**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 on how you want the code to run, use byebug to truly witness how it runs.