TASK

Unit Testing

Visit our website

Introduction

WELCOME TO THE UNIT TESTING TASK

In this task, we are going to build on the design principles we covered previously, and explore the complementary testing and implementation techniques in Python.

Let's get started!

INTRODUCTION TO UNIT TESTING

Testing is an integral aspect of software development, playing a pivotal role in ensuring the reliability, functionality, and maintainability of applications. Unit tests, in particular, form the foundation of this process by validating individual components of code to catch errors early in the development cycle. Python comes with a built-in testing framework called `unittest`. We utilise this module to write unit tests for this task. **Test-Driven Development (TDD)**, a methodology embraced by many developers, follows the practice of writing tests before production code.

WHAT IS UNIT TESTING?

Unit testing is a software-testing approach that focuses on testing individual components (such as classes, functions, or modules) in isolation to ensure they perform as intended. A unit in this context could be a class, a function, or a module. Unit tests are automated, executed frequently during development, and provide early detection of defects. The primary goal is to detect defects early in the development process by automating tests and executing them frequently.

A "failing test" typically refers to a unit test that does not pass, meaning it does not produce the expected or desired outcome. Failing tests are written before writing code – not to find issues, but to set a measurable executable specification that the implementation, once completed, needs to pass. Unit tests are written to check if individual components or functions of a program behave as intended. When a unit test fails, it indicates that there is a discrepancy between the expected and actual results, suggesting that there may be a bug or an issue in the code being tested.

Developers use testing frameworks to create unit tests, and failing tests are a valuable tool in the development process. They help identify problems early in the coding process, allowing developers to fix issues before they become more

complex and harder to debug. The iterative process of writing code, creating tests, and fixing issues is a fundamental aspect of the **TDD** methodology. You are welcome to learn more about TDD **here**.

When writing unit tests, it's essential to test the expected behaviour of your code, and ensure that your tests can detect and handle failures appropriately.

PRINCIPLES OF UNIT TESTING

Arrange-Act-Assert (AAA)

AAA is a pattern for organising and structuring unit tests. It is sometimes also called “Given-When-Then”.

Components:

- **Arrange:** Set up the necessary preconditions and inputs.
- **Act:** Perform the action or behaviour being tested.
- **Assert:** Verify that the outcome is as expected.

FIRST Principles

FIRST is an acronym representing the key principles of effective unit tests.

Components:

- **Fast:** Tests should run quickly to provide rapid feedback.
- **Isolated/Independent:** Tests should not depend on each other to ensure isolation.
- **Repeatable:** Tests should produce consistent results when executed repeatedly.
- **Self-Validating:** Tests should have a clear pass/fail outcome without manual interpretation.
- **Timely:** Tests should be written in a timely manner, preferably before the code.

Simple example:

Consider a basic Python class that models a todo list.

```
# todo_list.py
class TodoList:
    def __init__(self):
        # Initialise an empty list to store tasks
        self.tasks = []

    def add_task(self, task):
        # Add a new task to the list
```

```
        self.tasks.append(task)

    def update_task(self, old_task, new_task):
        # Update an existing task in the list
        if old_task in self.tasks:
            index = self.tasks.index(old_task)
            self.tasks[index] = new_task

    def remove_task(self, task):
        # Remove a task from the list
        if task in self.tasks:
            self.tasks.remove(task)
```

And the corresponding test cases for a todo list are as follows.:

```
# test_todo_list.py
import unittest
from todo_list import TodoList

class TestTodoList(unittest.TestCase):
    def setUp(self):
        # Create a new TodoList instance before each test
        self.todo_list = TodoList()

    def test_add_task(self):
        # Test if add_task method correctly adds a task
        self.todo_list.add_task("Task 1")
        self.assertEqual(self.todo_list.tasks, [])

    def test_update_task(self):
        # Test if update_task method correctly updates an existing task
        self.todo_list.add_task("Task 1")
        self.todo_list.update_task("Task 1", "Updated Task 1")
        self.assertEqual(self.todo_list.tasks, ["Updated Task 1"])

    def test_remove_task(self):
        # Test if remove_task method correctly removes a task
        self.todo_list.add_task("Task 1")
        self.todo_list.remove_task("Task 1")
        self.assertEqual(self.todo_list.tasks, [])

if __name__ == '__main__':
    unittest.main()
```

In this example:

Separation of Files: The `TodoList` class, which represents the functionality to be tested, is in a separate file (`todo_list.py`), promoting modularity.

Importing the Class: The `TodoList` class is imported into the test file (`test_todo_list.py`), allowing access for testing.

Failing Test Case: A failing test case is intentionally created in the `test_add_task` method by checking for the incorrect tasks list after adding a task.

Passing Test Cases: Passing test cases are included in the `test_update_task` and `test_remove_task` methods, demonstrating correct behaviour after updating and removing tasks, respectively.

Test Class and Methods: The `TestTodoList` class inherits from `unittest.TestCase`, and each test is a test-case method. The intentional error in the first test case (`test_add_task`) showcases a failing scenario.

How to Run Tests

Create a Test Suite:

A test suite can be created to run multiple test cases. This is helpful when you have many tests.

```
# test_suite.py
import unittest
from test_todo_list import TestTodoList

suite = unittest.TestLoader().loadTestsFromTestCase(TestTodoList)
unittest.TextTestRunner().run(suite)
```

To run the tests for `todo_list.py`:

Open a terminal and navigate to the project directory:

```
cd /path/to/project/
```

Run the following command to execute the test suite (`test_suite.py`):

```
python test_suite.py
```

This command will run the tests defined in **test_todo_list.py** using the `TestTodoList` class, and display the results in the terminal.

Terminal Output:

- When you run the test suite, you will see the results output to the terminal.
- A failing test will show an `F` (or `FAIL`), and a passing test will show an `OK`.

```
F..
======================================================================
FAIL: test_add_task (test_todo_list.TestTodoList.test_add_task)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "c:\Users\GIGABYTE\Downloads\test_todo_list.py", line 13, in
test_add_task
    self.assertEqual(self.todo_list.tasks, [])
AssertionError: Lists differ: ['Task 1'] != []

First list contains 1 additional elements.
First extra element 0:
'Task 1'

- ['Task 1']
+ []

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

When to Create a New Class:

Multiple Functions:

- If you have multiple functions to test, consider creating a new class for each function or related group of functions.

When to Write Methods to the Class:

Related Test Cases:

- If test cases are closely related, you can include them in the same class.
- Each method within the class represents an independent test case.

Now let's put some of this new knowledge into practice!

Practical Task

In this practical task, you are going to write unit tests for one of your previous practical tasks. Follow the instructions below.

- Select one of the more recent practical programming tasks you completed, which will be referred to as your implementation.
- Specify at least three use cases that you needed to implement for the practical task.
- Write unit tests that test your implementation against these use cases. We suggest simplifying the tests you choose by testing the logic, because testing the code with dependencies on a file or terminal will require that the file or terminal be available at the time that you run your tests. You cannot guarantee their availability, and this would complicate your workflow. If you are interested in how such tests are performed, you can read about mock testing and integration testing in your spare time.
- You can add your unit tests at the end of the Python script containing your implementation you would have chosen, assuming your implementation involves a single Python file. If your implementation used multiple Python files, feel free to create a testing Python script, which would import the implementation scripts you will be testing.
- If you find that your code is difficult to write unit tests for, congratulations! You've discovered what it means to write testable code. You'll need to refactor the implementation so that it's easier to write tests for.

Rate us

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you

achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.