**Pytest Framework Overview**

**Pytest** is a popular Python testing framework known for its simplicity and flexibility. It is widely used for writing and running test cases because it supports both simple unit tests and complex functional testing. It provides features like fixture management, parameterized testing, and support for plugins.

**Key Features of Pytest**

1. **Ease of Use**: Simple syntax for writing tests.
2. **Fixtures**: Powerful dependency injection for test data or setup/teardown logic.
3. **Parameterized Testing**: Simplifies writing tests with multiple sets of inputs.
4. **Assertions**: Enhanced assertion introspection for better debugging.
5. **Plugins**: Extensible with a rich set of plugins.
6. **Test Discovery**: Automatically finds tests based on the test\_ prefix.

**Installing Pytest**

To install pytest, use pip:

pip install pytest

1. **Basic Unit Test**

# test\_basic.py

def multiply(a, b):

    return a \* b

def test\_multiply():

    assert multiply(2, 3) == 6

    assert multiply(0, 5) == 0

    assert multiply(-1, 4) == -4

**2. Testing Exceptions**

# test\_exceptions.py

def divide(a, b):

    if b == 0:

        raise ValueError("Division by zero is not allowed")

    return a / b

def test\_divide():

    with pytest.raises(ValueError, match="Division by zero is not allowed"):

        divide(1, 0)

### Grouping Tests in Pytest

In pytest, you can group tests to organize your test suite better and improve maintainability. Tests can be grouped by **files**, **functions**, **classes**, or using **markers**.

### 1. ****Grouping by Test Files****

Simply organize tests into different files for logical separation.

#### File Structure:

tests/

├── test\_math\_operations.py

├── test\_string\_operations.py

#### File: test\_math\_operations.py

def test\_add():

    assert 1 + 1 == 2

def test\_subtract():

    assert 5 - 3 == 2

#### File: test\_string\_operations.py

def test\_uppercase():

    assert "hello".upper() == "HELLO"

def test\_split():

    assert "hello world".split() == ["hello", "world"]

Run all tests:

pytest

Run a specific file:

pytest tests/test\_math\_operations.py

### 2. ****Grouping by Test Classes****

You can use classes to group related tests logically. Classes should not have an \_\_init\_\_ method when used for pytest tests.

#### File: test\_group\_classes.py

class TestMathOperations:

    def test\_add(self):

        assert 2 + 3 == 5

    def test\_subtract(self):

        assert 7 - 4 == 3

class TestStringOperations:

    def test\_uppercase(self):

        assert "pytest".upper() == "PYTEST"

    def test\_split(self):

        assert "a b c".split() == ["a", "b", "c"]

Run all tests in a class:

pytest test\_group\_classes.py::TestMathOperations

### 3. ****Grouping by Markers****

Markers allow you to label and run specific groups of tests.

#### File: test\_group\_markers.py

import pytest

@pytest.mark.math

def test\_add():

    assert 1 + 1 == 2

@pytest.mark.math

def test\_multiply():

    assert 2 \* 3 == 6

@pytest.mark.string

def test\_uppercase():

    assert "hello".upper() == "HELLO"

@pytest.mark.string

def test\_split():

    assert "hello world".split() == ["hello", "world"]

Run only math tests:

pytest -m math

Run only string tests:

pytest -m string

**Note**: To register custom markers, add them to pytest.ini:

# pytest.ini

[pytest]

markers =

math: Tests related to mathematical operations

string: Tests related to string operations

### 4. ****Grouping with Test Classes and Fixtures****

Combine classes and fixtures for better setup and teardown management.

#### File: test\_classes\_fixtures.py

import pytest

@pytest.fixture

def setup\_data():

    return {"key": "value"}

class TestAPI:

    def test\_get\_data(self, setup\_data):

        assert setup\_data["key"] == "value"

    def test\_post\_data(self, setup\_data):

        setup\_data["new\_key"] = "new\_value"

        assert "new\_key" in setup\_data

### 5. ****Grouping by Directory Hierarchy****

Use directories to group related test files. For example:

tests/

├── math/

│ ├── test\_addition.py

│ ├── test\_subtraction.py

├── string/

│ ├── test\_uppercase.py

│ ├── test\_split.py

Run tests in a specific directory:

pytest tests/math/

### Summary of Commands for Grouping

* **Run all tests**:

pytest

* **Run tests in a specific file**:

pytest test\_math\_operations.py

* **Run tests in a specific class**:

pytest test\_group\_classes.py::TestMathOperations

* **Run tests by marker**:

pytest -m math

* **Run tests in a specific directory**:

pytest tests/math/

### Predefined Markers in Pytest

Pytest comes with several **predefined markers** to provide functionality like skipping tests, expecting tests to fail, or setting conditional execution. These markers are built into pytest and help handle various testing scenarios.

### Common Predefined Markers

1. **@pytest.mark.skip**  
   Skip a test unconditionally.

import pytest

@pytest.mark.skip(reason="This test is not implemented yet")

def test\_skip\_example():

    assert 1 == 2

**Run**:

pytest -v

**Output**:

swift

SKIPPED [1] test\_skip.py: This test is not implemented yet

1. **@pytest.mark.skipif**  
   Skip a test conditionally based on a condition.

import pytest

import sys

@pytest.mark.skipif(sys.version\_info < (3, 8), reason="Requires Python 3.8 or higher")

def test\_skipif\_example():

    assert 1 == 1

**Run on Python < 3.8**:

SKIPPED [1] test\_skipif.py: Requires Python 3.8 or higher

1. **@pytest.mark.xfail**  
   Mark a test as expected to fail.

import pytest

@pytest.mark.xfail(reason="Known bug, fixing in next release")

def test\_xfail\_example():

    assert 1 == 2

**Output**:

arduino

XFAIL test\_xfail.py::test\_xfail\_example

If the test passes unexpectedly:

XPASS test\_xfail.py::test\_xfail\_example

1. **@pytest.mark.parametrize**  
   Run the same test with multiple sets of parameters.

import pytest

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

    (1, 2, 3),

    (0, 0, 0),

    (-1, -1, -2)

])

def test\_parametrize\_example(a, b, expected):

    assert a + b == expected

**Output**:

test\_parametrize.py::test\_parametrize\_example[1-2-3] PASSED

test\_parametrize.py::test\_parametrize\_example[0-0-0] PASSED

test\_parametrize.py::test\_parametrize\_example[-1--1--2] PASSED

1. **@pytest.mark.usefixtures**  
   Apply a fixture to a test function or class.

import pytest

@pytest.fixture

def setup\_data():

    return {"key": "value"}

@pytest.mark.usefixtures("setup\_data")

def test\_fixture\_example(setup\_data):

    assert setup\_data["key"] == "value"

1. **@pytest.mark.filterwarnings**  
   Control warnings during test execution.

import warnings

import pytest

@pytest.mark.filterwarnings("ignore:.\*deprecated.\*")

def test\_warnings():

    warnings.warn("This is a deprecated feature", DeprecationWarning)

    assert True

### Running Tests with Predefined Markers

#### To List All Available Markers

pytest --markers

**Example Output**:

@pytest.mark.skip(reason): skip the test with an optional reason

@pytest.mark.skipif(condition, reason): skip the test if condition is True

@pytest.mark.xfail(condition, reason, ...): mark the test as expected to fail

### Additional Options for Predefined Markers

* **Run tests marked as xfail only**:

pytest -rx

* **Show reasons for skipped tests**:

pytest -rs

* **Combine markers**:

@pytest.mark.skipif(condition, reason="reason")

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

def test\_combined\_marker():

assert False

**3. Using Fixtures for Test Setup**

# test\_fixtures.py

import pytest

@pytest.fixture

def sample\_user():

    return {"username": "test\_user", "email": "test@example.com"}

def test\_user\_email(sample\_user):

    assert sample\_user["email"] == "test@example.com"

### ****What is a Fixture in Pytest?****

A **fixture** in pytest is a function used to provide a fixed baseline for tests. Fixtures allow you to set up prerequisites or shared resources needed for your tests, such as test data, database connections, or configurations. They can be reused across multiple test functions and provide a way to manage setup and teardown logic efficiently.

### ****Key Features of Fixtures****

1. **Reusable**: Defined once and used across multiple tests.
2. **Flexible Scope**: Can be applied at different levels (function, class, module, or session).
3. **Dependency Injection**: Automatically passed to the tests that need them.
4. **Automatic Teardown**: Cleanup logic can be implemented for resources after tests run.

### ****Creating and Using Fixtures****

#### Example: Basic Fixture

import pytest

@pytest.fixture

def sample\_data():

    return {"key": "value"}

def test\_sample\_data(sample\_data):

    assert sample\_data["key"] == "value"

* The sample\_data fixture provides a dictionary for the test function.
* Pytest automatically passes the sample\_data fixture to the test function when requested.

### ****Fixture Scope****

The default scope of a fixture is function, meaning the fixture is created and torn down for each test function. Other scopes include:

1. **function**: (Default) Fixture runs for each test function.
2. **class**: Fixture runs once per test class.
3. **module**: Fixture runs once per module.
4. **session**: Fixture runs once for the entire test session.

#### Example: Fixture with Different Scopes

import pytest

@pytest.fixture(scope="module")

def setup\_once():

    return "Module Level Setup"

def test\_1(setup\_once):

    assert setup\_once == "Module Level Setup"

def test\_2(setup\_once):

    assert setup\_once == "Module Level Setup"

### ****Using Fixtures for Setup and Teardown****

Fixtures can include setup and teardown logic using yield.

#### Example: Setup and Teardown

import pytest

@pytest.fixture

def resource\_setup():

    print("Setting up resource")

    resource = {"status": "ready"}

    yield resource

    print("Tearing down resource")

def test\_resource(resource\_setup):

    assert resource\_setup["status"] == "ready"

**Output**:

Setting up resource

Tearing down resource

* The code before yield is executed before the test.
* The code after yield is executed after the test, ensuring proper cleanup.

### ****Parameterizing Fixtures****

Fixtures can be parameterized to run tests with different sets of data.

#### Example: Parameterized Fixture

import pytest

@pytest.fixture(params=[1, 2, 3])

def numbers(request):

    return request.param

def test\_numbers(numbers):

    assert numbers in [1, 2, 3]

* The test runs three times, once for each value in the params list.

### ****Autouse Fixtures****

Fixtures can be set to run automatically for all tests without explicitly requesting them using autouse=True.

#### Example: Autouse Fixture

import pytest

@pytest.fixture(autouse=True)

def always\_run():

    print("This runs before every test")

def test\_example():

    assert True

### ****Fixtures with Dependencies****

Fixtures can depend on other fixtures, creating a chain of setup logic.

#### Example: Fixture Dependencies

import pytest

@pytest.fixture

def user\_data():

    return {"username": "test\_user"}

@pytest.fixture

def logged\_in\_user(user\_data):

    user\_data["logged\_in"] = True

    return user\_data

def test\_logged\_in\_user(logged\_in\_user):

    assert logged\_in\_user["logged\_in"]

    assert logged\_in\_user["username"] == "test\_user"

### ****Common Use Cases for Fixtures****

1. **Database Setup**:

@pytest.fixture

def db\_connection():

conn = connect\_to\_database()

yield conn

conn.close()

1. **Temporary File/Directory**:

@pytest.fixture

def temp\_file(tmp\_path):

file = tmp\_path / "test\_file.txt"

file.write\_text("Hello, pytest!")

return file

1. **API Mocking**:

@pytest.fixture

def mock\_api(mocker):

mock = mocker.patch("api\_client.get\_data")

mock.return\_value = {"key": "value"}

return mock

### ****Summary****

* **Fixtures** are essential for managing repetitive setup/teardown tasks in pytest.
* They simplify test logic by providing reusable and clean setups.
* With features like **parameterization**, **scoping**, and **dependencies**, they enable scalable and maintainable testing.

**What is conftest.py in Pytest?**

conftest.py is a special configuration file used in the pytest testing framework. It serves as a central location for defining **fixtures**, **hooks**, and other test configurations that can be shared across multiple test files in a project. The key advantage is that it allows test files to use these configurations without explicitly importing them.

**Features of conftest.py**

1. **Fixture Sharing**: Define reusable fixtures that are automatically discovered by pytest.
2. **Directory Scope**: The configurations in conftest.py apply to all test files in the same directory and its subdirectories.
3. **Test Hooks**: Customize pytest behavior using hooks, such as adding command-line options or modifying the test collection process.
4. **No Explicit Import**: Test files within the scope of conftest.py can use its fixtures and configurations without needing to import them explicitly.

### ****Example: Using**** conftest.py ****for Fixtures****

#### Directory Structure

tests/

├── conftest.py

├── test\_example.py

├── subdir/

│ ├── conftest.py

│ ├── test\_sub\_example.py

#### conftest.py in Root

import pytest

@pytest.fixture

def sample\_data():

    return {"key": "value"}

#### test\_example.py

def test\_sample\_data(sample\_data):

    assert sample\_data["key"] == "value"

#### subdir/conftest.py

import pytest

@pytest.fixture

def subdir\_data():

    return {"subdir\_key": "subdir\_value"}

#### subdir/test\_sub\_example.py

def test\_subdir\_data(subdir\_data):

    assert subdir\_data["subdir\_key"] == "subdir\_value"

### ****Scoping Fixtures in**** conftest.py

Fixtures in conftest.py can have a broader scope (module, session, etc.) to avoid being recreated for every test.

#### Example: Session Scoped Fixture

# conftest.py

import pytest

@pytest.fixture(scope="session")

def database\_connection():

    print("\nSetting up database connection")

    conn = {"db": "connected"}

    yield conn

    print("\nTearing down database connection")

#### test\_database.py

def test\_database\_connection\_1(database\_connection):

    assert database\_connection["db"] == "connected"

def test\_database\_connection\_2(database\_connection):

    assert database\_connection["db"] == "connected"

**Output**:

Setting up database connection

Tearing down database connection

The fixture is set up only once for the session and reused across tests.

### ****Using Hooks in**** conftest.py

You can define pytest hooks in conftest.py to modify or customize pytest’s behavior.

#### Example: Modifying Command-Line Options

python

Copy code

# conftest.py

def pytest\_addoption(parser):

parser.addoption("--env", action="store", default="dev", help="Environment to run tests against")

@pytest.fixture

def env(pytestconfig):

return pytestconfig.getoption("--env")

#### test\_environment.py

python

Copy code

def test\_environment(env):

assert env in ["dev", "staging", "prod"]

**Run the test**:

bash

Copy code

pytest --env=staging

### ****Using**** conftest.py ****in Subdirectories****

conftest.py files in subdirectories override or supplement those in parent directories.

#### Directory Structure

Copy code

tests/

├── conftest.py

├── test\_example.py

├── subdir/

│ ├── conftest.py

│ ├── test\_sub\_example.py

* tests/conftest.py: Fixtures and hooks for all tests.
* tests/subdir/conftest.py: Additional fixtures/hooks only for tests in subdir.

### ****Best Practices for Using**** conftest.py

1. **Organize logically**:
   * Use conftest.py for shared fixtures or hooks relevant to a specific directory.
   * Avoid putting too many unrelated fixtures in one conftest.py.
2. **Use clear names**:
   * Name fixtures descriptively to indicate their purpose.
3. **Limit scope**:
   * Use appropriate fixture scopes (function, module, session) to balance test isolation and performance.
4. **Avoid imports from conftest.py**:
   * Fixtures in conftest.py should be implicitly available; importing them in test files defeats the purpose.