ROADMAP

Topics to Know About Pytest:

1. **Introduction to pytest**
    - What is pytest?
    - Why use pytest?
2. **Installing and Setting Up pytest**
    - Installation
    - Directory structure for tests
3. **Writing Basic Tests**
    - Creating test files and functions
    - Running tests
4. **Assertions**
    - Understanding pytest's assert statement
    - Common assertion examples
5. **Fixtures**
    - What are fixtures?
    - Using `@pytest.fixture` for setup and teardown
6. **Parametrized Tests**
    - Testing with multiple input combinations
7. **Markers**
    - Adding metadata to tests (e.g., skip, xfail)
    - Custom markers
8. **Organizing Tests**
    - Grouping and structuring test cases
    - Test discovery
9. **Command-Line Options**
    - Running specific tests
    - Verbosity and test reports
10. **Mocking**
    - Using `unittest.mock` or `pytest-mock`
    - Patching external dependencies
11. **Plugins and Extensions**
    - Popular pytest plugins like `pytest-html` and `pytest-cov`
    - Writing custom plugins
12. **Integrating with CI/CD**
    - Running pytest in continuous integration pipelines
13. **Testing Frameworks Integration**

- Using pytest with frameworks like Flask, Django, etc.

14. **Customizing pytest**

- Configuring pytest using `pytest.ini` or `pyproject.toml`

NOTES:

n pytest, `call.excinfo` provides information about any exception that occurred during the execution of a test case. It is part of the `pytest_runtest_makereport` hook and is available when the `call.when` attribute is `"call"` (the test execution phase).

- **If no exception occurs**: `call.excinfo` is `None`, meaning the test passed.
- **If an exception occurs**: `call.excinfo` contains details about the exception, such as its type, value, and tracebac

Using `@pytest.mark.parametrize` allows you to test a function with **multiple sets of input values** in a single test function. It saves you from writing separate test functions for each input combination.

How It Works:

1. `@pytest.mark.parametrize` **decorator**:
    - Takes two arguments:
        - A string of parameter names (comma-separated).
        - A list of tuples where each tuple contains a set of input values.
2. **Test function**:
    - The parameters are passed to the test function, which runs once for each tuple in the list.

```
import pytest

# Function to test
def add(a, b):
    return a + b

# Parametrize the test function
@pytest.mark.parametrize("a, b, expected", [
    (1, 2, 3),   # First set of inputs
    (2, 3, 5),   # Second set of inputs
    (-1, 1, 0),  # Third set of inputs
    (0, 0, 0)    # Fourth set of inputs
])
def test_add(a, b, expected):
    assert add(a, b) == expected
```

What Are Fixtures in Pytest?

Fixtures in pytest are functions used to set up the required state or environment before running a test. They help manage repetitive tasks like creating test data, initializing a database connection, or setting up configuration. Fixtures can also clean up after the test is done.

Why Use Fixtures?

- **Reusable**: Fixtures can be shared across multiple test cases.
- **Automatic Setup and Teardown**: They prepare the required resources before a test and clean up afterward.
- **Better Organization**: Reduce duplication and make your tests cleaner.

How Fixtures Work:

1. You define a fixture using the `@pytest.fixture` decorator.
2. The fixture can be **used in tests by passing its name** as a parameter.
3. Pytest automatically runs the fixture before executing the test

```
import pytest

# Define a fixture
@pytest.fixture
def test_data():
    return {"a": 5, "b": 3}

# Use the fixture in a test
def test_addition(test_data):
    result = test_data["a"] + test_data["b"]
    assert result == 8
```

What Happens:

1. `@pytest.fixture`: Marks `test_data` as a fixture.
2. `test_addition(test_data)`: Pytest automatically calls the `test_data` fixture and provides its return value to the test.
3. **Output**: The test uses `a=5` and `b=3` from the fixture.

Fixtures with Parameters

You can parameterize fixtures to provide different values to tests.

Example: Parameterized Fixture

```
@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 `[1, 2, 3]`.

Scope of Fixtures

Fixtures can have different scopes to control how often they are executed:

- `function` **(default)**: Run once per test.
- `class`: Run once per test class.
- `module`: Run once per module (file).
- `session`: Run once per test session.

`pytest_runtest_makereport`

Purpose:

This hook is called after each test phase (setup, call, teardown) and provides detailed information about the test, such as:

- Whether the test passed, failed, or was skipped.
- Any exception details if the test failed.

Parameters:

- `item`: The test function being executed.
- `call`: Information about the current phase (`setup`, `call`, or `teardown`).

```
import pytest

def pytest_runtest_makereport(item, call):
    if call.when == "call":  # Only focus on the actual test phase
        outcome = "passed" if call.excinfo is None else "failed"
        # Attach the outcome to the test item for later use
        setattr(item, "outcome", outcome)
```

How It Works:

- `call.when` determines the phase (`setup`, `call`, or `teardown`).
- If `call.excinfo` is `None`, the test passed; otherwise, it failed.
- The test outcome is attached to the `item` object for further processing.

`pytest_sessionfinish`

Purpose:

This hook is executed after all tests in the session have run. It's typically used for generating summary reports or performing final cleanup.

Parameters:

- `session`: The pytest session object containing all test items and their results.
- `exitstatus`: The exit code of the test session.

Adding Metadata to Tests in Pytest

Metadata allows you to attach additional information (like author, description, priority, tags, etc.) to tests. This can be useful for organizing, categorizing, or documenting your test cases, and also for generating more informative test reports.

In pytest, you can use **markers** or **custom attributes** to add metadata.

1. Using Markers for Metadata

Markers are special annotations in pytest that allow you to label tests with metadata.

```
import pytest

@pytest.mark.metadata(author="Alice", description="Test addition functionality")
def test_add():
    assert 2 + 2 == 4

@pytest.mark.metadata(author="Bob", description="Test subtraction functionality")
def test_subtract():
    assert 5 - 3 == 2

Accessing Metadata in Hooks
def pytest_runtest_makereport(item, call):
    if call.when == "call":
        outcome = "passed" if call.excinfo is None else "failed"
        metadata = item.get_closest_marker("metadata")
        if metadata:
            print(f"Test '{item.name}' Metadata:")
            for key, value in metadata.kwargs.items():
                print(f"  {key}: {value}")

Using Custom Attributes for Metadata

def test_add():
    test_add.author = "Alice"
    test_add.description = "Test addition functionality"
    assert 2 + 2 == 4

def test_subtract():
    test_subtract.author = "Bob"
    test_subtract.description = "Test subtraction functionality"
    assert 5 - 3 == 2

def pytest_runtest_makereport(item, call):
    if call.when == "call":
        outcome = "passed" if call.excinfo is None else "failed"
        author = getattr(item.function, "author", "Unknown")
        description = getattr(item.function, "description", "No description")
        print(f"Test '{item.name}' Metadata:")
        print(f"  Author: {author}")
        print(f"  Description: {description}")
```

Using a Dictionary for Metadata

If you want to keep metadata organized, you can use a dictionary.

```
@pytest.mark.metadata({"author": "Alice", "description": "Test addition"})
def test_add():
    assert 2 + 2 == 4

def pytest_runtest_makereport(item, call):
    if call.when == "call":
        metadata = item.get_closest_marker("metadata")
        if metadata:
            for key, value in metadata.args[0].items():
                print(f"{key}: {value}")
```

```
import csv

def pytest_runtest_makereport(item, call):
    if call.when == "call":
        outcome = "passed" if call.excinfo is None else "failed"
        item.outcome = outcome
        metadata = item.get_closest_marker("metadata")
        if metadata:
            item.metadata = metadata.kwargs
        else:
            item.metadata = {}

def pytest_sessionfinish(session, exitstatus):
    results = []
    for item in session.items:
        metadata = getattr(item, "metadata", {})
        result = {
            "test_name": item.name,
            "outcome": getattr(item, "outcome", "unknown"),
            **metadata
        }
        results.append(result)

    with open("test_report_with_metadata.csv", "w", newline="") as csvfile:
        fieldnames = ["test_name", "outcome", "author", "description"]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(results)
```

How to Trace Back to the Function Details

When generating reports, you can use the metadata to identify:

1. **Which function the test belongs to.**
2. **Additional details about the function, like purpose or owner.**

Summary:

- **Metadata** is tied to the test function and provides **details about the function itself**.
- For individual test inputs (in parameterized tests), the function metadata remains the same, but you can combine it with the inputs for detailed reporting.
- You can use pytest hooks to access both metadata and inputs to generate enriched reports.

Grouping Tests Using Markers

Markers let you group related tests logically, even if they are in different files.

```
import pytest

@pytest.mark.math
def test_addition():
    assert 1 + 1 == 2

@pytest.mark.math
def test_subtraction():
    assert 5 - 3 == 2

@pytest.mark.string
def test_string_concatenation():
    assert "Hello " + "World" == "Hello World"
```

--
pytest -m math

Grouping Tests Inside Classes

You can group related tests into classes using `Test` as a prefix.

```
class TestMathOperations:
    def test_addition(self):
        assert 1 + 1 == 2

    def test_subtraction(self):
        assert 5 - 3 == 2

class TestStringOperations:
    def test_concatenation(self):
        assert "Hello " + "World" == "Hello World"
```

pytest tests/test_operations.py::TestMathOperations

#Running specific test inside a python file
pytest tests/test_file.py::test_function_name

collection test list
pytest –collect-only
pytest --collect-only -v

Running Tests in Parallel

Install `pytest-xdist` to run tests in parallel:

pytest -n 4

Unit Test Mocking and Pytest Mock

Mocking is a technique used in unit testing to replace parts of your application with mock objects. These mock objects simulate the behavior of real objects, making it possible to isolate and test a specific component of your application without relying on external dependencies (like databases, APIs, or file systems).

Why Use Mocking?

1. **Isolate Testing**: Test a unit of code without executing its dependencies.
2. **Control Behavior**: Simulate various scenarios (e.g., API returns error, database is unavailable).
3. **Improve Speed**: Avoid slow external operations like network calls or database queries.
4. **Ensure Test Reliability**: Mock unpredictable behavior, such as random numbers or current timestamps.

Pytest-Mock

Pytest provides a plugin called `pytest-mock` to make mocking easier and more pytest-friendly.

```
def test_fetch_data_from_api(mocker):
    mock_get = mocker.patch("requests.get")
    mock_get.return_value.json.return_value = {"key": "value"}

    result = fetch_data_from_api("http://example.com")

    assert result == {"key": "value"}
    mock_get.assert_called_once_with("http://example.com")
```

Mocking a Function

```
# math_operations.py
def add(a, b):
    return a + b
```

```
from math_operations import add

def test_add(mocker):
    mock_add = mocker.patch("math_operations.add", return_value=10)
    result = add(3, 5)
    assert result == 10
    mock_add.assert_called_once_with(3, 5)
```

```
from math_operations import add

def test_add(mocker):
```

```
mock_add = mocker.patch("math_operations.add", return_value=10)
result = add(3, 5)
assert result == 10
mock_add.assert_called_once_with(3, 5)
```