Python

Full stack Skills Bootcamp

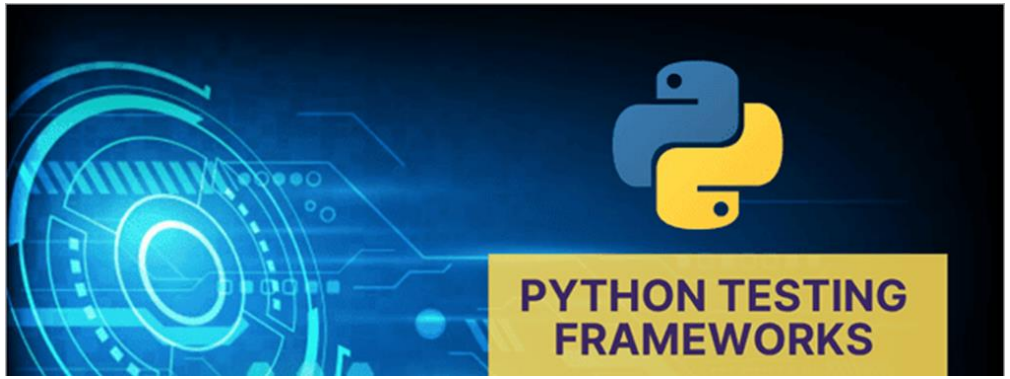
Introducing Test Frameworks

Overview

Python offers several frameworks for testing, with two of the most popular being:

- unittest: Built-in, classic testing framework.
- pytest: More modern, flexible, and powerful.

We'll focus on pytest for our examples due to its ease of use and rich features



Installing pytest

- To get started with pytest, you need to install it first.
- Installation can be done easily using pip.

```bash
pip install pytest
```

Once installed, you can run tests using the command line!

Running Basic Test Cases

After writing your tests, you can run them with pytest from the command line.

- By default, pytest will search for files that start with test_ or end with _test.py and execute any test functions found within.

```bash
pytest <test_file_name>.py
```

If you have a file named test_example.py, simply run “pytest test_example.py”

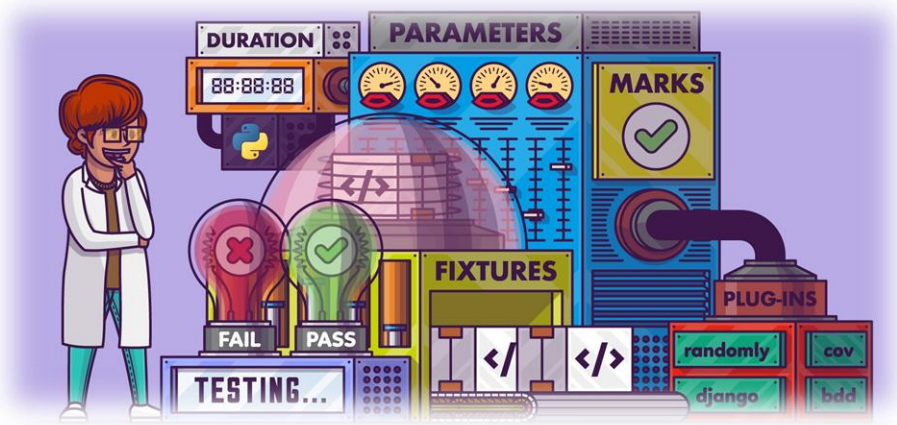
Writing Test Functions Using pytest Conventions

- Writing test functions with pytest is straightforward. Just define a function that starts with test_.:

Example:

```python
def test_add():
    assert add(2, 3) == 5  # Simple test case for addition
```

In this example, if add(2, 3) returns 5, the test passes; otherwise, it fails.



Examples

```python
def test_add_negative_numbers():
    assert add(-1, -1) == -2  # Testing addition of negative numbers

def test_add_with_zero():
    assert add(0, 5) == 5  # Testing addition with zero
```

- You can write tests for different scenarios, including edge cases and error handling.
- This way, you ensure that your function works for a variety of inputs

Python Fixtures

What Are Python Fixtures ?

- Fixtures are functions in pytest that run before tests to set up the required environment.
- They help you reuse setup code across multiple tests.
- You can think of them as pre-test setup or post-test teardown routines.

Example

Here’s how we define a simple fixture using pytest :

The sample_data fixture returns a list, and we reuse that in the test_sum function.

```python
import pytest

@pytest.fixture
def sample_data():
    return [1, 2, 3, 4, 5]

def test_sum(sample_data):
    assert sum(sample_data) == 15
```

Example

You can use more than one fixture in a test, and pytest will handle the setup automatically

Here, two fixtures' data_a and data_b are used in the same test function.

```python
@pytest.fixture
def data_a():
    return [1, 2, 3]

@pytest.fixture
def data_b():
    return [4, 5, 6]

def test_combined_data(data_a, data_b):
    assert sum(data_a + data_b) == 21
```

Fixture Scope and Autouse

- Fixture Scope: You can specify how often a fixture runs using scopes like function, class, module, or session.
- Autouse: Fixtures can automatically be applied to all tests without being explicitly called.

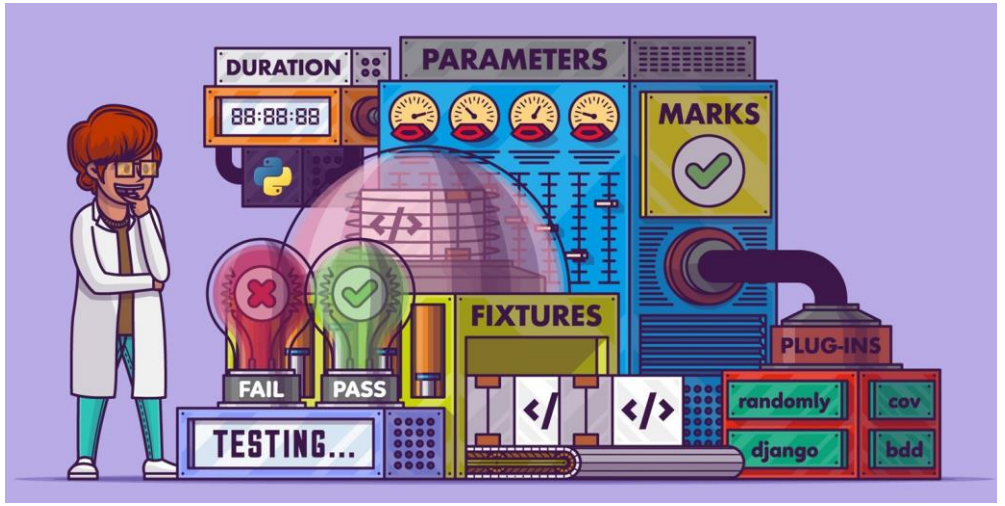
```python
@pytest.fixture(scope="module", autouse=True)
def setup_module():
    print("Setting up module...")

# This fixture will run before any tests in the module.
```

Python Test Parametrization

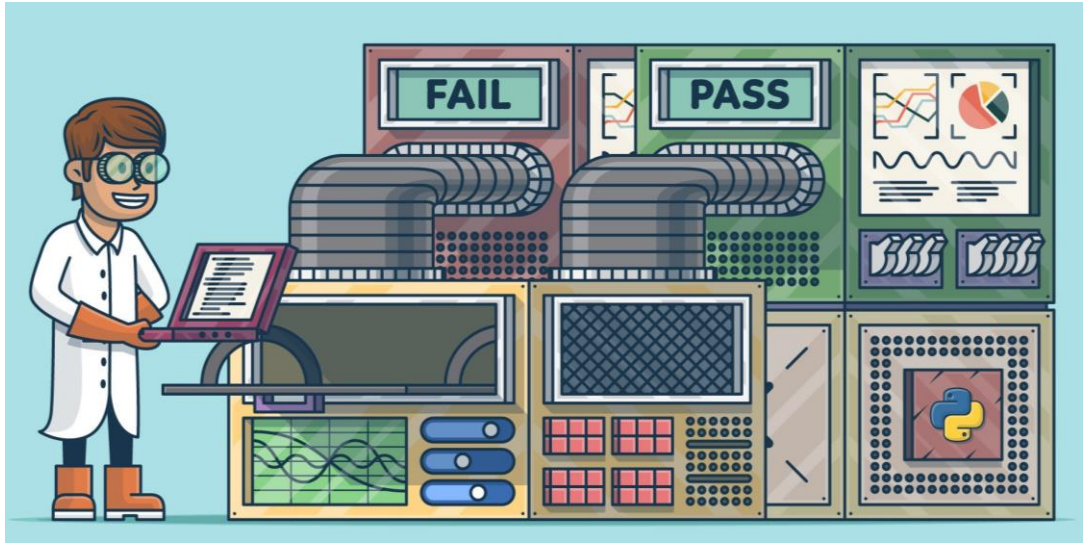
What is Test Parametrization?

- Test parametrization allows you to run a single test function with different sets of inputs.
- It helps in reducing code duplication and makes tests cleaner.
- Using parameterized tests, you can easily cover multiple scenarios without writing repetitive test code.



Why Use Test Parametrization?

- **Efficiency**: Write less code while covering more test cases.
- **Clarity**: Each test case is clear and specific, helping to document behaviour.
- **Maintenance**: Easier to add new test cases by simply adding more parameters.



Example

Here’s a simple example using pytest.mark.parametrize:

This test will run three times with different input values!

```python
import pytest

@pytest.mark.parametrize("input,expected", [
    (1, 2),   # Test case 1
    (2, 3),   # Test case 2
    (3, 4),   # Test case 3
])
def test_increment(input, expected):
    assert input + 1 == expected
```

Example

You can also test functions with multiple parameters. Here's an example:

This tests the addition of two numbers, validating several scenarios in a single test function.

```python
@pytest.mark.parametrize("a,b,expected", [
    (2, 3, 5),   # Test case 1
    (0, 0, 0),   # Test case 2
    (-1, 1, 0),  # Test case 3
])
def test_addition(a, b, expected):
    assert a + b == expected
```

Advanced Pytest Techniques

Overview

- Advanced pytest techniques help improve the quality and maintainability of your tests.

Key techniques include:
- Grouping tests
- Handling exceptions
- Running coverage reports

These practices ensure your tests are effective and your code is complete.

Example

Let’s start with a simple function using inbuilt functions that we’ll be testing.

```python
def divide(a, b):
    """Function to divide two numbers. Raises an error for division by zero."""
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
```

- This function will help us demonstrate various pytest techniques, particularly in error handling and validation.

Example

Grouping Tests with Parametrization

- Grouping tests allows us to organize related tests together, improving clarity and structure.
- Here’s an example of grouping tests using pytest.mark.parametrize:

```python
@pytest.mark.parametrize("input_a, input_b, expected", [
    (6, 3, 2),   # Test case 1
    (10, 2, 5),  # Test case 2
    (15, 5, 3),  # Test case 3
])
def test_divide(input_a, input_b, expected):
    """Test the divide function with valid parameters."""
    result = divide(input_a, input_b)
    assert result == expected
```

This approach keeps related tests together, making it easy to see what is being validated.

Example

Handling exceptions is crucial for validating edge cases. Let's test our divide function for division by zero using pytest

```python
def test_divide_by_zero():
    """Test the divide function for division by zero."""
    with pytest.raises(ValueError, match="Cannot divide by zero"):
        divide(10, 0)
```

- Using pytest.raises, we can assert that our function raises the expected error, which enhances our test coverage.

Example

Grouping Tests in a Class

Another way to group tests is by using test classes. This allows you to organize related tests under a common structure:

```python
class TestMathOperations:

    @pytest.mark.parametrize("input_a, input_b, expected", [
        (20, 4, 5),     # Test case 1
        (100, 10, 10),  # Test case 2
    ])
    def test_divide_grouped(self, input_a, input_b, expected):
        """Grouped test for the divide function."""
        assert divide(input_a, input_b) == expected

    def test_divide_by_zero_grouped(self):
        """Grouped test for division by zero."""
        with pytest.raises(ValueError, match="Cannot divide by zero"):
            divide(1, 0)
```

This structure makes it easy to manage related tests and maintain organization in your testing suite.

Recap

Advanced techniques in pytest can significantly enhance your testing process.

Key Takeaways:

- Grouping tests improves organization.
- Handling exceptions ensures robustness.
- Coverage reports help verify code completeness.

By implementing these practices, you'll create a more reliable testing environment!.