may jump out at you. If not, don't worry. We're about to learn better techniques.

**Use a REPL to isolate the problem**

Now that we've broken the code up into testable bits, let's actually test those parts. That lets us quickly isolate the problem to a few lines.

Open the Pry REPL. Make sure you have done gem install pry first.

david ~/Dropbox/TA $

pry

Load your file and start testing.

[1] pry(main)> load 'primes.rb'

=> true

[2] pry(main)> prime?(2)

=> false

Awesome. We've already found a *regression*; an input which produces the wrong output. There might also be problems with primes, but it would have been a real PITA to try to fix those when the underlying prime? method is broken.

Decomposition for the win.

Now we need to take a more fine-grained look at exactly what is wrong with our prime? method.

**Use a debugger to zero in on the problem**

In Ruby versions 2.0 and greater, we use byebug for debugging:

gem install byebug

Byebug lets us do many cool things. We can step through our code one line at a time, and along the way...

* check the value of our variables at any time (no p required!)
* continuously watch the value of a variable, so that we can see when it changes
* change the value of variables in the middle of program execution
* set breakpoints so that we can pause whenever we reach a certain line in our code
* examine the call stack to determine exactly which methods brought us to a certain line of code
* execute short snippets of code to test an idea (just like in pry or irb)

Note: a minor downside of the byebug gem is that it does not support colored syntax highlighting. However, I will apply coloring to the following examples so that they are easier to read.

**Step through code**

Once you've isolated a bug to a small amount of code, the best way to uncover the problem is to single-step through the code, checking what the program does along the way. This is what a *debugger* (such as byebug) does.

To start, we need to modify our program slightly so that we *drop into* the debugger when prime? is called:

require 'byebug'

def prime?(num)

debugger # drops us into the debugger right after this point

(1..num).each do |idx|

if num % idx == 0

return false

end

end

end

def primes

# ... etc.

**N.B.** Don't forget to require 'byebug' at the top of your file.

Let's load our code into pry and call primes?(2) to start testing the primes? method. The debugger at the top of prime? will pause our code there. At this point, you are basically like [Neo](http://img1.wikia.nocookie.net/__cb20131002032735/matrix/images/b/b5/Matrix-neo-stops-bullets-wallpaper.jpg).

david ~/Dropbox/TA $

pry

[1] pry(main)> load 'primes.rb'

=> true

[2] pry(main)> prime?(2)

[1, 10] in primes.rb

1: require 'byebug'

2:

3: def prime?(num)

4: debugger # drops us into the debugger right after this point

5:

=> 6: (1..num).each do |idx|

7: if num % idx == 0

8: return false

9: end

10: end

(byebug)

We are now inside of the byebug debugger, inside of the pry REPL. (Note that the byebug debugger is not built into pry. If we didn't have require 'byebug' at the top of our file then pry would have raised an error when it came to line 4.) The debugger prompt looks like (byebug). Our position is indicated by the arrow; we're at line 6.

On line 6 we are calling the each method on the range (1..num). step (or s) is the command that we use to step into a method call. There is a bug in Ruby 2.1 that causes us to get stuck on line 6 the first time we type step, so we'll have to step twice.

(byebug) step

[1, 10] in primes.rb

1: require 'byebug'

2:

3: def prime?(num)

4: debugger # drops us into the debugger right after this point

5:

=> 6: (1..num).each do |idx|

7: if num % idx == 0

8: return false

9: end

10: end

(byebug) step

[2, 11] in primes.rb

2:

3: def prime?(num)

4: debugger # drops us into the debugger right after this point

5:

6: (1..num).each do |idx|

=> 7: if num % idx == 0

8: return false

9: end

10: end

11: end

(byebug)

You can see how the arrow has advanced. Let's see what happens at this if statement. Since there is no method call on line 7, we advance with next (or n).

(byebug) next

[3, 12] in primes.rb

3: def prime?(num)

4: debugger # drops us into the debugger right after this point

5:

6: (1..num).each do |idx|

7: if num % idx == 0

=> 8: return false

9: end

10: end

11: end

12:

(byebug)

Wait; we entered the if? How? Let's check the values of num and idx:

(byebug) num

2

(byebug) idx

1

Hmm... We shouldn't check for divisibility by one. Upon reflection, we shouldn't start the index at 1 at all; we should start at 2. We can quit byebug by typing exit, then y to confirm.

Let's fix our code:

def prime?(num)

debugger

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

if num % idx == 0

return false

end

end

end

Let's go back into pry and see if prime? works now:

david ~/Dropbox/TA $

pry

[1] pry(main)> load 'primes.rb'

=> true

[2] pry(main)> prime?(2)

[1, 10] in primes.rb

1: require 'byebug'

2:

3: def prime?(num)

4: debugger

5:

=> 6: (2..num).each do |idx|

7: if num % idx == 0

8: return false

9: end

10: end

(byebug)

We still have our debugger on line 4, and so we stop at the next line of code (line 6). Right now, though, we don't want to debug step-by-step; we just want to see the result of calling prime?(2). We can type c (for continue) to tell the debugger to keep running the code.

9: end

10: end

(byebug) c

=> false

[3] pry(main)>

The code never brought us back to the debugger at line 4, so the method finished, and spit us back out at the pry prompt. We can see that our method returned false, though, so we still have work to do.

The line we really want to focus on is line 8, because that's where we are erroneously returning false. So, let's add a breakpoint to line 8 with the break command. This tells byebug to make sure to stop when we hit line 8. I then tell the program to run freely until it hits a breakpoint (c, or continue), and shortly thereafter we arrive at line 8.

[3] pry(main)> prime?(2)

[1, 10] in primes.rb

1: require 'byebug'

2:

3: def prime?(num)

4: debugger

5:

=> 6: (2..num).each do |idx|

7: if num % idx == 0

8: return false

9: end

10: end

(byebug) break 8

Created breakpoint 1 at primes.rb:8

(byebug) c

Stopped by breakpoint 1 at primes.rb:8

[3, 12] in primes.rb

3: def prime?(num)

4: debugger

5:

6: (2..num).each do |idx|

7: if num % idx == 0

=> 8: return false

9: end

10: end

11: end

12:

(byebug)

Now we can have a look at the relevant variables.

(byebug) num

2

(byebug) idx

2

Groan. We are testing whether num is divisible by itself. That's because (2..num) includes num; we wanted (2...num). Fix and then reload:

[1] pry(main)> load 'primes.rb'

=> true

[2] pry(main)> prime?(2)

[1, 10] in primes.rb

1: require 'byebug'

2:

3: def prime?(num)

4: debugger

5:

=> 6: (2...num).each do |idx|

7: if num % idx == 0

8: return false

9: end

10: end

(byebug) c

=> 2...2

[3] pry(main)>

Weird, but better; at least this isn't false. But because we don't return true at the end of prime?, the last returned value is used. Note that Enumerable#each returns self; in this case the range itself. Let's finish fixing this method.

def prime?(num)

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

if num % idx == 0

return false

end

end

true

end

Does it really work? We ought to check with a few values other than 2:

[5] pry(main)> load 'primes.rb'

=> true

[6] pry(main)> prime?(2)

=> true

[7] pry(main)> prime?(3)

=> true

[8] pry(main)> prime?(10)

=> false

[9] pry(main)> prime?(17)

=> true

All looks good. Notice how we can quickly check a number of values in the REPL.

**Reading stack traces**

Now that prime? appears to be working, it's time to test primes. I've put a call to debugger at the start of primes. Again, let's use pry:

[10] pry(main)> load 'primes.rb'

=> true

[11] pry(main)> primes(2)

[11, 20] in primes.rb

11: end

12:

13: def primes(num\_primes)

14: debugger

15:

=> 16: ps = []

17: num = 1

18: while ps.count < num\_primes

19: primes << num if prime?(num)

20: end

(byebug) c

ArgumentError: wrong number of arguments (0 for 1)

from primes.rb:13:in 'primes'

The method failed. When an exception is thrown and no code catches and handles the exception, then the program stops (crashes) and the exception and line where it occurred are printed.

The line ArgumentError: wrong number of arguments (0 for 1) states the exception type (ArgumentError) and the message. This message says that we're passing the wrong number of arguments to a method: zero arguments instead of one argument.

Where did this happen in the code? The subsequent line tells us: from primes.rb:13:in 'primes'. If we want more detail about just how we came to this line of code, we can type wtf to look at the *stack trace*. (You can also add question marks and exclamation points to get longer stack traces, like wtf?, wtf?!!, etc.)

[9] pry(main)> wtf

Exception: ArgumentError: wrong number of arguments (0 for 1)

--

0: /home/david/Dropbox/app-academy-TA/primes.rb:13:in 'primes'

1: /home/david/Dropbox/app-academy-TA/primes.rb:19:in 'primes'

2: (pry):21:in '\_\_pry\_\_'

3: /home/david/.rbenv/versions/2.1.2/lib/ruby/gems/2.1.0/gems/pry-0.10.1/lib/pry/pry\_instance.rb:355:in 'eval'

4: /home/david/.rbenv/versions/2.1.2/lib/ruby/gems/2.1.0/gems/pry-0.10.1/lib/pry/pry\_instance.rb:355:in 'evaluate\_ruby'

[10] pry(main)>

The top line of the stack trace tells us what method (primes) and line of code (13) were executing when the error occurred. The next line tells us what called primes; it looks like primes calls itself, on line 19. The next line starts with '(pry)'; this is pry executing the code we gave it.

It's certainly odd that primes is calling itself. Let's check this out:

[12] pry(main)> primes(2)

[11, 20] in primes.rb

11: end

12:

13: def primes(num\_primes)

14: debugger

15:

=> 16: ps = []

17: num = 1

18: while ps.count < num\_primes

19: primes << num if prime?(num)

20: end

(byebug)

Ah. Line 19 says that if prime?(num), then primes << num. This tries to call primes again, but what we really wanted was to push num into our list, named ps. This is confusing because it's not super clear that primes is calling a method (equivalent to self.primes).

Fix this and restart pry.

[13] pry(main)> load 'primes.rb'

=> true

[14] pry(main)> primes(2)

[11, 20] in primes.rb

11: end

12:

13: def primes(num\_primes)

14: debugger

15:

=> 16: ps = []

17: num = 1

18: while ps.count < num\_primes

19: ps << num if prime?(num)

20: end

(byebug) c

=> nil

Oops. A few more simple bugs. You catch them.

**Step vs Next**

Here, using n, I have line by-line advanced through primes:

[14, 23] in primes.rb

14: debugger

15:

16: ps = []

17: num = 1

18: while ps.count < num\_primes

=> 19: ps << num if prime?(num)

20: end

21: end

22:

23: if \_\_FILE\_\_ == $PROGRAM\_NAME

(byebug)

I could type n to execute this line and advance (back to line 19, actually). But what if I wanted to "step into" the call to prime?? To do this, I use s or step:

(byebug) step

[1, 10] in primes.rb

1: require 'byebug'

2:

3: def prime?(num)

=> 4: (2...num).each do |idx|

5: if num % idx == 0

6: return false

7: end

8: end

9:

10: true

(byebug)

This is handy when you want to go down into methods. If I'm no longer interested in stepping through all of prime?, I can finish it and move up a level by using finish:

(byebug) finish

[14, 23] in primes.rb

14: debugger

15:

16: ps = []

17: num = 1

18: while ps.count < num\_primes

=> 19: ps << num if prime?(num)

20: end

21: end

22:

23: if \_\_FILE\_\_ == $PROGRAM\_NAME

(byebug)

**Debugging and Testing**

We've gone through a lot of work testing that these methods work as they should. It would be good if we could record these tests so that they can be run in the future, to make sure new bugs do not sneak in as we continue to develop the software. We'll talk later about RSpec, a way to write tests that can be automatically run by a system called Guard.

When a bug is discovered, it is good practice to write a new test that verifies we don't make that mistake again.

**Resources**

* [why Pry is awesome](http://www.sitepoint.com/rubyists-time-pry-irb/)
* [Byebug documentation](https://github.com/deivid-rodriguez/byebug/blob/master/README.md)