Motivation

This week we took a look at *Test Driven Development*. We walked through the steps on how to properly conduct TDD. The goal this week is to practice TDD while working to implement a specification.

Course Learning Outcome(s):

- **Apply** testing techniques, including black-box and white-box techniques, automatic testing activities, and regression testing (CLO 4)

Module Learning Outcome(s):

- **Apply** test driven development techniques

Description

For this assignment you will utilize TDD to implement the following specification:

`check_pwd` accepts a `string` and returns `True` if it meets the criteria listed below, otherwise it returns `False`:

- Must be between 8 and 20 characters (inclusive)
- Must contain at least one lowercase letter
- Must contain at least one uppercase letter
- Must contain at least one digit
- Must contain at least one symbol from: `~`!@#$%^&*()_+-=` (copy and paste to avoid missing characters)

You may assume that only strings will be sent to the `check_pwd`.

This should look familiar, it is the specification we used in Week 5's Random Testing Exploration. The difference is that you now have to use TDD to create this function.

You may be thinking, "Couldn't we just copy what you wrote in the examples?" The answer is no. First, it doesn't work 100% correctly. Second, for this assignment you will have to *show your work*.

To accomplish this, you will be using a **private** GitHub repo to maintain a change log for each step of the TDD process. I have provided the TDD steps below in case you forgot.

TDD Steps

1. Write a test
2. Run all currently written tests
    - If the tests all pass, return to Step 1
    - If a test fails, proceed to Step 3
3. Write the bare minimum of code to make the test pass
4. Run all the currently written tests
    - If tests all pass, return to Step 1
    - If the failing test is still failing, return to Step 3
5. Occasionally evaluate if the code can be refactored to reduce duplication or eliminate no longer used parts of the code
6. Eventually stop development after adding "enough" tests without triggering a new failure

You are required to follow these steps when writing your code. You will need to maintain a file with the code you are writing and a file that houses your tests. If you write large chunks of the implementation at once or more than a single test at a time, you will lose points. You will create a new commit each time you do the following:

1. Write a test
2. Implement code to cause a failing test to pass

Each commit requires a message describing the changes. The message for each type of commit is as follows:

1. Name of the test you wrote and a brief description of what part of the specification you are testing
2. Briefly describe the changes you made to the code and if it caused the test to pass

Sample Workflow

1. Read specification
2. Write first test and run it to ensure it fails
3. Commit changes, "Wrote test1 to see if an empty string is rejected correctly"
4. Write the bare minimum implementation to get the test to pass
5. Run test to see if it passes
6. Commit changes, "Made check_pwd return False no matter what, it passed"
7. Write second test...

Please note, you need to commit your changes after writing a test, but BEFORE you write the code to make it pass. You then need to commit your changes BEFORE writing another test. So, write test and commit, then write implementation and commit.

It is very important you follow these steps carefully as we will be checking your commit history to see that you adhered to the TDD process.

In order to submit this assignment, you will need to host your work on a private GitHub repo. You will also need to ensure that you have `coeCS362` added as a collaborator on the repo. Failure to do this will result in your assignment not receiving a grade.

Hints

- Do not immediately jump to writing all inclusive tests. Follow the examples in the exploration for how small your changes should be. Do not write the entire specification in one pass, do it in pieces
- Once written, a test should never be changed. So if you have a test that passes but then starts failing once you write more code, then something is wrong with your approach, reach out for help
- It is a good idea to start with simply `return True` in your `check_pwd` and then your tests should `assert_False`
- Go slow and ensure you are committing BOTH after writing a test and the changes you made to pass said test
- Make your commit messages descriptive enough for the grader to understand or you risk losing points
- Don't fake this process or you won't gain the needed experience with TDD
- After you feel you have implemented the specifications, you can reassure yourself about your code by running some random testing before your final commit similar to in Random Testing's exploration (we will! HINT HINT)

What to turn in

- You will supply the link to the **private** GitHub repo with `coeCS362` as a collaborator
- Your repo must have two files:
    - `check_pwd.py` that contains your password checking function (`check_pwd()`)
    - `tests.py` that contains the tests you wrote during the TDD process (with a main function)
- You will also need to turn in your files to Gradescope so we can run them against a set of Instructor tests (see below)