TDM 30100: Project 5 — 2022

Motivation: Code, especially newly written code, is refactored, updated, and improved frequently. It is for these reasons that testing code is imperative. Testing code is a good way to ensure that code is working as intended. When a change is made to code, you can run a suite a tests, and feel confident (or at least more confident) that the changes you made are not introducing new bugs. While methods of programming like TDD (test-driven development) are popular in some circles, and unpopular in others, what is agreed upon is that writing good tests is a useful skill and a good habit to have.

Context: This is the first of a series of two projects that explore writing unit tests, and doc tests. In The Data Mine, we will focus on using `pytest`, doc tests, and `mypy`, while writing code to manipulate and work with data.

Scope: Python, testing, pytest, mypy, doc tests

Learning Objectives

- Write and run unit tests using `pytest`.
- Include and run doc tests in your docstrings, using `pytest`.
- Gain familiarity with `mypy`, and explain why static type checking can be useful.
- Comprehend what a function is, and the components of a function in Python.

Make sure to read about, and use the template found here, and the important information about projects submissions here.

Dataset(s)

The following questions will use the following dataset(s):

- `/anvil/projects/tdm/data/goodreads/goodreads_book_authors.json`
- `/anvil/projects/tdm/data/goodreads/goodreads_book_series.json`
- `/anvil/projects/tdm/data/goodreads/goodreads_books.json`
- `/anvil/projects/tdm/data/goodreads/goodreads_reviews_dedup.json`

Questions

Question 1

There are a variety of different testing packages: `doctest`, `unittest`, `nose`, `pytest`, etc. In addition, you can write actual tests, or even include tests in your documentation!

For the sake of simplicity, we will stick to using two packages: `pytest` and `mypy`.

Create a new working directory in your `$HOME` directory.

```
mkdir $HOME/project05
```

Copy the following, provided Python module to your working directory.

```
cp /anvil/projects/tdm/data/goodreads/goodreads.py $HOME/project05
```

Look at the module. Use `pytest` to run the doctests in the module.

TIP

See here for instructions on how to run the doctests using `pytest`.

NOTE

One of the tests will fail. This is okay! We will take care of that later.

NOTE

Run the doctests from within a `bash` cell, so the output shows in the Jupyter Notebook.

Items to submit

- Code used to solve this problem.
- Output from running the code.

Question 2

One of the doctests failed. Why? Go ahead and fix it so the test passes.

WARNING

This does *not* mean modifiy the test itself — the test is written exactly as intended. Fix the *code* to handle that scenario.

Items to submit

- Code used to solve this problem.
- Output from running the code.

Question 3

Add 1 more doctest to `split_json_to_n_parts`, 3 to `get_book_with_isbn`, and 2 more to `get_books_by_author_name`. In a bash cell, re-run your tests, and make sure they all pass.

Items to submit

- Code used to solve this problem.
- Output from running the code.

Question 4

Doctests are great, but a bit clunky. It is likely better to have 1 or 2 doctests for a function that documents *how* to use the function with a concrete example, rather than putting all your tests as doctests. Think of doctests more along the lines of documenting usage, and as a bonus you get a couple extra tests to run.

For example, the first `split_json_to_n_parts` doctest, would be much better suited as a unit test, so it doesn't crowd the readability of the docstring. Create a `test_goodreads.py` module in the same directory as your `goodreads.py` module. Move the first doctest from `split_json_to_n_parts` into a `pytest` unit test.

In a bash cell, run the following in order to make sure the test passes.

```
%%bash

cd ~/project05
python3 -m pytest
```

Items to submit

- Code used to solve this problem.
- Output from running the code.

Question 5

Include your `scrape_image_from_url` function from the previous project in your `goodreads.py`. Write at least 1 doctest and at least 1 unit test for this function. Make sure the tests pass. Run the tests from a bash cell so the graders can see the output.

NOTE

For this question, it is okay if the doctest and unit test test the same thing. This is all just for practice.

WARNING

Make sure you submit the following files:

- the `.ipynb` notebook with all cells executed and output displayed (including the output of the tests).
- the `goodreads.py` file containing all of your code.
- the `test_goodreads.py` file containing all of your unit tests (should be 2 unit tests total).

Items to submit

- Code used to solve this problem.
- Output from running the code.

WARNING

Please make sure to double check that your submission is complete, and contains all of your code and output before submitting. If you are on a spotty internet connection, it is recommended to download your submission after submitting it to make sure what you *think* you submitted, was what you *actually* submitted.

In addition, please review our submission guidelines before submitting your project.