From: **lizbot** deid@appacademy.io
Subject: Evening Announcements for 16 Week - W1D2
Date: September 1, 2020 at 9:09 PM
To: 2020-08-31-ny@appacademy.io
Cc: instructors-ny@appacademy.io

Ahoy, Mateys!

Nightly Workflow

- Zip and submit your project to open.appacademy.io if you haven't already
- **IMPORTANT: Submit your daily report on Progress Tracker before 9am the next day.** Make sure to do this every night!
- Note there is no *Homework* section tonight (looking at **W1D3**). **Your homework tonight is to review all the material learned so far, and work on finishing projects you haven't completed.**
- Watch the solutions
    - Note: You can speed up the video by clicking on the *Gear Icon,* and clicking on *Speed*

Testing & Debugging Key Points

Welcome to the world of debugging! Debugging is a crucial skill to your success as a software developer! It is important you take every opportunity to practice debugging using these new techniques to get better and better as we go through the cohort and the content gets more difficult and involved.

pry

pry is a REPL (Read, Evaluate, Print, Loop), meaning you can play around with code and see the result immediately. Some common pry usage:

- `load "filename.rb"` → loads the file into pry, and now you can call all methods in that file
- `ls` → allows you to see methods for an object (ex: `ls String` will show you all of the String instance methods)
- `show-doc` → shows you specifics for a method (ex: `show-doc String#include?` will show you documentation on the String's `include?` method)
- `show-source` → shows the source code for a method (ex: `show-source my_method` will show you the implementation of `my_method`)

byebug

byebug is a wonderful tool for assisting with the debugging process. It allows us to stop our code, and slowly run through it to see how things are changing. To use byebug:

- `require "byebug"` → needed at the top of a file you want to use byebug in
- `debugger` → write this in your code; when your program is run and hits the line with debugger, byebug will take over and pause the program's execution

Common byebug usage while in debugger:

- `next` or `n` → takes you to the *next* line of code that will run according to control flow (it respects if statements, loops, etc if the condition is met)
- `step` or `s` → *steps* inside of the method (if the line about to be evaluated has a call to `is_prime?(11)`, step will take you to the first line in the `is_prime?` method)
- `continue` or `c` → *continues* normal execution until it hits a `debugger` or breakpoint
- `list=` → shows you where the debugger is currently at
- `break <line_number>` or `b <line_number>` → adds a breakpoint; if the code hits this line in the future, the debugger will stop there
- `display <variable>` → tracks this variable and it's value as you progress through your code in the debugger
- `l <start_line>-<end_line>` → lists the lines from start_line to end_line
- `quit` or `exit` → stops the debugger and execution of your program

You can also evaluate code by simply typing it in; if you are curious what the variable `my_cat` is, but you don't want to track it, you can type `my_cat.`

Errors

Reading errors is a skill that is extremely important, and requires practice. Always check out the *stack-trace*: this is the message that shows the error type, along with the files and line numbers for the methods that were called right before this error happened. Common errors:

- `SyntaxError`: you have a syntax error, commonly an extra `end` or missing `end`, and you'll have to do some investigation across the file to solve this
- `NameError`: you are trying to reference a variable or method that hasn't been defined (usually a spelling mistake)
- `NoMethodError`: you are trying to call a method on an object that doesn't exist, like `"abc".banana` (Strings don't have a

banana method)
- `ArgumentError`: you gave method too few or too many arguments (Ruby requires that you give an exact num of arguments, excluding optional arguments)

TDD

- TDD (test driven development) is a widely used strategy among software engineering teams. The idea is to write your test (to describe how something should work/behave), run your tests (and make sure the newly written test fails), change/add code to satisfy the tests, and make sure all the tests (old and new) pass with the updated code.
- Tests will keep your codebase accountable for adhering to expected behavior; without tests, there is chaos. If a company offers you a job but they don't test their code, *you might want to reconsider.*

RSpec

- RSpec is a popular library for testing Ruby code, and will be the library we will be using to test code at App Academy. For now, you won't be writing any RSpec tests, but it is important to know how to read them. If you look at a spec file, you will find the following keywords:
    - `describe`: describes the method you want to test
    - `context`: (optional) describes a certain situation that may arise
    - `it`: describes the behavior you are testing
    - `expect`: shows how this behavior is actually being tested (Rspec will run the method in `expect`, and check to see if it's equal to (`eq`) the result specified)
- You can run RSpec tests by:
    1. navigating to the directory (folder) containing your code and the tests (you should have a lib and spec folder here).
    2. `bundle install` (you only need to do this once in a project – it installs the *gems* listed (like `rspec`) from a file called *Gemfile* so you can use them in your project).
    3. Then run `bundle exec rspec` (run the version of `rspec` that the Gemfile specifies) to run all the tests in the suite.
    4. (You should see a summary of failures and successes in your terminal, along with details of the failures).

When in doubt, pry it out. – Anonymous Instructor

Blocks & Procs

What's up, proc? We've also entered the world of blocks and procs!

- *Blocks* are chunks of code that are often given as arguments to methods.
- `[1, 2, 3].each { |el| puts el }` the bolded text is the block given to the `each` method.
    - You can also have a multi-line block by using `do` and `end`.

So what's a proc?

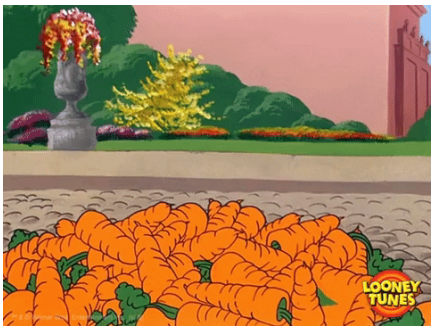
- *Procs* are blocks that *are objects*, meaning:
    - you can create a proc:
        - `Proc.new { |el| puts el }`
    - you can assign a variable to a proc:
        - `printer = Proc.new { |el| puts el }`
    - you can call methods on the proc:
        - `printer.call(10)` → this calls the Proc `call` method (Procs have a method named *call*), meaning it's going to execute the block with the argument of 10, so it should `puts 10`.

Why is this useful?

- We can define our own methods to take in blocks as arguments, like the `Array#each` method!
- When defining the method to take in a block, you can list the parameters, with the last named something like `&prc`
    - The & converts a block given to a proc
    - We can then call `prc.call` with any arguments that should be given to the block to execute the block!

- We can then call `proc.call` with any arguments that should be given to the block to execute the block!

Pretty neat!



Y'all can do it!

-Lizbot