A Guide to Python unittest Framework
Software Testing and Maintenance Course

Whitecliffe

March 30, 2025

Contents

1 Introduction to unittest

Python's `unittest` framework is part of the Python standard library and provides a rich set of tools for constructing and running tests. It's inspired by JUnit and has a similar flavor to major unit testing frameworks in other languages.

The framework supports:

- Test automation
- Sharing of setup and shutdown code for tests
- Aggregation of tests into collections
- Independence of the tests from the reporting framework

2 Basic Structure

A typical unittest consists of:

- **Test fixture**: Setup needed to run one or more tests and any associated cleanup actions
- **Test case**: The individual unit of testing
- **Test suite**: A collection of test cases
- **Test runner**: Component that orchestrates test execution and provides results

3 Creating Your First Test

```python
import unittest

# The class we want to test
class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

# Our test case
class TestCalculator(unittest.TestCase):

    def setUp(self):
        # This runs before each test
        self.calc = Calculator()

    def test_add(self):
        result = self.calc.add(3, 5)
        self.assertEqual(result, 8)

    def test_subtract(self):
        result = self.calc.subtract(10, 4)
        self.assertEqual(result, 6)

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

In this example:

- We create a test class that inherits from `unittest.TestCase`
- Test methods must start with the word `test`
- `setUp` method initializes common resources before each test
- `assertEqual` is an assertion method that verifies expected results

4 Running Tests

There are several ways to run unittest tests:

4.1 From command line

```
# Run a specific test file
python -m unittest test_calculator.py

# Run tests with discovery (automatically finds tests)
python -m unittest discover

# Run a specific test case
python -m unittest test_calculator.TestCalculator

# Run a specific test method
python -m unittest test_calculator.TestCalculator.test_add
```

4.2 From within the file

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

5 Test Lifecycle Methods

```
class TestExample(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        # Runs once before all tests in the class
        print("setUpClass: Setting up class-wide resources")

    def setUp(self):
        # Runs before each test
        print("setUp: Setting up test resources")

    def test_example1(self):
        print("Running test_example1")
        self.assertTrue(True)

    def test_example2(self):
        print("Running test_example2")
        self.assertFalse(False)

    def tearDown(self):
        # Runs after each test
        print("tearDown: Cleaning up test resources")

    @classmethod
    def tearDownClass(cls):
        # Runs once after all tests in the class
        print("tearDownClass: Cleaning up class-wide resources")
```

The execution order for this class would be:

1. `setUpClass()`
2. `setUp()`
3. `test_example1()`
4. `tearDown()`
5. `setUp()`
6. `test_example2()`

7. `tearDown()`
8. `tearDownClass()`

6 Common Assertions

6.1 Equality Assertions

```
# Equality
self.assertEqual(a, b)           # a == b
self.assertNotEqual(a, b)        # a != b
self.assertAlmostEqual(a, b)     # round(a-b, 7) == 0 (for floating point)
self.assertNotAlmostEqual(a, b)  # round(a-b, 7) != 0
```

6.2 Boolean Assertions

```
# Boolean
self.assertTrue(x)               # bool(x) is True
self.assertFalse(x)              # bool(x) is False
```

6.3 Identity Assertions

```
# Identity
self.assertIs(a, b)              # a is b
self.assertIsNot(a, b)           # a is not b
self.assertIsNone(x)             # x is None
self.assertIsNotNone(x)          # x is not None
```

6.4 Type Checking

```
# Type checking
self.assertIsInstance(a, b)      # isinstance(a, b)
self.assertNotIsInstance(a, b)   # not isinstance(a, b)
```

6.5 Container Assertions

```
# Containers
self.assertIn(a, b)              # a in b
self.assertNotIn(a, b)           # a not in b
self.assertCountEqual(a, b)      # Same elements, regardless of order
```

6.6 Exception Assertions

```
# Exceptions
self.assertRaises(Exception, callable, *args, **kwargs)
with self.assertRaises(Exception):
    # Code that should raise the exception
    raise Exception("Expected exception")
```

6.7 Regex Assertions

```
# Regex
self.assertRegex(text, regex)    # regex.search(text)
self.assertNotRegex(text, regex) # not regex.search(text)
```

7 Organizing Test Suites

You can organize tests into custom suites:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(TestCalculator('test_add'))
    suite.addTest(TestCalculator('test_subtract'))
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

This is helpful when you want to:

- Run a specific subset of tests
- Group tests from different classes
- Run tests in a specific order

8 Skipping Tests

unittest provides decorators for skipping tests under certain conditions:

```
import unittest
import sys

class TestSkipping(unittest.TestCase):
    @unittest.skip("Demonstrating skip")
    def test_skipped(self):
        self.fail("This shouldn't run")

    @unittest.skipIf(sys.version_info < (3, 9), "Requires Python 3.9+")
    def test_python_39_feature(self):
        # Test some Python 3.9+ feature
        pass

    @unittest.skipUnless(sys.platform == "win32", "Windows only test")
    def test_windows_feature(self):
        # Test some Windows-specific feature
        pass

    def test_expected_failure(self):
        # This test is expected to fail
        self.assertEqual(1, 0)
    test_expected_failure = unittest.expectedFailure(test_expected_failure)
```

The available skip decorators include:

- `@unittest.skip(reason)`: Unconditionally skip the test
- `@unittest.skipIf(condition, reason)`: Skip if condition is true
- `@unittest.skipUnless(condition, reason)`: Skip unless condition is true
- `@unittest.expectedFailure`: Test is expected to fail

9 Parameterized Tests

While unittest doesn't natively support parameterized tests, you can implement them using `subTest`:

```
class TestParameterized(unittest.TestCase):
    def test_multiplication(self):
        test_cases = [
            (2, 3, 6),    # (a, b, expected)
            (0, 5, 0),
            (-1, -1, 1),
            (10, -2, -20)
```

```
        ]

        for a, b, expected in test_cases:
            with self.subTest(a=a, b=b):
                result = a * b
                self.assertEqual(result, expected)
```

The advantages of using `subTest` include:

- All test cases run even if some fail
- Clear reporting of which test case failed
- No need for separate test methods for each case

10 Mock Objects with unittest.mock

The `unittest.mock` module provides powerful facilities for creating mock objects:

```
from unittest.mock import Mock, patch

class TestWithMocks(unittest.TestCase):
    def test_mock_function(self):
        # Create a mock object
        mock = Mock(return_value=42)

        # Call the mock
        result = mock(1, 2, 3)

        # Assert it was called with correct arguments
        mock.assert_called_with(1, 2, 3)

        # Assert the result
        self.assertEqual(result, 42)

    def test_mock_method(self):
        # Create an object with mock methods
        obj = Mock()
        obj.method.return_value = 'mocked value'

        # Call the method
        result = obj.method(x=1)

        # Assert it was called correctly
        obj.method.assert_called_once_with(x=1)
        self.assertEqual(result, 'mocked value')

    @patch('module.function')
    def test_patched_function(self, mock_function):
        # Set up the mock
        mock_function.return_value = 'patched result'

        # Call the function (imported from 'module')
        import module
        result = module.function()

        # Assert the mock was used
        self.assertEqual(result, 'patched result')
```

The `unittest.mock` provides:

- Mocks for functions, methods, and classes
- Automatic verification of call counts and arguments
- Patching of objects at import time or during runtime
- Recording and verification of call history

11 Testing Classes with Context Managers

```
class ResourceManager:
    def __enter__(self):
        print("Acquiring resource")
        return "resource"

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Releasing resource")
        return False  # Don't suppress exceptions

class TestContextManager(unittest.TestCase):
    def test_resource_manager(self):
        with ResourceManager() as resource:
            self.assertEqual(resource, "resource")
            # Test operations with resource
```

This demonstrates how to test classes that implement the context manager protocol (used with `with` statements).

12 Testing Exceptions

```
class TestExceptions(unittest.TestCase):
    def test_exception(self):
        # Simple way to test for exceptions
        with self.assertRaises(ValueError):
            int('not a number')

    def test_exception_message(self):
        # Testing both the exception type and message
        try:
            int('not a number')
            self.fail("ValueError was not raised")
        except ValueError as e:
            self.assertEqual(str(e), "invalid literal for int() with base 10: 'not a number'")

    def test_context_manager_exception(self):
        # Using a context manager with additional assertions
        with self.assertRaises(ValueError) as context:
            int('not a number')

        # Check the exception message
        self.assertIn('invalid literal', str(context.exception))
```

Exception testing allows you to verify:

- That exceptions are raised when expected
- The specific type of exception
- The exception message or contents

13 Best Practices for Writing Effective Tests

13.1 Test Independence

Each test should be able to run independently of others. Tests should not rely on the state left by other tests.

13.2 Descriptive Names

Use descriptive names that explain what's being tested. For example:

```
def test_addition_with_negative_numbers(self):
    # Better than test_add_2()
```

13.3 Single Assertion

Ideally, include only one assertion per test. This makes it clear what failed and why.

13.4 Clean Setup/Teardown

Use `setUp`/`tearDown` methods to avoid code duplication and ensure proper cleanup.

13.5 Test Isolation

Tests should not depend on external resources like networks unless specifically testing those integrations.

13.6 Mocking External Dependencies

Use mocks to isolate the code you're testing from external dependencies.

13.7 Arrange-Act-Assert

Structure tests with clear setup, execution, and verification phases:

```
def test_example(self):
    # Arrange - set up the test conditions
    calculator = Calculator()

    # Act - execute the code being tested
    result = calculator.add(2, 3)

    # Assert - verify the results
    self.assertEqual(result, 5)
```

13.8 Test Edge Cases

Include tests for boundary conditions and edge cases, such as:

- Empty inputs
- Very large values
- Negative values
- Zero values
- Invalid inputs

13.9 Coverage

Aim for good test coverage, but prioritize critical paths. Quality of tests is more important than quantity.

13.10 Keep Tests Fast

Unit tests should execute quickly to provide rapid feedback during development.

14 Test-Driven Development (TDD) with unittest

TDD follows this cycle:

1. Write a failing test
2. Write minimal code to pass the test
3. Refactor the code
4. Repeat

Example TDD process:

```python
# Step 1: Write a failing test
class TestStringReversal(unittest.TestCase):
    def test_reverse_string(self):
        self.assertEqual(reverse_string("hello"), "olleh")

# Step 2: Run the test (it will fail since reverse_string doesn't exist)
# Step 3: Implement minimal code to make the test pass
def reverse_string(s):
    return s[::-1]

# Step 4: Run the test again (it should pass)
# Step 5: Add more tests and refine the implementation
class TestStringReversal(unittest.TestCase):
    def test_reverse_string(self):
        self.assertEqual(reverse_string("hello"), "olleh")

    def test_reverse_empty_string(self):
        self.assertEqual(reverse_string(""), "")

    def test_reverse_palindrome(self):
        self.assertEqual(reverse_string("racecar"), "racecar")
```

15 Advanced Tips

15.1 Custom Test Assertions

```python
class CustomAssertionsTest(unittest.TestCase):
    def assertInRange(self, value, min_val, max_val, msg=None):
        """Assert that value is between min_val and max_val (inclusive)."""
        if not (min_val <= value <= max_val):
            standardMsg = f"{value} not in range [{min_val}, {max_val}]"
            self.fail(self._formatMessage(msg, standardMsg))

    def test_in_range(self):
        self.assertInRange(5, 1, 10)
        # This will fail
        # self.assertInRange(15, 1, 10)
```

Custom assertions allow you to:

- Create domain-specific checks
- Simplify complex verification logic
- Provide clearer error messages

15.2 Using TestCase.subTest for Clearer Failure Reports

```python
class SubTestExample(unittest.TestCase):
    def test_even_numbers(self):
        test_cases = [2, 4, 6, 8]
        for num in test_cases:
            with self.subTest(number=num):
                self.assertEqual(num % 2, 0)

        # If you add 9 to test_cases, only that specific subtest will fail
        # while the others will pass, making it easier to identify the problem
```

The benefits of using `subTest` include:

- All test cases run even if some fail
- Failure reports clearly indicate which specific test case failed
- Contextual information is included in the test report

15.3 Load Tests from Modules

```
import unittest

# Create a test suite from all test cases in multiple modules
suite = unittest.TestLoader().loadTestsFromNames([
    'test_module1',
    'test_module2'
])

# Run the tests
unittest.TextTestRunner().run(suite)
```

This approach allows you to:

- Combine tests from multiple modules
- Create custom test suites programmatically
- Control the test execution order

16 Conclusion

Python's unittest framework provides a robust foundation for testing your code. By mastering these aspects of the framework, you'll be able to write effective, maintainable tests that help ensure the quality of your software.

Key takeaways:

- Start with simple test cases and expand coverage gradually
- Use the appropriate assertion methods for clear test failures
- Structure tests using the Arrange-Act-Assert pattern
- Leverage setup and teardown methods for test efficiency
- Consider Test-Driven Development for critical components
- Use mocks to isolate units under test
- Write independent, repeatable, and fast tests

Testing is an investment in code quality that pays dividends throughout the software lifecycle.