Hyperiondev

Task

Unit Testing

Visit our website

Introduction

WELCOME TO THE UNIT TESTING TASK

This task focuses on essential Python testing techniques, emphasising reliability and maintainability. By using the `unittest` framework and test-driven development (TDD) methodology, you will write tests before code, catching errors early. Tests follow the Arrange-Act-Assert (AAA) pattern and FIRST principles (fast, isolated, repeatable, self-validating, and timely), ensuring quick, independent, consistent, and clear results. This approach improves development efficiency and robustness.

UNIT TESTING AND TEST-DRIVEN DEVELOPMENT IN PYTHON

Unit tests form the foundation of this process by validating individual components of code to catch errors early in the development cycle. Python's built-in `unittest` framework is used for writing these tests. Test-driven development (TDD), widely embraced by developers, is a methodology where tests are written before production code, guiding the development process and ensuring robust and error-free implementations.

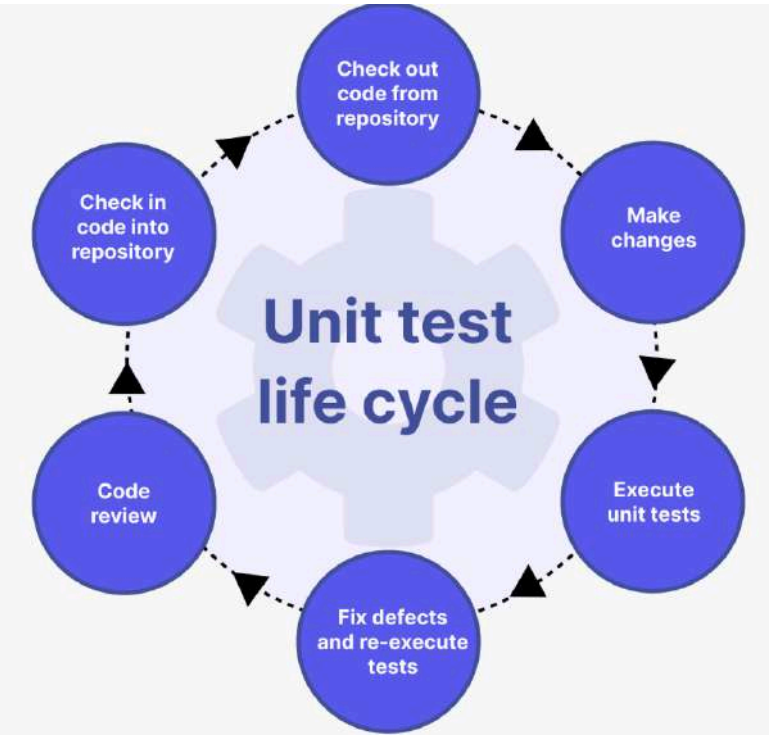
WHAT IS UNIT TESTING?

Unit testing is a software-testing approach that focuses on testing individual components 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 in **this video**.



The unit test life cycle (Kumar, 2023)

The unit test life cycle, illustrated in the image above, represents a systematic approach to software development that prioritises code quality and reliability. This cyclical process integrates testing directly into the development workflow, ensuring that each code change is verified before it's merged into the main codebase. By continuously validating code through unit tests, fixing issues promptly, and incorporating code reviews, developers can catch and address problems early, leading to more stable and maintainable software. This methodology supports **agile development practices** and helps teams deliver higher-quality code more efficiently.

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”.

Steps:

- **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.

Principles:

- **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.

Consider this example below as a basic Python class that models a to-do list. You may experiment with this code by creating a file named **todo_list.py** and copying the code into it.

```python
# 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)
```

The corresponding test cases for the `TodoList` class are as follows, which you can create in a file named **test_todo_list.py**:

```
# 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, we can make note of the following:

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

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 "path\to\file\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 that you've mastered unit testing in Python, you're equipped to catch bugs early and elevate code quality through TDD. By specifying code behaviour upfront, iterating with failing tests, coding to pass them, and refining with principles like

Arrange-Act-Assert and FIRST, you'll create more efficient, maintainable software. Unit testing isn't just about bugs; it empowers you to craft reliable code and ensure confidence in your software's evolution.

Take note

The task(s) below is/are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select “Request Review”, the task is automatically complete, you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you’ve done that, feel free to progress to the next task.

Auto-graded task

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

- Select one of the recent practical programming tasks you completed. This will be the focus of your unit tests. Consider the selected task your “implementation”.
- Identify at least three different scenarios (use cases) in which your implementation will be used. These use cases will guide the creation of your unit tests to ensure your implementation works correctly in different situations.

- Create tests for each use case. Focus on testing your implementation directly without external dependencies.
    - Single file: If your code is in one Python file, add the unit tests at the end of that same file.
    - Multiple files: If your code is spread across multiple files, create a separate Python file for the tests and import the files you need to test.
- If your code is difficult to test, then it means you likely need to refactor it so that it's easier to test. This is a normal and necessary part of writing good code and precise tests.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

Reference list

Kumar, R. (2023, October 18). *What is unit testing?* DevOpsSchool. **https://www.devopsschool.com/blog/what-is-unit-testing/**