

# Professional Practice Assignment 1 --- Intro to Unit Testing & T/BDD

By: Kyle Bassignani and Daniel Tymecki

**Due: Tuesday, September 11th**

## Project Layout and Organization Conventions

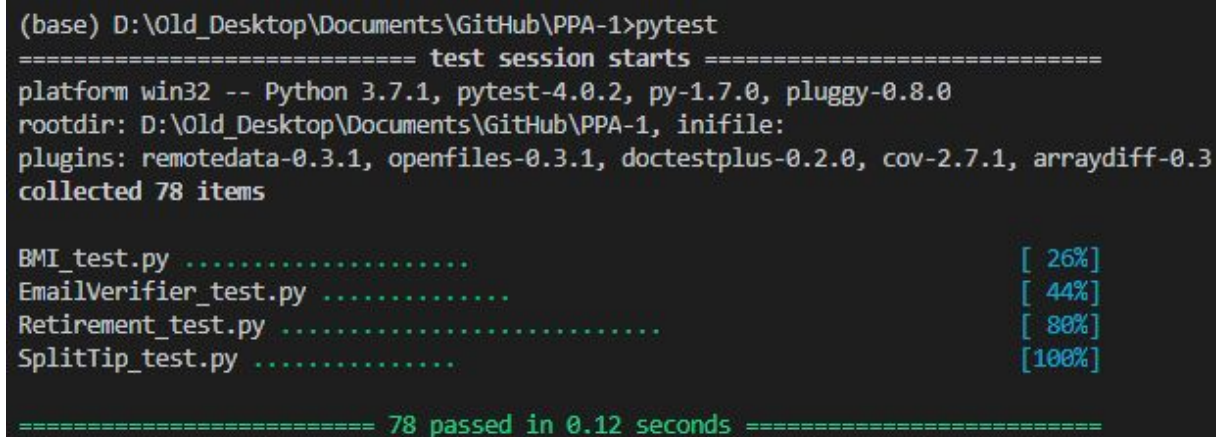
This project was built using Python 3.7.1. Our unit testing framework leveraged pytest version 4.0.2, the current industry standard for Python testing.

In terms of organization, our functionality and our tests are divided into separate modules. The functionality module is named according to the function it defines (e.g. BMI.py). The testing module is defined using the name of the functionality it tests, with an additional “\_test” suffix (e.g. BMI\_test.py). Within these testing modules, each unit test is named descriptively by exactly what it tests with a distinctive “test\_” prefix. For example, BMI\_test.py has a unit test that checks the function returns a str variable. The naming convention is as follows test\_BMI\_returns\_string\_argument(). The name of the function clearly defines what the test check for.

## Setup and Execution Instructions

See the [README](#) on the GitHub Repo for instructions.

## Testing Evidence



```
(base) D:\Old_Desktop\Documents\GitHub\PPA-1>pytest
===== test session starts =====
platform win32 -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: D:\Old_Desktop\Documents\GitHub\PPA-1, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, doctestplus-0.2.0, cov-2.7.1, arraydiff-0.3
collected 78 items

BMI_test.py ..... [ 26%]
EmailVerifier_test.py ..... [ 44%]
Retirement_test.py ..... [ 80%]
SplitTip_test.py ..... [100%]

===== 78 passed in 0.12 seconds =====
```

*Image 1: Depicts all tests passing via our unit testing framework - pytest 4.0.2*

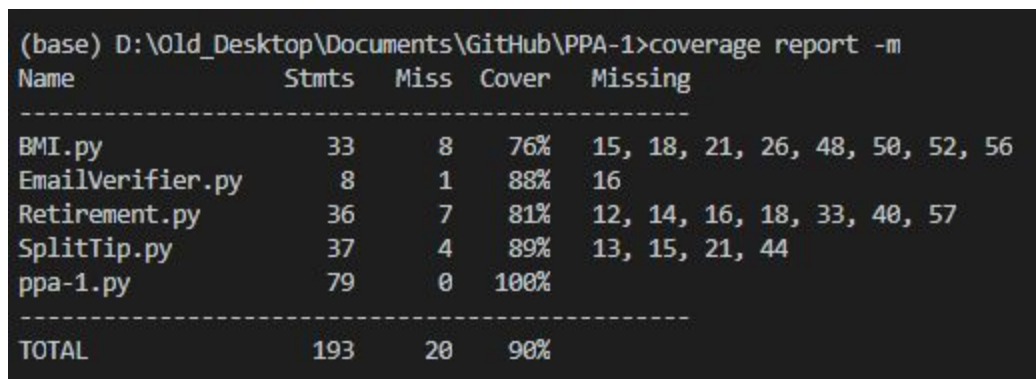
## Testing Coverage

To test our coverage, we utilized the coverage.py package add on to analyze our code and summarize the testability of our application. One caveat to this was that in order to count as covered code, the test needed to be run during the execution of the program. This meant that we purposely had to fail every failsafe for it to appear tested on our report. This explains why this coverage report is slightly inaccurate, as our tests covered all facets of our function, but we could not fail every condition during the execution. We left some type checking in our program,

so that the code would be safe for different uses, but in the case of this program there was input checking on the front end to ensure the user enters correct values.

To generate this report, run the following commands:

1. `coverage run ppa-1.py`
  - a. Here you have to run through the program, running all the functions and failing every condition possible to generate a report like the one depicted below. Results may vary depending on how much of the program is run.
2. `coverage report -m`
  - a. This generates a report based on the coverage generated by the preceding command. The `-m` flag is used to show the lines when tests appear to be missing.



Name	StmtS	Miss	Cover	Missing
BMI.py	33	8	76%	15, 18, 21, 26, 48, 50, 52, 56
EmailVerifier.py	8	1	88%	16
Retirement.py	36	7	81%	12, 14, 16, 18, 33, 40, 57
SplitTip.py	37	4	89%	13, 15, 21, 44
ppa-1.py	79	0	100%	
TOTAL	193	20	90%	

Image 2: Depicts the coverage of our program through the use of coverage.py API

## Red Green Demo Video

<https://youtu.be/VdGb4ALTDmE>

## Application Demo Video

<https://youtu.be/6v68m0qQqTc>

## TDD Reflection - Daniel

After completing this project, I gained a newfound appreciation for writing unit tests and using TDD. I have used TDD a little bit in the past, but never for a full project. I used it more to write tests for complex systems rather than tests in simplistic contexts. I believe writing tests in this manner is extremely useful, but not for basic tasks. For some methods in this project, it seemed

like overkill to use TDD because they were so basic. In this case, I hated writing tests because it seemed too tedious to be useful. I would have rather spent time working on more functionality than write seeming feckless tests. Due to the simple nature of it, I often found myself frustrated while working, which caused a few mistakes while writing tests. But because I strictly followed TDD, these mistakes came to light rather quickly and were henceforth corrected. On the flip side, other functions like the email verified had definite merit for TDD utilization, as something more complex than simple calculations needs a lot of tests to ensure it has the desired behavior. Nevertheless, unit tests on their own seem like a great idea, especially when working in teams. It was great to be able to verify code changes made by my partner simply by running tests. And self documenting code was a wonderful innovation. Overall, while I may not have enjoyed writing tests using TDD, I definitely see the merit and I have been inspired by this project to write more unit tests for my code moving forward.

## TDD Reflection - Kyle

This is my first experience with TDD; overall I had a good experience with the practice. I do believe using some aspects of TDD can be very beneficial to a development project. I do think for the programs we were writing for this project TDD was a bit overkill. I understand that the project was for learning, but I did struggle to break down my code to small enough chunks to test effectively. For what I was able to break down, some of the functions I wrote were overly simple. However; this project did show me that TDD would be very beneficial for larger projects. The biggest benefit of TDD I noticed was that when I went to make a bug fix, I would not have to spend large amounts of time retesting my program to ensure that nothing broke during the fix. The biggest draw back I noticed during this project was that some of the extra work required for the tests seemed excessive. I think for many of the functions having one test for all the business logic would have sufficed, but I do understand that in the real world the logic is more complicated and in those cases it would be good to break up the code into smaller testable chunks.