QUEENSBOROUGH COMMUNITY COLLEGE
The City University of New York
Department of Engineering Technology

Project: Unit Tests

Overview:

In this project, you will learn how to write unit tests for a Python program using the `'unittest'` module. The primary focus is on testing a program that calculates the average grade of a given number of students. You will use mocking techniques to simulate user input and capture printed output. By the end of the project, you should be able to write and understand basic unit tests and extend them to cover additional cases.

Description:

Program to be Tested: `'main_program.py'`

The `'main'` function is designed to calculate the average grade for a specified number of students. Here's a breakdown of its functionality:

1. Prompt for Number of Students:

   - The program repeatedly prompts the user to enter the number of students until a positive integer is entered.

   - If the input is not a positive integer, the program prints an error message and asks again.

2. Collect Grades:

   - For each student, the program prompts the user to enter a grade between 0 and 100.

   - If the input is not within the valid range, the program prints an error message and asks again.

3. Calculate Average:

   - The program calculates the average of the entered grades and prints the result.

Unit Test Program: `'test_main_program.py'`

This unit test program uses the `'unittest'` module along with `'unittest.mock'` to simulate user inputs and capture printed outputs. Here's a breakdown of the test cases:

1. Test Case: `'test_main_valid_input'`:

   - Simulates valid inputs for the number of students and grades.

   - Verifies that the program calculates the correct average and prints the expected output.

2. Test Case: `'test_invalid_number_of_students'`:

   - Simulates an invalid input for the number of students followed by a valid input.

   - Verifies that the program eventually accepts the valid input and calculates the correct average.

3. Test Case: `'test_invalid_grades':`

- Simulates invalid grades followed by valid grades.

- Verifies that the program eventually accepts the valid grades and calculates the correct average.

Explanation of Each Test Case

1. `'test_main_valid_input':`

- Uses `@patch` to mock `input` and `print` functions.

- Simulates the following user inputs: `3` (number of students), `85`, `90`, and `95` (grades).

- Checks if the output `'The class average is: 90.00'` is printed.

2. `'test_invalid_number_of_students':`

- Simulates an invalid input (`0`) followed by a valid input (`3` for number of students), then grades `85`, `90`, and `95`.

- Verifies that the program handles the invalid input and eventually prints the correct average.

3. `'test_invalid_grades':`

- Simulates invalid grades (`105` and `-5`) followed by valid grades (`85`, `90`, and `95`).

- Verifies that the program handles the invalid grades and eventually prints the correct average.

Instructions for Implementing Additional Test Cases

To implement additional test cases, follow these steps:

1. Identify New Scenarios:

- Think about different ways users might interact with the program. Consider edge cases and unusual inputs.

- Examples: Entering negative numbers for grades, entering non-numeric characters, entering a very large number of students, etc.

2. Create New Test Methods:

- Use the given samples and create new testing methods. Make sure to give each testing method a meaningful name. Each testing method name starts with “test_”.

3. Run and Verify:

- Run the testing program and check if all results are coming out correct.

Example of an Additional Test Case

Here's an example of an additional test case:

```
@patch('builtins.input', side_effect=['2', '-10', '110', '75', '85'])
@patch('builtins.print')
def test_out_of_range_grades(self, mock_print, mock_input):
    main_program.main()
    mock_print.assert_called_with('The class average is: 80.00')
    self.assertIn(
        unittest.mock.call('The class average is: 80.00'),
        mock_print.mock_calls
    )
```

This test case simulates entering out-of-range grades before providing valid ones. It ensures the program correctly calculates and prints the average of the valid grades.

Assignment: Adding Three Different Test Cases

In this assignment, you will extend the test coverage for the `main_program.py` by adding **three** different testing cases to the `test_main_program.py`. These test cases should cover various scenarios that the existing tests do not address. The objective is to ensure that the main function handles a wide range of inputs correctly and robustly.

Test Case Assignment 1: Handling Non-Numeric Input for Number of Students

Verify that the program handles non-numeric input for the number of students and prompts the user again until a valid input is provided. Simulate non-numeric input ('abc') followed by a valid numeric input ('3'). Provide valid grades for the students. Use the example above and the existing testing cases as the reference to add the new testing case in `test_main_program.py`.

Test Case Assignment 2: Handling Non-Numeric Input for Grades of Students

Verify that the program handles non-numeric input for grades and prompts the user again until a valid input is provided. Simulate a valid number of students ('3'). Provide non-numeric input for grades ('abc') followed by valid numeric grades. Add the new testing case in `test_main_program.py`.

Test Case Assignment 3: Handling a Single Student

Verify that the program correctly calculates and prints the average grade when there is only one student. Simulate a valid input for the number of students ('1'). Provide a valid grade for the single student. Add the new testing case in `test_main_program.py`.

Submission Requirements:

You must submit your project with a pdf file and two source code files:

PDF Submission:

- Create a PDF document with screenshots demonstrating the steps of your project.
- Include the following:
    - Screenshots of your source files of `main_program.py` and `test_main_program.py`.
    - Screenshots of running the unittest tests and the results displayed in the terminal.
    - Along with each screenshot, add brief descriptions for each testing case you have added explaining what is being done.
    - Ensure the PDF is well-organized.

Source Code:

- Include both `main_program.py` and `test_main_program.py`.
- Ensure the code is well-commented.
- Submit the code files in a ZIP file.