

Python Unittest

What is unittest?

unittest is a Python module that allows you to write small pieces of code called tests to check the correctness of your main code. Think of unittest as a tool that acts like an automated reviewer—checking if your functions return the expected output when given specific inputs. It's part of Python's standard library, meaning you don't need to install anything extra to start using it.

Why Use unittest?

Imagine you write a function that adds two numbers. How do you know that function works correctly for all kinds of numbers—positive, negative, or zero? You could manually run the function several times with different inputs and check the results. However, this approach becomes difficult and error-prone as your code grows. This is where unittest comes in:

- **Automates the Testing Process:** unittest runs your tests automatically and reports the results.
- **Catches Bugs Early:** By running tests frequently, you can catch errors as soon as they happen, making debugging easier.
- **Improves Code Confidence:** With thorough testing, you can make changes to your code with confidence, knowing that tests will alert you if something breaks.

How does unittest work?

To use unittest, you create a new class that inherits from `unittest.TestCase`. This class can have one or more methods that test specific pieces of your code. Each method that starts with `test_` is recognized as a test by the framework.

Key Components of unittest:

- **Test Case:** A single unit of testing. Each test checks one aspect of your code.
- **Assertions:** Statements that verify a condition is true. If an assertion fails, the test fails.
- **Test Suite:** A collection of test cases that can be run together.
- **Test Runner:** The part of unittest that runs the tests and displays the results.

Writing Your First Test Case

Here's how to write a basic test using unittest:

Step 1: Import the unittest module.

Step 2: Create a class that inherits from `unittest.TestCase`.

Step 3: Write methods that start with `test_` to test different aspects of your code.

Step 4: Use assertions to check if the actual output matches the expected output.

Example

Imagine you have a simple function that adds to numbers:

```
1 # math_operations.py
2 def add(a, b):
3     return a + b
```

Now, let's write a unit test for this function:

```
1 # test_math_operations.py
2 import unittest
3 from math_operations import add
4
5 class TestMathOperations(unittest.TestCase):
6     def test_add_positive_numbers(self):
7         self.assertEqual(add(2, 3), 5) # Checks if 2 + 3 = 5
8
9     def test_add_negative_numbers(self):
10        self.assertEqual(add(-1, -1), -2) # Checks if -1 + -1
11        = -2
12
13    def test_add_mixed_numbers(self):
14        self.assertEqual(add(-1, 1), 0) # Checks if -1 + 1 = 0
15
16 if __name__ == '__main__':
17     unittest.main()
```


Breaking Down the Code

- `import unittest`: Imports the unittest module.
- `from math_operations import add`: Imports the function we want to test.
- `class TestMathOperations(unittest.TestCase)`: Creates a new test class that unittest recognizes as a collection of test cases.
- `def test_add_positive_numbers(self)`:: A test method that checks if the `add()` function works correctly with positive numbers.
- `self.assertEqual()`: An assertion that checks if the first argument equals the second. If not, the test fails.

Understanding assertions:

1. `self.assertEqual(a, b)`

- **When to Use**: Use this assertion to test if two values are exactly the same. This is ideal when you want to verify that a function's output matches an expected result.
- **Example Use Case**: Verifying that an `add()` function correctly adds two numbers.



```


1  import unittest
2
3  def add(x, y):
4      return x + y
5
6  class TestMathOperations(unittest.TestCase):
7      def test_add(self):
8          result = add(2, 3)
9          self.assertEqual(result, 5) # Passes because 2 + 3
10         = 5
11     if __name__ == '__main__':
12         unittest.main()

```

Explanation: The test checks if `add(2, 3)` equals 5. If it does, the test passes. If it doesn't (e.g., `add(2, 3)` returned 6), the test would fail and show that `5 != 6`.

2. `self.assertNotEqual(a, b)`

- When to Use: Use this when you want to ensure that two values are not equal. This is helpful for tests where certain outputs or results should not be equal to a specific value.
- Example Use Case: Confirming that a `subtract()` function does not mistakenly return a positive number for certain inputs.



```

1  import unittest
2
3  def subtract(x, y):
4      return x - y
5
6  class TestMathOperations(unittest.TestCase):
7      def test_subtract(self):
8          self.assertNotEqual(subtract(5, 5), 1) # Ensures 5 - 5 is n
9          ot 1
10     if __name__ == '__main__':
11         unittest.main()

```

Explanation: The test ensures that `multiply(2, 3)` does not return 8. Since `2 * 3` is 6, the test passes.

3. `self.assertTrue(x)`

- When to Use: Use this to check if a condition or value evaluates to `True`. It's useful when testing conditions or boolean outputs.
- Example Use Case: Checking if a function correctly identifies positive numbers.

```

1 import unittest
2
3 def is_positive(n):
4     return n > 0
5
6 class TestMathOperations(unittest.TestCase):
7     def test_is_positive(self):
8         self.assertTrue(is_positive(10)) # Ensures that 10 is correctly identified as positive
9
10 if __name__ == '__main__':
11     unittest.main()

```

Explanation: The test checks if `is_positive(5)` returns `True`. Since $5 > 0$, the condition is `True` and the test passes.

4. `self.assertFalse(x)`

- When to Use: Use this when you want to check if a condition or value evaluates to `False`. This is helpful for tests where something should return `False`.
- Example Use Case: Confirming that a function correctly identifies non-positive numbers.

```

1 import unittest
2
3 def is_negative(n):
4     return n < 0
5
6 class TestMathOperations(unittest.TestCase):
7     def test_is_negative(self):
8         self.assertFalse(is_negative(5)) # Ensures that 5 is not identified as negative
9
10 if __name__ == '__main__':
11     unittest.main()

```

Explanation: The test ensures that `is_negative(5)` returns `False`. Since 5 is not less than 0, the function correctly returns `False`, and the test passes.

5. `self.assertIsNone(x)`

- When to Use: Use this to verify that a value or function output is `None`. This is common when checking for empty returns or non-existent results.
- Example Use Case: Validating that a search function returns `None` when an item is not found.

```

1 import unittest
2
3 def find_item_in_list(item, lst):
4     return item if item in lst else None
5
6 class TestListOperations(unittest.TestCase):
7     def test_find_item_not_in_list(self):
8         self.assertIsNone(find_item_in_list(10, [1, 2, 3])) # Ensures the result is None if 10 is not in the list
9
10 if __name__ == '__main__':
11     unittest.main()

```

Explanation: The test ensures `find_item_in_list(10, [1, 2, 3])` returns `None`. Since 10 is not in the list, the function returns `None`, and the test passes.

6. `self.assertRaises(exception, callable, *args, **kwargs)`
- When to Use: Use this to check if a specific exception is raised during the execution of a function. This is helpful when testing error handling in your code.
 - Example Use Case: Ensuring that a `divide()` function raises a `ValueError` when dividing by zero.

```
1  import unittest
2
3  def divide(x, y):
4      if y == 0:
5          raise ValueError("Cannot divide by zero")
6      return x / y
7
8  class TestMathOperations(unittest.TestCase):
9      def test_divide_by_zero(self):
10         with self.assertRaises(ValueError): # Use this to check if ValueError is r
11             divide(10, 0)
12
13 if __name__ == '__main__':
14     unittest.main()
```

Explanation: The test checks that calling `divide(10, 0)` raises a `ValueError`. The `with self.assertRaises()` block runs `divide(10, 0)`, and if it raises `ValueError`, the test passes. If no exception or the wrong exception is raised, the test fails.

Quick Recap of When to Use Each Assertion:

Assertion Method	When to Use
<code>self.assertEqual(a, b)</code>	When checking if a equals b (e.g., validating results).
<code>self.assertNotEqual(a, b)</code>	When ensuring a does not equal b (e.g., unwanted results).
<code>self.assertTrue(x)</code>	When checking if x is True (e.g., conditions or flags).
<code>self.assertFalse(x)</code>	When ensuring x is False (e.g., negative conditions).
<code>self.assertIsNone(x)</code>	When confirming x is None (e.g., empty results).
<code>self.assertRaises(exception, callable, *args, **kwargs)</code>	When testing if an exception is raised (e.g., error handling).

Understanding TDD:

- **Red-Green-Refactor Cycle:** The TDD process follows a loop consisting of three main steps:
- **Red:** Write a test case for the function or feature that does not yet exist, ensuring it will fail.
- **Green:** Implement the minimum code required to make the test pass.
- **Refactor:** Improve the code while keeping the test green, ensuring maintainability and optimization without changing the behavior.

1. Understanding Unit Tests Conceptually

- **Unit Test Purpose:** Unit tests are designed to validate that a specific function or component of your code works as expected. Each test should focus on one small part of the functionality.
- **Framework Basics:** Python's unittest module provides a framework for creating and running unit tests. It allows you to create test cases as classes that inherit from unittest.TestCase.

2. Reading and Analyzing Test Cases

To understand which functions to create from test cases, you need to:

- **Identify Inputs and Expected Outputs:** Look at the test methods to see what inputs are being tested and what outputs or behavior is expected.
- **Understand Edge Cases and Coverage:** Check if the test includes edge cases (e.g., empty inputs, boundary values). This helps you determine the scope and robustness required for your function.
- **Follow the Naming and Structure:** Test method names often indicate what functionality they test (e.g., test_add_valid_numbers suggests a function related to addition).

3. Mapping Test Cases to Function Creation

Step-by-Step Process:

- **Read a Test Case:** For example, if you have a test case called test_is_even_number, it likely tests a function that checks if a number is even.
- **Identify the Purpose:** What does the test check? For test_is_even_number, it might pass an integer and expect True for even numbers and False otherwise.
- **Create a Function Outline:** Based on the test, write a function with the appropriate name and parameters (def is_even(n):).
- **Implement Based on Expected Behavior:** Use the inputs and expected results in the test to determine the logic inside the function.

4. Examples for Practice

- Consider the following example test:
- function knowing that any deviation from expected behavior will be caught by a failing test.



```
1 import unittest
2 from my_module import is_even
3
4 class TestMathFunctions(unittest.TestCase):
5     def test_is_even_number(self):
6         self.assertTrue(is_even(2))
7         self.assertFalse(is_even(3))
8         self.assertTrue(is_even(0))
9         self.assertFalse(is_even(-1))
```

Breaking Down the Test:

Test Function Name: `test_is_even_number` suggests the function `is_even` should check if a number is even.

Expected Behavior:

- Input of 2 should return True.
- Input of 3 should return False.
- Input of 0 should return True (zero is even).
- Input of -1 should return False.

Creating the Corresponding Function:



```
1 def is_even(n):
2     return n % 2 ==
    0
```

5. Iterative Development with TDD (Test-Driven Development)

- Write a Test First: Start by writing a failing test that defines what the function should do.
- Run the Test: Execute the test with unittest to confirm it fails (this is expected initially).
- Implement the Function: Write just enough code to make the test pass.
- Refactor and Expand: Once the test passes, refactor the code if necessary and add more tests to cover additional cases.

6. Common Patterns for Test-Driven Function Design

- CRUD Operations: Tests that create, read, update, or delete data suggest functions that handle those operations.
- Validation Checks: If a test checks for valid input, the function should handle input validation and return appropriate responses.
- Calculations and Conversions: If a test asserts a mathematical or conversion result, the function should perform those operations.

7. Hints to Know What's Needed for Functions

- Assertions in Tests: `self.assertEqual`, `self.assertTrue`, `self.assertFalse`, etc., show what outcomes the function needs to provide.
- Setup Methods: `setUp` or `setUpClass` can indicate the function needs preconditions or a certain state before running tests.
- Mocking and Dependencies: If a test uses `mock` or `patch`, it implies your function interacts with external dependencies (e.g., files, databases).

8. Practical Tips

- Start Simple: Implement the simplest solution that can pass the initial test.
- Use Debugging: If a test fails, use print statements or a debugger to understand why.
- Document Assumptions: If the test isn't clear, add comments to explain what you assume about the function's purpose.