Python

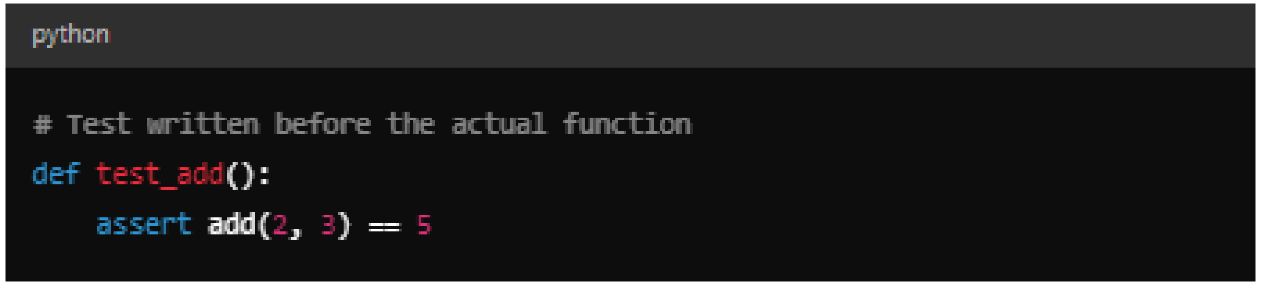
Full stack Skills Bootcamp

Step8Up Academy

Introducing Python TDD

What is Test-Driven Development (TDD)?

- TDD is a software development process where developers create tests before writing the code. This ensures that every piece of code written has a purpose and is tested.
- The idea is to start by defining the desired functionality through tests, which guide the development.

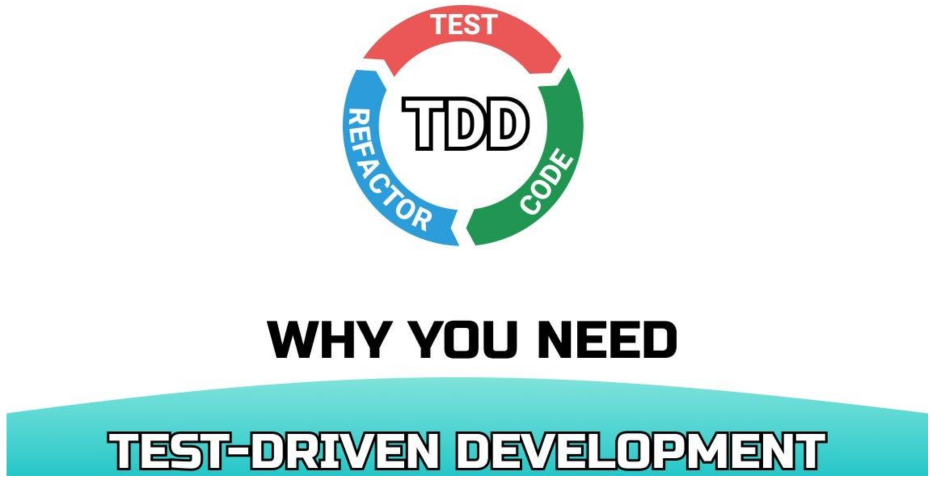


```python
# Test written before the actual function
def test_add():
    assert add(2, 3) == 5
```

Why Use TDD ?

TDD brings several advantages that help developers write better and more reliable code. Here's why it's widely adopted:

- **Ensures Code Quality**: Since tests are written first, developers are constantly checking that their code works as expected.
- **Reduces Bugs**: Early testing means fewer bugs during the later stages of development.
- **Simplifies Debugging**: If a bug is found, it's easier to identify because the tests are designed to highlight broken areas.
- **Encourages Minimal Code**: You write only enough code to make the test pass—nothing more, nothing less.



TDD Workflow: Red-Green-Refactor Cycle

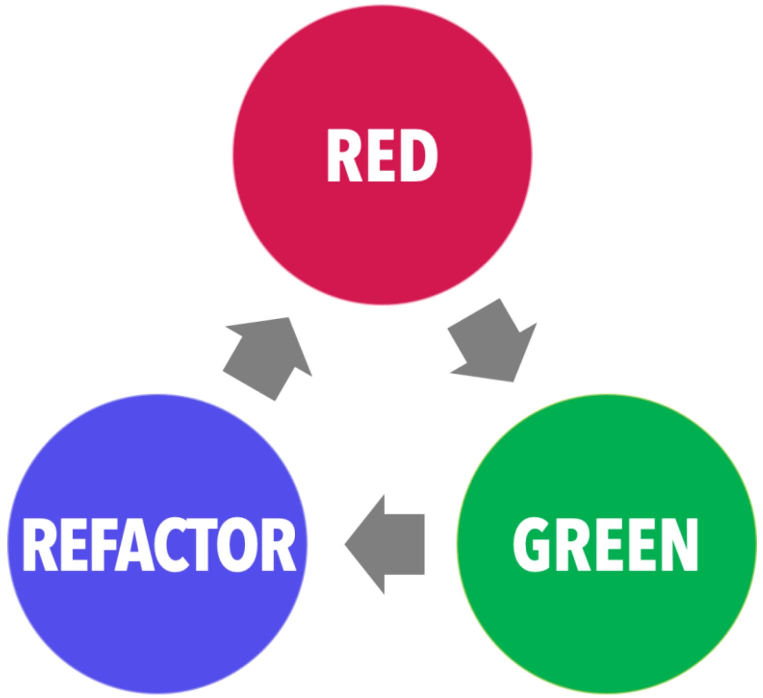
Overview

- The Red-Green-Refactor cycle is the heart of Test-Driven Development (TDD).
- It's a simple three-step process that you repeat for each feature or piece of functionality:

  Red: Write a test that fails (since you haven't written the code yet.

  Green: Write just enough code to make the test pass.

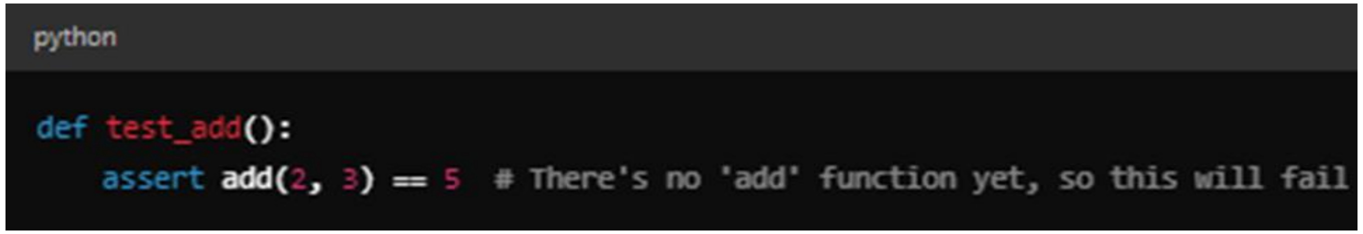
  Refactor: Clean up the code while keeping the tests passing.



Step 1

Red: Write a Failing Test:

- The first step is Red, meaning you write a test for functionality that doesn't exist yet.
- This test will fail because the code hasn't been written yet, but that's expected!
- Writing the test first forces you to think about what you want your code to do.



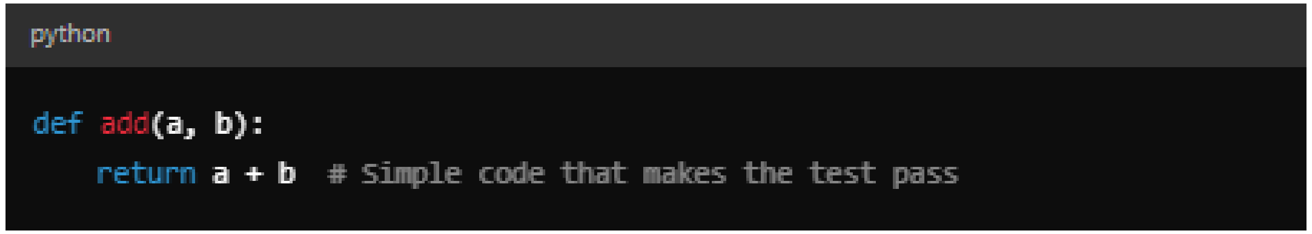
```python
def test_add():
    assert add(2, 3) == 5  # There's no 'add' function yet, so this will fail
```

Running the test now will give a failure message, which means we are in the Red stage

Step 2

Green: Write Just Enough Code to Pass

- Now we move to the Green stage.
- This is where you write just enough code to make the failing test pass. No extra features, just what's required for the test.
- The goal is to get the test from failing to passing as quickly as possible.



python

```python
def add(a, b):
    return a + b  # Simple code that makes the test pass
```

Once the test passes, you can move on. But don’t forget, passing the test doesn’t mean the code is perfect!

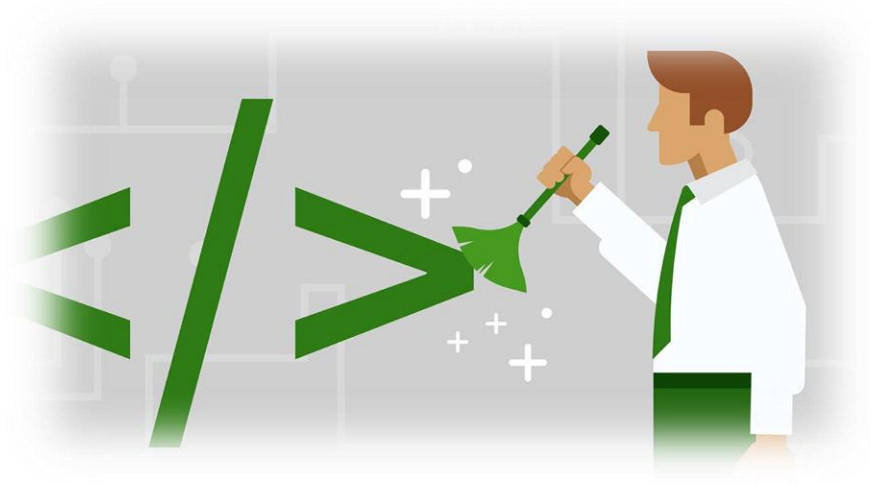
Step 3

Refactor: Improve the Code:

- After the test passes, we enter the Refactor stage.
- Here, we focus on improving the code without changing its behaviour.
- This might mean cleaning up the logic, removing duplication, or renaming variables for clarity.

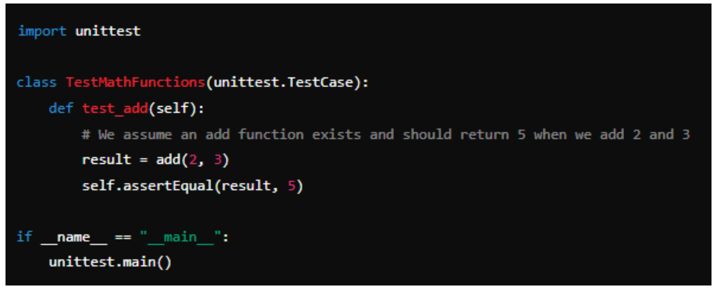
Example:

1. Perhaps we realize that the add() function is fine for now, so we don’t need to make changes.
2. But if we had complex logic, we’d aim to simplify it.
3. The key rule here: Don’t break the test! Keep running the test as you refactor to ensure it still passes.



Example of TDD in Action

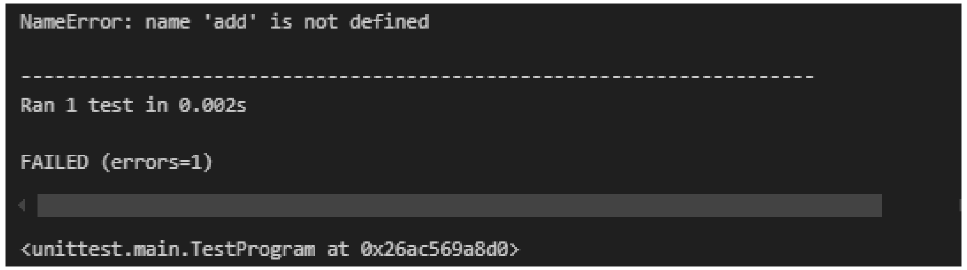
Step 1: Write the test first, ensuring that it describes the desired behaviour:



```python
import unittest

class TestMathFunctions(unittest.TestCase):
    def test_add(self):
        # We assume an add function exists and should return 5 when we add 2 and 3
        result = add(2, 3)
        self.assertEqual(result, 5)

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

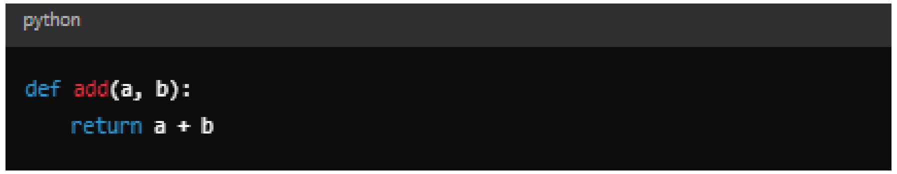
- The test_add function expects an add() function that adds two numbers together.

Step 2: Run the test—it will fail because the add function doesn't exist yet:



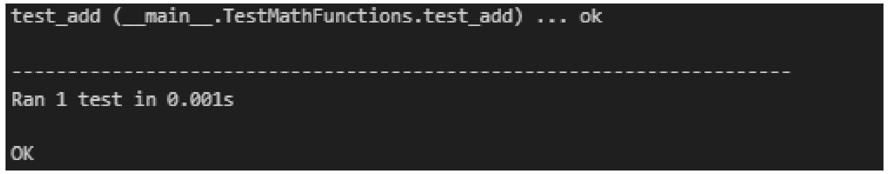
```
NameError: name 'add' is not defined

----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (errors=1)

<unittest.main.TestProgram at 0x26ac569a8d0>
```

Step 3: Now, write the simplest add() function to make the test pass:



```python
def add(a, b):
    return a + b
```

Step 4: Run the test again, and now it passes:



```
test_add (__main__.TestMathFunctions.test_add) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Step 5: Optionally, refactor the add() function if needed. In this simple case, there's no need for refactoring..

Testing In Python Using Inbuilt Module

Overview

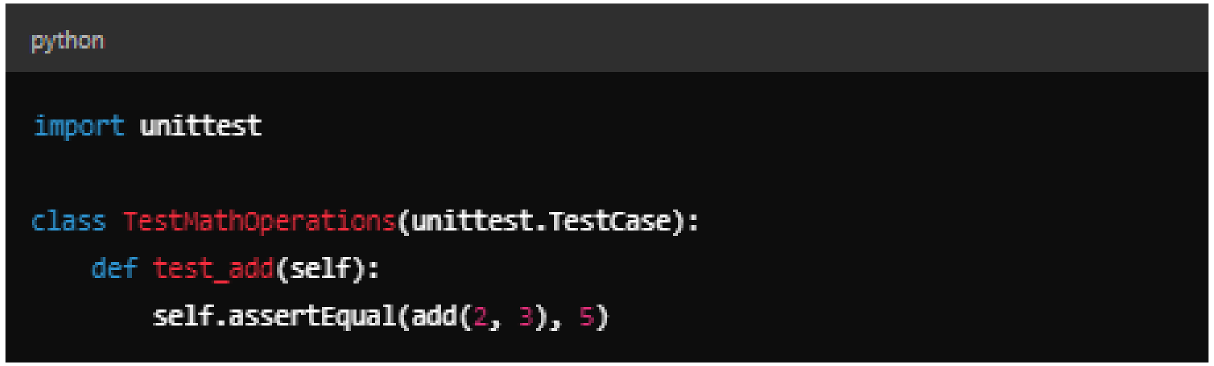
- Python has built-in support for testing with the unittest module.
- Writing tests allows us to ensure that our code works as expected.
- Testing early and often is crucial in development to catch bugs.
- Let’s explore the basics of writing and running tests in Python using unittest.

Test Structure

- Tests in Python are written as methods inside a class that inherits from unittest.TestCase.
- Test cases are individual scenarios where you check if the output matches your expectations.

A typical structure includes:

- Set up the code you want to test.
- Write assertions to compare expected and actual results.



python

```
import unittest

class TestMathOperations(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)
```

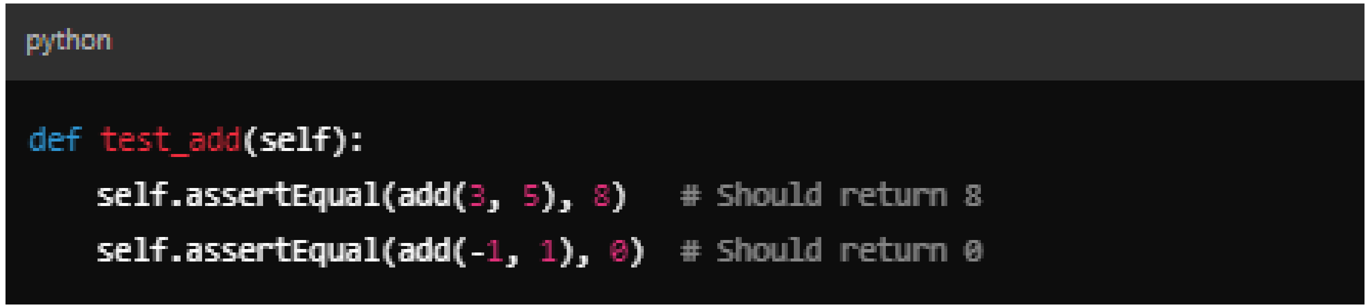
unittest Test Cases

In unittest, test methods start with the word "test" so they can be recognized by the testing framework.

Assertions are used to check if the results match what's expected.

Common assertions:

- assertEqual(): Check if two values are equal.
- assertTrue(): Check if a value is True.
- assertFalse(): Check if a value is False..

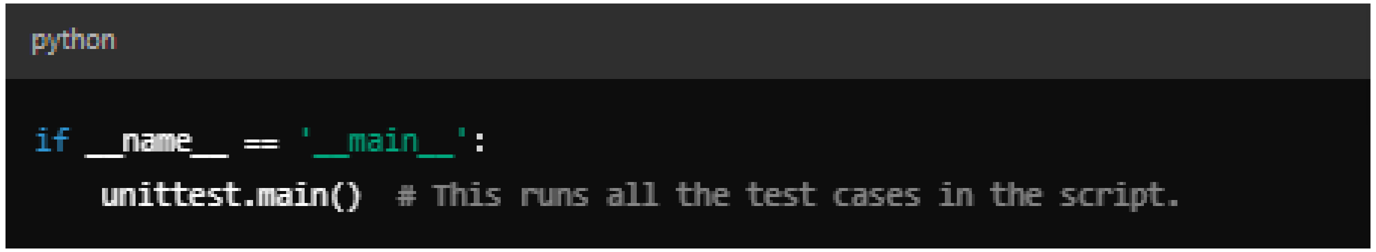


python

```python
def test_add(self):
    self.assertEqual(add(3, 5), 8)   # Should return 8
    self.assertEqual(add(-1, 1), 0)  # Should return 0
```

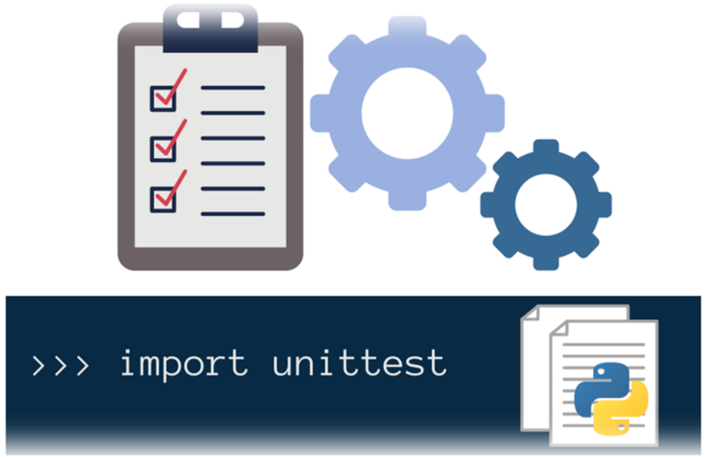
Running Tests

- Once your test cases are written, you can run them using Python's test runner.
- Simply call unittest.main() at the bottom of your script, and it will automatically find and run all test cases.
- When you run the file, you'll see results for each test, showing whether it passed or failed.



python

```python
if __name__ == '__main__':
    unittest.main()  # This runs all the test cases in the script.
```



Thoughts

- Catch bugs early: Writing tests helps catch issues in your code before they become big problems.
- Confidence in code changes: When you make changes, you can run tests to ensure nothing breaks.
- Document your expectations: Tests serve as documentation of what your code should do.
- Testing is an essential habit for professional developers!