Python Unit Testing

@overview

Unit Testing

Unit testing is a software testing technique where individual units or components of a software are tested in isolation from the rest of the application. The primary goal is to validate that each unit of the software performs as expected.

What are Test Cases?

Test cases are the building blocks of unit testing. Each test case is a small, focused check that verifies whether a specific part of your code (such as a function or method) behaves as expected under certain conditions. By running test cases, you can quickly identify bugs and ensure your code remains reliable as it evolves.

Test cases typically include:

- **Input data:** The values or objects you provide to the code under test.
- **Expected outcome:** What you anticipate the code should return or how it should behave.
- **Assertions:** Statements that compare the actual outcome to the expected one.

Why Write Test Cases?

Writing test cases helps you:

- Catch bugs early, before they reach users.
- Document how your code is supposed to work.
- Collaborate with others by providing clear expectations for code behavior.
- Improve code quality and reliability over time.
- Support debugging by isolating issues to specific units of code.
- Encourage better design practices by promoting modular and testable code.
- Save time and effort in the long run by reducing the need for manual testing.
- Build confidence in your codebase, especially when making significant changes or adding new features.
- Ensure compliance with industry standards and regulations that may require thorough testing.
- Provide a safety net for experimenting with new features or optimizations.

Types of Test Cases in Unit Testing

Unit tests generally fall into two categories:

1. Positive Test Cases

- **Purpose:** Confirm that your code works correctly when given valid, expected inputs.
- **Example:** Testing that `add(2, 3)` returns `5`.

2. Negative Test Cases

- **Purpose:** Check how your code handles invalid or unexpected inputs, such as wrong data types or out-of-range values.
- **Example:** Testing that `divide(10, 0)` raises a `ZeroDivisionError`.

Both types are essential:

- **Positive tests** ensure your code does what it should.
- **Negative tests** ensure your code fails gracefully and securely when things go wrong.

Characteristics of a Good Test Suite

A test suite is a collection of tests designed to catch errors in your software before it reaches users. An effective test suite runs quickly, gives you confidence in your code when all tests pass, and provides helpful feedback when something goes wrong.

A poor test suite, on the other hand, may be slow, fail to inspire confidence even when passing, or provide unhelpful feedback when a bug is detected, making it harder to identify and fix issues.

There are six key characteristics that make a test suite effective. These are:

- **Fast:** Tests should run quickly, so you get feedback without long waits. Fast tests make it easier to run them frequently during development, saving time and improving productivity.
- **Complete:** A complete test suite covers as much of your codebase as possible, catching errors that might arise from changes or new features. Striking a balance between speed and completeness is important.
- **Reliable:** Reliable tests produce consistent results, regardless of changes outside the test's scope. Flaky tests that fail intermittently can erode trust in your test suite.
- **Isolated:** Each test should run independently, without affecting others. This often means cleaning up any data or state changes after each test, so tests don't interfere with one another.
- **Maintainable:** A maintainable test suite is easy to update as your code evolves. You should be able to add, modify, or remove tests without difficulty, keeping your suite relevant and effective.
- **Expressive:** Tests should be clear and descriptive, serving as documentation for your code. Well-written tests help others (and your future self) understand what your software is supposed to do.

By focusing on these qualities, you'll build test suites that are efficient, trustworthy, and easy to work with as your projects grow.

Multiple Choice Question

Which of the following best describes a *negative test case*?

[[ ]] A test that checks if the code works with valid inputs. [[X]] A test that checks if the code handles invalid or unexpected inputs properly. [[ ]] A test that measures the speed of the code. [[ ]] A test that checks the code formatting.
****

Negative test cases are designed to ensure that the code can handle erroneous or unexpected inputs gracefully, such as raising appropriate exceptions or returning error messages.

How to Write Unit Tests for Python Modules

Follow these steps to create effective unit tests in Python:

1. Choose a Test Framework

- **unittest:** Python's built-in testing framework. Great for beginners.
- **pytest:** A popular third-party framework with advanced features and a simple syntax.

We are going to use `unittest`.

2. Organize Your Tests

- Create a new file named `test_{module_name}.py` (e.g., `test_math_utils.py`).
- Import the test framework and the module you want to test.

3. Write Test Classes and Methods

- Define a class that inherits from `unittest.TestCase`.
- Write methods that start with `test_` to represent individual test cases.

@unittest_fix

```
import unittest

class Tests(unittest.TestCase):
    def test_bigger(self):
        self.assertTrue( 1 < 0 )

    def test_equals(self):
        self.assertEqual( 1+1, 2 )
```

```
    def test_div(self):
        with self.assertRaises(ZeroDivisionError):
            1 / 0

if __name__ == '__main__':
    unittest.main(verbosity=2) # verbosity=2 for more detailed output
```

@Pyodide.eval

4. Run Your Tests

In this lab you are able to run the code directly in the browser but that will not be the case for your own projects.

How exactly you run your tests will depend on the contents of your test file, specifically if you included the test runner with the line `unittest.main()`.

- If you have included the test runner, you can run the test file directly, e.g. `python3 testfile.py`.
- Alternative you can call the runner yourself, i.e. `python3 -m unittest testfile.py`.
  - This also works even if you have included `unittest.main()` line and superceeds it.

Using the test runner approach is often preferrable as it allows you to have more control over how the tests are run.

For example:

Command	Description
`python3 -m unittest testfile.py`	Run all tests in the specified file
`python3 -m unittest discover`	Run all tests in the current directory
`python3 -m unittest discover -k pattern`	Run all tests in whose names match the pattern
`python3 -m unittest discover --verbose`	Run all tests with detailed output

So if, for example, you had multiple test files covering all the different modules in your project, you could run all the tests with a single command.

Multiple Choice Question

What is the main purpose of assertions in a test case?

[[ ]] To print the output of the function. [[X]] To compare the actual result with the expected result. [[ ]] To import the module under test. [[ ]] To document the code. **********

Assertions are used to verify that the actual outcome of a function matches the expected outcome.

Summary

- Test cases help ensure your code works as intended and handles errors gracefully.
- Use both positive and negative test cases for comprehensive coverage.
- Organize your tests clearly and use assertions to check outcomes.

Test Driven Development (TDD)

Test Driven Development (TDD) is a software development approach where you write tests before writing the actual code. The process follows a simple cycle known as "Red-Green-Refactor":

1. **Red:** Write a failing test case that defines a function or improvements of a function.
2. **Green:** Write the minimum amount of code necessary to make the test pass.
3. **Refactor:** Clean up the code while keeping the tests green (passing).

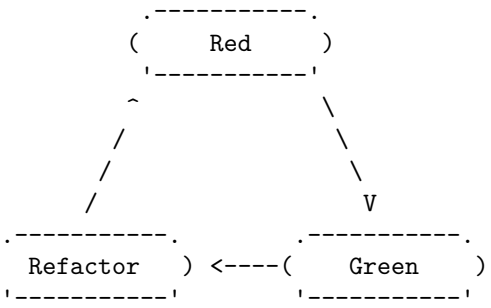
This approach helps ensure that your code is well-tested from the start, encourages simple designs, and improves code quality.

Benefits of TDD

- **Improved Code Quality:** Writing tests first helps clarify requirements and leads to cleaner, more maintainable code.
- **Early Bug Detection:** Since tests are written before the code, bugs are caught early in the development process.
- **Refactoring Confidence:** With a comprehensive test suite, you can refactor code without fear of breaking existing functionality.
- **Better Design:** TDD encourages modular, loosely coupled code, making it easier to extend and maintain.

TDD Workflow Example

Let's walk through a simple TDD cycle for a function that counts the number of vowels in a string.



1. Red Phase

Write a failing tests that defines the desired functionality.

In this case we want to count the number of vowels in a string.

@unittest_fix

```
# test_myvowels.py
import unittest

class TestCountVowels(unittest.TestCase):
    def test_consonants(self):
        self.assertEqual(count_vowels("rhythm"), 0)

    def test_all_vowels(self):
        self.assertEqual(count_vowels("aeiou"), 5)

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

@Pyodide.eval

At this point, running the tests will fail because `count_vowels` does not exist yet. But make sure to run the tests to confirm they fail as expected.

Warning

If you complete the later steps and then return to this page, the tests might pass because the function will have been implemented. To start fresh and see the tests fail as intended, click the button below to reset the environment.

```
# Remove count_vowels if it exists in the current environment
if 'count_vowels' in globals():
    del count_vowels
```

@Pyodide.hide

2. Green Phase

Write the minimum code to pass the tests.

```
# myvowels.py
def count_vowels(string):
    vowels = "aeiou"

    count = 0
    for char in string:
        if char in vowels:
            count += 1
    return count
```

@Pyodide.eval

Run the tests again

```
if "TestCountVowels" not in globals():
    print("Unit tests not defined, please run the Red phase first.")
else:
    suite = unittest.TestLoader().loadTestsFromTestCase(TestCountVowels)
    unittest.TextTestRunner(verbosity=2).run(suite)
```

@Pyodide.hide

Now, running the tests should pass. If the tests do not pass, adjust the implementation until they do.

Importantly, the tests define the requirements for the function. So definitive decisions about how the function should behave are made before the implementation is written, i.e. does 'y' count as a vowel?

3. Refactor

Review your code for improvements. In this simple case, the function is quite straightforward but there are more concise ways available.

```
# myvowels.py
def count_vowels(string):
    vowels = "aeiou"
    return len([char for char in string if char in vowels])
```

@Pyodide.eval

Run the tests again

```
if "TestCountVowels" not in globals():
    print("Unit tests not defined, please run the Red phase first.")
else:
    suite = unittest.TestLoader().loadTestsFromTestCase(TestCountVowels)
    unittest.TextTestRunner(verbosity=2).run(suite)
```

@Pyodide.hide

4. Red Phase (again)

We want to expand the functionality of our function so that it can handle uppercase letters as well. So we write more failing tests covering this new functionality.

```
# test_myvowels.py
import unittest

class TestCountVowels(unittest.TestCase):
    def test_consonants(self):
        self.assertEqual(count_vowels("rhythm"), 0)

    def test_all_vowels(self):
        self.assertEqual(count_vowels("aeiou"), 5)

    def test_uppercase_consonants(self):
        self.assertEqual(count_vowels("RHYTHM"), 0)

    def test_uppercase_vowels(self):
        self.assertEqual(count_vowels("AEIOU"), 5)

    def test_mixed_case(self):
        self.assertEqual(count_vowels("An example sentence"), 7)
```

@Pyodide.eval

Make sure to run the tests to confirm they fail as expected.

```
if "TestCountVowels" not in globals():
    print("Unit tests not defined, please run the Red phase first.")
else:
    suite = unittest.TestLoader().loadTestsFromTestCase(TestCountVowels)
    unittest.TextTestRunner(verbosity=2).run(suite)
```

@Pyodide.hide

Important note

It is important that your tests should fail in the Red phase.

Failing tests demonstrate that the tests *can* fail. As such when they pass, it is evidence that the code is working as intended. If the tests pass by default, are they actually testing your code or do they just pass no matter what?

5. Green Phase (again)

Write the minimum code to pass the new tests.

```
def count_vowels(string):
    vowels = "aeiouAEIOU"
    return len([char for char in string if char in vowels])
```

@Pyodide.eval

Check that the tests pass

@unittest_run

If the tests do not pass, adjust the implementation until they do.

6. Refactor Again if Necessary

Our previous implementation was quite concise already but we can make it even more efficient by using a generator expression with `sum`.

```
def count_vowels(string):
    vowels = set("aeiouAEIOU")
    return sum(1 for char in string if char in vowels)
```

@Pyodide.eval

Make sure to run the tests again to confirm they still pass.

```
unittest.main()
""
```

@Pyodide.hide

We can continue this cycle, adding more tests and functionality as needed.

TDD Best Practices

- Write small, focused tests for each piece of functionality.
- Only write enough code to make the test pass.
- Refactor regularly to keep code clean.
- Use descriptive test names to clarify intent.

Multiple Choice Question

What is the first step in the TDD cycle?

[[ ]] Write the implementation code. [[ ]] Refactor the code. [[X]] Write a failing test. [[ ]] Deploy the application. **********

~~Test~~ Driven Development always starts with creating the tests first and ensuring they fail.

Which step of the TDD cycle is focused on writing the smallest amount of code necessary to make the failing test pass?

[[ ]] Red.
[[X]] Green.
[[ ]] Refactor.
[[ ]] Deploy.
****

The Green phase is where you write the minimal implementation required to satisfy the failing test, keeping changes small and focused.

Which step of the TDD cycle is focused on writing the smallest amount of code necessary to make the failing test pass?

[[ ]] Red.
[[X]] Green.
[[ ]] Refactor.
[[ ]] Deploy.
****

The Green phase is where you write the minimal implementation required to satisfy the failing test, keeping changes small and focused.

Additional Resources

- python.org is a great resource for documentation, FAQs, and tutorials for beginners, as well as information about what is happening in the wider Python community. Check it out and explore!
- If you're interested in practicing more with Google Colab, check out this notebook looking at statistics.
- If you are ready to actually write some Python code, check out the Python Basics: Functions, Methods, and Variables module.

Recap

@recap