Test Automation Frameworks in Python

Contents

[unitTest 1](#_Toc174001765)

[PyTest 6](#_Toc174001766)

[UnitTest vs PyTest 12](#_Toc174001767)

[PyTest-BDD 16](#_Toc174001768)

# unitTest

**Unit Testing in**

Unit testing is the process of testing individual components (units) of software to ensure that each part functions as expected. 's unittest module, inspired by JUnit in Java, provides a framework for writing and running tests.

**Key Concepts**

1. **Test Case**: The smallest unit of testing. It checks for a specific response to a particular set of inputs.
2. **Test Suite**: A collection of test cases, test suites, or both.
3. **Test Runner**: A component that orchestrates the execution of tests and provides the outcome.
4. **Assertions**: Statements that check if a particular condition is true. If false, the test fails.

**Writing Unit Tests**

To write unit tests in , you typically follow these steps:

1. **Import the unittest module**.
2. **Create a test class** that inherits from unittest.TestCase.
3. **Define test methods** within the class. Each method should start with test\_ to be recognized as a test case.
4. **Use assertions** to check for expected outcomes.

Here’s an example to illustrate these steps:

**Example Code to be Tested**

# mymodule.py

def add(a, b):

return a + b

def subtract(a, b):

return a - b

**Writing Unit Tests**

# test\_mymodule.py

import unittest

from mymodule import add, subtract

class TestMyModule(unittest.TestCase):

def test\_add(self):

self.assertEqual(add(2, 3), 5)

self.assertEqual(add(-1, 1), 0)

self.assertEqual(add(-1, -1), -2)

def test\_subtract(self):

self.assertEqual(subtract(3, 2), 1)

self.assertEqual(subtract(2, 3), -1)

self.assertEqual(subtract(-1, -1), 0)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**Running the Tests**

To run the tests, you can use the following command in the terminal:

python -m unittest test\_mymodule.py

This will discover and run all test cases defined in test\_mymodule.py and report the results.

**Advanced Features**

**1. Setup and Teardown Methods**

Sometimes, you need to prepare resources before running tests or clean up afterward. You can use setUp and tearDown methods for this purpose.

class TestMyModule(unittest.TestCase):

def setUp(self):

# Code to set up test fixtures

self.a = 10

self.b = 5

def tearDown(self):

# Code to tear down test fixtures

pass

def test\_add(self):

self.assertEqual(add(self.a, self.b), 15)

**2. Skipping Tests**

You can skip tests using decorators:

class TestMyModule(unittest.TestCase):

@unittest.skip("Skipping this test")

def test\_add(self):

self.assertEqual(add(2, 3), 5)

@unittest.skipIf(condition, "Skipping if condition is true")

def test\_subtract(self):

self.assertEqual(subtract(3, 2), 1)

**3. Parameterized Tests**

Parameterized tests allow you to run a test with different sets of data. This can be done using the unittest library along with the parameterized library:

pip install parameterized

from parameterized import parameterized

class TestMyModule(unittest.TestCase):

@parameterized.expand([

(2, 3, 5),

(-1, 1, 0),

(-1, -1, -2)

])

def test\_add(self, a, b, expected):

self.assertEqual(add(a, b), expected)

**Executing Tests with a Test Runner**

While you can run tests using the command line, you can also use test runners like pytest for more features and better output.

**Installing pytest**

pip install pytest

**Writing Tests for pytest**

The same tests written for unittest can often run with pytest without changes. However, pytest can also provide more concise syntax:

# test\_mymodule.py

import pytest

from mymodule import add, subtract

def test\_add():

assert add(2, 3) == 5

assert add(-1, 1) == 0

assert add(-1, -1) == -2

def test\_subtract():

assert subtract(3, 2) == 1

assert subtract(2, 3) == -1

assert subtract(-1, -1) == 0

**Running Tests with pytest**

Simply run:

pytest

**Summary**

* **Unit Testing**: Testing individual units of code to ensure they work as expected.
* **unittest Module**: ’s built-in module for writing and running tests.
* **Assertions**: Used to check for expected outcomes in tests.
* **Setup and Teardown**: Methods to prepare and clean up resources for tests.
* **Skipping Tests**: Decorators to skip certain tests conditionally.
* **Parameterized Tests**: Running the same test with different data sets.
* **pytest**: A more powerful test runner with additional features and better output.

Unit testing is an essential practice in software development to ensure the reliability and correctness of code. By using frameworks like unittest and pytest, developers can write and run tests effectively.

# PyTest

**Overview of pytest**

pytest is a popular testing framework that simplifies writing and running tests. It is known for its simple syntax, powerful features, and extensive plugin ecosystem.

**Key Features**

* **Simple Syntax**: Write tests using plain assert statements.
* **Fixtures**: Easily manage setup and teardown code using fixtures.
* **Parameterization**: Run tests with multiple sets of data using parameterized tests.
* **Plugins**: Extend functionality with a wide range of plugins.
* **Detailed Output**: Provides rich and readable test output.

**Example**

Let’s walk through an example to illustrate how to use pytest.

**1. Install pytest**

First, you need to install pytest if you haven’t already:

pip install pytest

**2. Create a Simple Test**

Let’s assume we have a module called calculator.py with the following code:

# calculator.py

def add(a, b):

return a + b

def subtract(a, b):

return a - b

We’ll create a file named test\_calculator.py to write our tests:

# test\_calculator.py

import pytest

from calculator import add, subtract

def test\_add():

assert add(2, 3) == 5

assert add(-1, 1) == 0

assert add(-1, -1) == -2

def test\_subtract():

assert subtract(3, 2) == 1

assert subtract(2, 3) == -1

assert subtract(-1, -1) == 0

**3. Running Tests**

To execute the tests, navigate to the directory containing test\_calculator.py and run:

pytest

pytest will discover and execute all files named test\_\*.py or \*\_test.py and report the results.

**4. Understanding Output**

When you run pytest, you will see output similar to this:

diff

============================= test session starts ==============================

collected 2 items

test\_calculator.py .. [100%]

========================== 2 passed in 0.01 seconds ===========================

This output shows that two tests were collected and both passed.

**Advanced Features**

**1. Fixtures**

Fixtures allow you to set up and tear down resources needed by your tests.

# test\_calculator.py

import pytest

from calculator import add, subtract

@pytest.fixture

def input\_values():

return (10, 5)

def test\_add(input\_values):

a, b = input\_values

assert add(a, b) == 15

def test\_subtract(input\_values):

a, b = input\_values

assert subtract(a, b) == 5

**2. Parameterized Tests**

You can use @pytest.mark.parametrize to run a test with multiple sets of parameters.

# test\_calculator.py

import pytest

from calculator import add, subtract

@pytest.mark.parametrize("a, b, expected", [

(2, 3, 5),

(-1, 1, 0),

(-1, -1, -2),

])

def test\_add(a, b, expected):

assert add(a, b) == expected

@pytest.mark.parametrize("a, b, expected", [

(3, 2, 1),

(2, 3, -1),

(-1, -1, 0),

])

def test\_subtract(a, b, expected):

assert subtract(a, b) == expected

**3. Skipping Tests**

You can skip tests or mark them as expected to fail.

# test\_calculator.py

import pytest

from calculator import add

@pytest.mark.skip(reason="Skipping this test")

def test\_addition():

assert add(1, 2) == 4

@pytest.mark.xfail(reason="Expected failure")

def test\_addition\_failure():

assert add(1, 2) == 4

**4. Custom Markers**

You can define custom markers to categorize and run tests selectively.

# test\_calculator.py

import pytest

from calculator import add

@pytest.mark.slow

def test\_slow\_addition():

import time

time.sleep(5)

assert add(1, 2) == 3

To run tests with a specific marker:

pytestpython -m slow

**Summary**

* **Install pytest**: Use pip install pytest.
* **Write Tests**: Create test files and use plain assert statements or pytest fixtures and decorators.
* **Run Tests**: Execute with pytest in the terminal.
* **Advanced Features**: Utilize fixtures, parameterization, skipping, and custom markers for more complex testing scenarios.

pytest is a versatile and powerful testing tool that simplifies the testing process and provides extensive features for various testing needs.

# UnitTest vs PyTest

pytest and unittest are different testing frameworks in , each with its own features, advantages, and use cases. Here’s a detailed comparison:

**unittest**

**Overview**

* **Standard Library**: Part of ’s standard library, inspired by JUnit.
* **Class-Based**: Tests are written within test case classes that inherit from unittest.TestCase.
* **Test Discovery**: Discovers and runs test cases based on specific naming conventions.

**Features**

* **Test Fixtures**: Uses setUp and tearDown methods for setup and cleanup before and after tests.
* **Assertions**: Provides a wide range of assert methods for checking test conditions (assertEqual, assertTrue, assertIn, etc.).
* **Test Suites**: Groups of test cases can be combined into test suites.
* **Skipping Tests**: Supports skipping tests and expected failures with decorators.

**Example**

import unittest

class TestExample(unittest.TestCase):

def setUp(self):

self.value = 10

def tearDown(self):

pass

def test\_addition(self):

self.assertEqual(self.value + 5, 15)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**pytest**

**Overview**

* **Third-Party Library**: Needs to be installed separately (pip install pytest).
* **Function-Based**: Tests can be written as simple functions without the need for classes.
* **Advanced Features**: Provides fixtures, parameterized tests, and plugins for extended functionality.

**Features**

* **Simple Syntax**: Tests are written as functions with plain assert statements.
* **Fixtures**: Provides a powerful and flexible fixture system with @pytest.fixture decorator.
* **Parameterized Tests**: Easily write parameterized tests with @pytest.mark.parametrize.
* **Plugins and Extensions**: Supports a wide range of plugins for additional functionalities.
* **Detailed Output**: Provides detailed and readable output for test results.
* **Built-in Assert Helpers**: Enhanced assert introspection for better error messages.

**Example**

import pytest

@pytest.fixture

def setup\_value():

return 10

def test\_addition(setup\_value):

assert setup\_value + 5 == 15

@pytest.mark.parametrize("a, b, expected", [

(2, 3, 5),

(-1, 1, 0),

(-1, -1, -2),

])

def test\_addition\_param(a, b, expected):

assert a + b == expected

**Key Differences**

1. **Ease of Use**:
   * unittest: Requires boilerplate code with classes and methods. More verbose.
   * pytest: Simpler, function-based syntax with less boilerplate. More concise.
2. **Fixtures**:
   * unittest: Uses setUp and tearDown methods for setup and cleanup.
   * pytest: Uses @pytest.fixture decorator, which is more flexible and reusable.
3. **Parameterization**:
   * unittest: Lacks built-in parameterization support. Requires additional libraries or custom code.
   * pytest: Built-in support for parameterized tests using @pytest.mark.parametrize.
4. **Plugins and Extensibility**:
   * unittest: Basic extensibility; fewer built-in plugins.
   * pytest: Rich ecosystem of plugins and extensions for various needs (e.g., coverage, mocking, fixtures).
5. **Assertions**:
   * unittest: Uses self.assert\* methods for assertions.
   * pytest: Uses plain assert statements with detailed introspection for better error messages.
6. **Test Discovery**:
   * unittest: Discovers test cases in files named test\*.py and methods starting with test.
   * pytest: Discovers test cases in any file named test\_\*.py or in any function/method starting with test.

**When to Use Each Framework**

* **Use unittest**:
  + When you prefer a class-based approach or are already familiar with JUnit-style testing.
  + When you need a framework that comes with ’s standard library without external dependencies.
  + For projects where the test cases fit well into the structured approach provided by unittest.
* **Use pytest**:
  + When you want a more flexible, concise, and powerful testing framework.
  + When you need advanced features like fixtures, parameterized tests, and detailed test output.
  + For larger projects that may benefit from the extensive plugin ecosystem and customizability.
  + When migrating existing tests written in other frameworks to a more modern and readable format.

**Summary**

* unittest is part of the standard library and provides a structured, class-based approach to testing.
* pytest is a third-party library offering a more flexible, concise, and feature-rich testing experience.
* Both frameworks are capable and widely used, but pytest is often preferred for its simplicity, readability, and powerful features.

# PyTest-BDD

**Behavior Driven Development (BDD)**

**Behavior Driven Development (BDD)** is a software development methodology that extends Test Driven Development (TDD) by emphasizing collaboration between developers, QA, and non-technical stakeholders to ensure the software meets business requirements. It focuses on the behavior of an application from the end-user’s perspective.

**Key Concepts**

1. **User Stories**: Describe features from the user's perspective, often written in a specific format:
   * **As a [role]**, I want [feature] so that [benefit].
2. **Scenarios**: Define specific examples of how a feature should behave. Scenarios are written in plain language and include:
   * **Given**: The initial context or state.
   * **When**: The action or event that occurs.
   * **Then**: The expected outcome or result.
3. **Gherkin Syntax**: A language used to write scenarios in a readable and structured format. It helps ensure that scenarios are understandable to all stakeholders.
4. **Feature Files**: Files containing scenarios written in Gherkin syntax, usually with a .feature extension.

**Testing with pytest-bdd**

**pytest-bdd** is a plugin for pytest that integrates BDD with pytest. It allows you to write BDD scenarios using Gherkin syntax and execute them with pytest.

**Installing pytest-bdd**

To get started with pytest-bdd, you need to install it:

pip install pytest-bdd

**Example of Using pytest-bdd**

Let’s walk through a simple example where we develop a feature for a calculator application.

**1. Define the Feature**

Create a feature file named calculator.feature:

gherkin

# calculator.feature

Feature: Basic arithmetic operations

Scenario: Adding two numbers

Given I have numbers 2 and 3

When I add them

Then the result should be 5

Scenario: Subtracting two numbers

Given I have numbers 5 and 3

When I subtract them

Then the result should be 2

**2. Write the Step Definitions**

Create a file named test\_calculator.py for the step definitions:

# test\_calculator.py

import pytest

from pytest\_bdd import scenarios, given, when, then

# Import functions to be tested

from calculator import add, subtract

# Specify the feature file

scenarios('calculator.feature')

# Step Definitions

@given('I have numbers <a> and <b>')

def step\_given\_numbers(a, b):

return int(a), int(b)

@when('I add them')

def step\_when\_add(numbers):

a, b = numbers

result = add(a, b)

return result

@when('I subtract them')

def step\_when\_subtract(numbers):

a, b = numbers

result = subtract(a, b)

return result

@then('the result should be <expected>')

def step\_then\_result(result, expected):

assert result == int(expected)

**3. Running the Tests**

Run the tests using pytest:

pytest

pytest-bdd will read the feature file and execute the corresponding step definitions.

**Key Components in pytest-bdd**

1. **Feature Files**: Written in Gherkin syntax and stored in .feature files. They describe the behavior and scenarios.
2. **Step Definitions**: Implement the steps defined in the feature files. They are written in and are linked to the Gherkin steps.
3. **Scenarios**: Test cases defined in feature files that describe the expected behavior of the application.
4. **Hooks**: Optional functions that can be used to execute setup or teardown code before or after scenarios.

**Summary**

* **Behavior Driven Development (BDD)**: Focuses on the behavior of an application from the end-user's perspective, using user stories and scenarios written in plain language.
* **pytest-bdd**: A plugin for pytest that integrates BDD with pytest, allowing you to write scenarios in Gherkin syntax and execute them as tests.
* **Feature Files**: Describe behavior in a human-readable format using Gherkin syntax.
* **Step Definitions**: Implement the behavior described in feature files and connect them to the code.
* **Running Tests**: Use pytest to execute BDD scenarios defined in feature files.

BDD with pytest-bdd helps bridge the communication gap between technical and non-technical stakeholders and ensures that software development aligns closely with user expectations.