Introduction

This week we discussed how *random testing* can help find pesky bugs that other approaches are likely to miss. The goal of this assignment is two fold:

1. Practice writing random tests
2. Practice trying to diagnose the cause of the discovered bugs

You will write a test suite that uses random testing to try to find bugs, but more importantly you will need to theorize why each bug is triggered.

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** random testing techniques

Description

If you recall, random testing is a form of black box testing, so you will have to write tests based on a specification. Luckily, we are going to reuse the same specification from Week 3's assignment. You will be testing, again, `credit_card_validator`. The specification is reproduced below.

You will test a function called `credit_card_validator` that is passed a sequence of digits as a `string` that represents as credit card number. This function will return `True` if it is a valid credit card number, otherwise it will return `False`.

Depending on the credit card issuer, the length of a credit card number can range between 10 and 19 digits. The first few digits of the number are the *issuer prefix*. Each credit card issuer has an assigned range of numbers. For example, only Visa credit card numbers may begin with `4`, while American Express card numbers must begin with either a `34` or `37`. Sometimes, credit card providers are assigned multiple ranges. For example, MasterCard card numbers must start with the numbers between `51` through `55` or `2221` through `2720` (inclusive).

The last digit of the number is referred to as the check digit and acts as a checksum. Most credit cards calculate this check digit using the Luhn algorithm (see resources below for how this is calculated).

In order to limit the scope of this assignment, we are going to limit the number of credit card issuers to 3: Visa, MasterCard, and American Express. Each has their own prefixes and length requirements.

- **Visa**
    - Prefix(es): 4
    - Length: 16
- **MasterCard**
    - Prefix(es): 51 through 55 and 2221 through 2720
    - Length: 16
- **American Express**
    - Prefix(es): 34 and 37
    - Length: 15

Your task is to create a series of tests that attempt to reveal **bugs** in the implementation. As random testing is a form of black box testing, you will not have access to the source.

You will be submitting your code to Gradescope which will autograde your tests. In order to get full credit on the assignment, you will need to locate *all* **5** bugs in the code (refer to the rubric for full details).

Considering that our tests will stop running as soon as one of it's asserts fails, for this assignment please DO NOT use asserts. It is sufficient to just call `credit_card_validator` with your test cases, Gradescope will still catch the bugs. We will just assume that your tests have the correct assert.

This assignment is a bit different than Week 3's. There you just submitted a testing suite on Gradescope, this week you will *also* be submitting a PDF where you try to theorize as to the cause of each of the 5 bugs. Please use the supplied Word document

Download supplied Word document

to format your PDF. Below is an example of how to fill out the document. Notice you will need to provide *evidence* and a *theory* for each bug.

- **Bug 1**
    - **Triggering credit card numbers (at least 5)**
        - 123456789
        - 133549798
        - 187456315

        - 154897463
        - 123545668
    - **Theory that explains what triggered the bug**
        - This bug is caused by credit card numbers that begin with `1`

It is important that your theory *fits* your supplied triggering numbers to receive any credit. For example, a theory of "This bug is caused by credit card numbers that begin with `12`" would not fit the supplied evidence. To receive full credit for each bug, your theory must match the actual coded error in the source. This means you would be wise to gather as much evidence as possible for each bug to ensure your theory holds true across all triggering numbers. It is important to be as specific as possible about the patterns you observe in your theories (i.e. prefixes, length, checkbit, and etc.). Also, please do not pad your PDF with numbers that don't trigger a given bug , we will be verifying.

Finally, your test suite needs to be free of linting errors using the PEP8 standard; this will be important later when working on shared repositories. If you are unfamiliar with linting, please see the resources below. The easiest way to accomplish this is to ensure that there are no "squiggly" lines under your code in PyCharm. You can also use the PEP8 Online tool below to copy and paste your code to verify it has no errors.

Please see the Rubric for the exact point breakdown and deductions.

Do Tests Have to be Random?

It may seem odd to state this, but your tests MUST be random. You may not have tests that have hardcoded credit card numbers in them. Also, you need to make sure your tests are random *enough*. To put this in a quantifiable way, the longest prefix listed above is 4 digits so you may not have more than 4 digits of a number hardcoded into any given random test generator for a prefix. Failure to have random tests will result in a 50% deduction to your score.

Finally, your tests need to catch all the bugs during a single run. We will be rerunning your latest submission to grade and you will only receive points for those bugs that are triggered.

Developing solid theories

For this assignment you are being asked to develop theories that explain what all the triggering test cases have in common for each bug. For HW1: Black Box Testing, it was fairly easy to identify these theories if you used a Partition Testing approach: Bug 1 is triggered by credit card numbers that begin with a MasterCard prefix, are 16 digits long, and have a valid check digit (not the actual bug in the assignment).

This time around, things are so obvious, because these bugs seem to appear at random (see what I did there?). It is important to gather as many examples (not just the 5 in the PDF) to have a solid data set to base our theories on. If you look at the Rubric below, you will see that to receive full points on this assignment you need to not only develop a theory that *fits* the test cases in the PDF, but matches the actual cause of each bug (thus the need for as much data points as possible).

To help get you into the correct state of mind to theorize effectively, please review the following pointers.

1. You should view these numbers as credit card numbers, just like you did in HW1
2. After you have exhausted #1, stop thinking of them as credit card numbers
3. What do all the triggering test cases have in common when it comes to being actual integers?
4. What do all the triggering test cases have in the common when it comes to patterns that appear *within* the numbers?

Hints

- Finding the bugs is fairly trivial and could be done with just a single test case, but this risks running over the time limit
- Each time you trigger a bug, Gradescope will print the triggering numbers in the autograde report
- The main focus of this assignment is to diagnose the bugs. To do this, you will require many data points. Some of the bugs are less common than others. It is OK to rerun your tests on Gradescope multiple times to gather these triggering numbers
- You may run over the time limit while gathering data, but just make sure your tests run within the time limit in your final submission to Gradescope

What to turn in

- Submit to Gradescope your testing suite; it must be named `tests.py`
- Submit to Canvas your PDF following the supplied template
- 'tests.py' must be free of PEP8 linting errors
- Tests must run within 20 seconds on Gradescope. See Gradescope results for your test runtime.