S18 : Pytest

Pytest Testing Framework

- Install pytest : pip3 install pytest
- Naming convensions to follow for Pytest tests
- Running pytests from command line and VSCode
- Running selected test files using pytest
- Running selected test methods based on matching keywords
- Pytest Tags Mechanism to run tests based on functionality
- Failing and Skipping tests with Annotations using Pytest
- What are fixtures and importance of their Hooks in Pytest
- How fixtures can be configured in Conftest file for better readability
- Different scopes of fixtures and their related annotations to setup Pre and Post conditions on the Test
- How parameterization can be achieved for tests with multiple sets of data
- How to pass command line arguments into Pytests
- HTML report generation for pytests execution

Naming convensions to follow for Pytest tests

When writing tests in Pytest, following proper naming conventions is essential for readability, consistency, and automatic discovery of tests by the Pytest framework. Below are the recommended naming conventions to follow for Pytest tests:

1. **Test File Names:**

- Test files should be named in a way that Pytest can automatically discover them.
- **Convention**: Files should either:
  - **Start with** `test_` (e.g., `test_example.py`), or
  - **End with** `_test.py` (e.g., `example_test.py`).

This ensures Pytest recognizes them as test files when scanning your codebase.

2. **Test Function Names:**

- Test function names should be descriptive and clearly indicate the expected behavior or the functionality being tested.
- **Convention**: Test function names should **start with** `test_` (e.g., `test_addition_function` or `test_login_valid_credentials`).

By starting with `test_`, Pytest can automatically discover and run these functions.

3. **Test Class Names:**

- Although Pytest does not require test classes, if you use them to organize related tests, the class name should also follow certain conventions.
- **Convention**: Class names should start with `Test` (e.g., `TestUserLogin`, `TestMathOperations`).
  - Unlike test functions, you do **not** need to prefix each method in a test class with `test_` if the class name starts with `Test`.

Note: Pytest does not require test classes to inherit from `unittest.TestCase`.

4. Naming Parameters for Parametrized Tests:

- When using `@pytest.mark.parametrize`, ensure that parameter names are meaningful and related to the functionality being tested.
- **Example**:

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

5. Fixture Naming:

- Fixtures in Pytest provide a mechanism for setup and teardown. Fixtures should have descriptive and short names that reflect the resource they are setting up.
- **Convention**: Use snake_case for fixture names (e.g., `db_connection`, `temporary_file`).
- **Example**:

```
@pytest.fixture
def db_connection():
    return Database.connect()
```

6. Constants Naming in Tests:

- If you're defining constants or mock data within your test, they should follow Python's constant naming convention: **ALL_CAPS** with underscores to separate words.
- **Example**:

```
MAX_RETRIES = 3
VALID_USERNAME = "test_user"
```

7. Test Naming for Expected Behavior:

- Test function names should convey the expected behavior or outcome. Include the condition being tested and the expected result in the function name for clarity.
- **Convention**: Use the pattern `test_<functionality>_<expected_behavior>` (e.g.,

`test_login_success`, `test_addition_negative_numbers`, `test_search_no_results`).

8. Use Verbose Names:

- Avoid abbreviations or overly shortened names. Instead, use names that clearly describe the test's purpose.
- **Example**:
  - Prefer `test_calculate_total_with_discount()` over `test_calc_disc()`.

9. Negative and Edge Case Test Naming:

- If you're testing for failure cases or edge conditions, explicitly include that in the test name.
- **Convention**: Use words like `invalid`, `error`, `empty`, or `failure` in the test name.
- **Example**:

```
def test_login_invalid_credentials():
    pass

def test_add_empty_list():
    pass
```

10. Use Underscores Instead of CamelCase:

- Pytest prefers using `snake_case` for naming functions, files, and variables over `camelCase`, which is more common in other testing frameworks like Java's JUnit.
- **Example**:

```
def test_calculate_total_cost():
    pass
```

Summary of Naming Conventions:

- **Test files**: `test_*.py` or `*_test.py`
- **Test functions**: `test_<description_of_test_case>()`
- **Test classes**: `Test<ClassName>` (without `test_` prefix for class methods)
- **Fixtures**: Use descriptive, snake_case names
- **Constants**: ALL_CAPS with underscores
- **Be descriptive**: Include expected behaviors in names

Following these conventions ensures consistency and readability across your test suite, making your tests easier to maintain and more self-explanatory to collaborators.

Running pytests from VSCode

There are several ways to run individual tests in Visual Studio Code (VS Code) using plugins that support Pytest. One of the most popular plugins is **"Python"** by Microsoft, which includes support for running Python tests, including Pytest.

Steps to Run Single Pytest with VS Code:

1. **Install the Python Extension**:
   - Install the official **Python** extension for VS Code, which includes support for testing with Pytest.
   - Go to the Extensions view in VS Code (`Ctrl + Shift + X`), search for "Python", and install it.
2. **Configure Pytest in VS Code**:
   - Open the **command palette** (`Ctrl + Shift + P`), then search for `Python: Configure Tests` .
   - Select **Pytest** as your test framework.
   - VS Code will automatically detect test files and configure them for Pytest.
3. **Running a Single Test**: After configuring Pytest in VS Code, you can run individual tests directly from the editor.

   Method 1: Run Test with CodeLens (In-editor option)
   - When you open a test file, you'll notice small **Run Test** and **Debug Test** links above each test function. This feature is called **CodeLens**.
   - Click **Run Test** above the individual test function to run that specific test.

   Method 2: Use Testing Sidebar:
   - In the activity bar on the left-hand side, click the **Testing** icon (flask icon).
   - This will open the **Testing Panel**, where you can view all discovered tests.
   - You can select individual tests and click the play button to run them.
4. **Run Single Test from the Command Palette**:
   - Open the command palette (`Ctrl + Shift + P`), and search for `Python: Run Tests` .
   - Select a single test from the list to run it.

Example Configuration in VS Code:

In your workspace folder, you can also create a `settings.json` file to configure Pytest:

```
{
    "python.testing.pytestArgs": ["tests"],
    "python.testing.unittestEnabled": false,
    "python.testing.nosetestsEnabled": false,
    "python.testing.pytestEnabled": true
}
```

Additional Plugins:

- **Pytest Test Explorer**: If you want a more feature-rich experience, you can install the **"Pytest Test Explorer"** extension. It integrates with the **Test Explorer UI** extension, making it easier to explore, run, and debug tests from a test explorer window.

Conclusion:

The **Python** extension by Microsoft is the most commonly used VS Code plugin for running single Pytest tests. It provides excellent integration, and you can easily run individual tests using CodeLens, the Testing Sidebar, or the Command Palette.

Running pytests from command line

Running selected test files using pytest

Running selected test methods based on matching keywords

Running Pytest from the command line is simple and straightforward. You can execute your tests and even specify various options to customize how Pytest runs. Below are the basic and more advanced ways to run Pytests from the command line:

1. **Basic Command to Run All Tests**:

To run all test files in the current directory and its subdirectories:

```
pytest
```

2. **Run a Specific Test File**:

You can specify a particular test file to run:

```
pytest test_file.py
```
For example:

```
pytest test_example.py
```

3. **Run a Specific Test Function**:

To run a specific test function within a file:

```
pytest test_file.py::test_function
```
For example:

```
pytest test_example.py::test_addition_function
```

4. **Run Tests in a Specific Directory**:

To run all tests in a specific directory:

```
pytest path/to/directory/
```

5. Run Tests Matching a Pattern:

You can run tests that match a specific pattern. For instance, to run all tests whose filenames start with `test_add`:

```
pytest -k "test_add"
```
The `-k` option allows you to select tests based on their names, filtering out others.

6. Run Tests Verbosely:

For more detailed output, use the `-v` (verbose) flag:

```
pytest -v
```

7. Run Tests and Show Output for Print Statements:

If you want to see print statements from your tests in the console output, use the `-s` flag:

```
pytest -s
```

8. Run Tests with Detailed Failures:

To show a more detailed report of failures (with the entire stack trace), use the `-vv` flag (extra verbose):

```
pytest -vv
```

9. Stop after First Failure:

To stop the test run after the first failure, use the `-x` flag:

```
pytest -x
```

10. Run Tests in Parallel:

If you want to run tests in parallel, you can use the `pytest-xdist` plugin with the `-n` flag to specify the number of processes:

```
pytest -n 4
```
This runs your tests in parallel using 4 processes.

11. Generate a Test Report (JUnit XML):

You can generate a test report in JUnit XML format, useful for continuous integration systems:

```
pytest --junitxml=report.xml
```

12. **Running with a Custom Configuration File:**

If you have a custom configuration file (e.g., `pytest.ini`, `tox.ini`, or `setup.cfg`), Pytest will automatically pick up settings from it when running tests:

```
pytest
```

13. **Run with Coverage (Using pytest-cov Plugin):**

To check code coverage while running tests, use the `pytest-cov` plugin:

```
pytest --cov=path/to/module
```

14. **Passing Arguments to Test Functions:**

If you are using Pytest's `@pytest.mark.parametrize`, you can specify the arguments directly in the test function call:

```
pytest test_example.py::test_function[param]
```

15. **Run Tests with a Maximum Failures Limit:**

To specify a maximum number of failures before stopping:

```
pytest --maxfail=3
```

16. **Run Tests with Output Capture Disabled:**

If you want Pytest not to capture output from your tests (e.g., logs or print statements), use the `-s` option:

```
pytest -s
```

17. **Run Tests available in folder:**

If you want Pytest to run all test available in folder then navigate to folder and then run py.test or for more details py.test -v. To see print statements in test : py.test -v -s:

```
py.test
py.test -v
py.test -v -s
```

Example of Running Pytest from Command Line:

- Run all tests:
  ```
  pytest
  ```
- Run tests in a specific folder:
  ```
  py.test
  py.test -v
  py.test -v -s
  ```
- Run tests in a specific file:
  ```
  pytest test_calculator.py
  ```

- Run a specific test:
  `pytest test_calculator.py::test_add`

By combining these options, you can customize the way Pytest runs your tests from the command line.

---

wPytest Tags Mechanism to run tests based on functionality

Failing and Skipping tests with Annotations using Pytest

what is "mark" in pytest

In **Pytest**, `mark` refers to **custom markers** that allow you to categorize and selectively run tests based on different labels or tags. Markers are annotations you can add to your test functions to group them, indicate certain behaviors, or provide additional metadata for tests. They can be built-in or custom-defined.

Key Features of Pytest Markers:

1. **Selective Test Execution**: Run specific sets of tests (e.g., slow tests, database tests, etc.).
2. **Parametrization**: Use markers to pass different sets of inputs to a test.
3. **Skipping or Expecting Failures**: Skip certain tests or mark tests as expected to fail.

1. Built-in Pytest Markers:

Pytest includes several built-in markers, such as:

a. @pytest.mark.skip:

This marker is used to skip a specific test.

```
import pytest

@pytest.mark.skip(reason="Not implemented yet")
def test_addition():
    assert 1 + 1 == 2
```

b. @pytest.mark.skipif(condition, reason):

Skip a test if a certain condition is met (e.g., specific Python version).

```
@pytest.mark.skipif(sys.platform == 'win32', reason="Doesn't run on Windows")
```

```
def test_platform_dependent_code():
    assert some_code()
```

c. **@pytest.mark.xfail**:

Mark a test as expected to fail. If the test fails, it will not be counted as a failure.

```
@pytest.mark.xfail
def test_function_expected_to_fail():
    assert 1 + 1 == 3
```

2. **Custom Markers**:

You can create your own markers to categorize and run tests selectively. For example, you might want to mark tests as `slow`, `database`, or `api`.

a. **Defining Custom Markers**:

To define custom markers, you need to add them to the `pytest.ini` configuration file to avoid warnings. For example:

```
# pytest.ini
[pytest]
markers =
    slow: marks tests as slow (deselect with '-m "not slow"')
    database: marks tests as related to database
```

b. **Applying Custom Markers**:

You can apply custom markers to your test functions:

```
@pytest.mark.slow
def test_large_data_processing():
    pass

@pytest.mark.database
def test_database_connection():
    pass
```

3. **Running Marked Tests**:

Once you mark your tests, you can run them selectively using the `-m` option with the `pytest` command. For example, to run only the `slow` tests:

```
pytest -m slow
```

Or, to run tests that are **not** slow:

```
pytest -m "not slow"
```

4. **Parametrization with Markers**:

Markers can also be used for test parameterization to run the same test with different input values.

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

Example of Using Pytest Markers:

Defining the marker:

```
@pytest.mark.api
def test_api_endpoint():
    response = requests.get("https://example.com/api")
    assert response.status_code == 200
```

Running only the marked tests:

```
pytest -m api
```

5. **Combining Multiple Markers:**

You can combine multiple markers when running tests. For example, if you want to run both `slow` and `database` tests:

```
pytest -m "slow or database"
```

Summary:

- **Markers** allow you to categorize, skip, or expect failures for your tests.
- **Built-in markers** (like `skip`, `skipif`, `xfail`) help control test execution.
- **Custom markers** allow you to label tests and run them selectively.
- Custom markers must be defined in the `pytest.ini` file.

Markers make it easier to manage and execute specific sets of tests, making your test suite more organized and manageable.

What are fixtures and importance of their Hooks in Pytest

Different scopes of fixtures and their related annotations to setup Pre and Post conditions on the Test

In **Pytest**, **fixtures** are a powerful feature that provide a way to set up and clean up the state or environment before and after tests are executed. They help in organizing test code, reusing setup code, and ensuring the proper initialization of resources such as databases, APIs, or files.

Fixtures can be used to share setup logic among multiple tests and ensure that the required resources are available for the test to run, making them more maintainable, modular, and clean.

Key Features of Fixtures in Pytest:

1. **Reusability**: You can define a fixture once and use it across multiple test functions.
2. **Modularity**: You can break down complex setups into smaller, independent components.
3. **Automatic Invocation**: Fixtures are automatically invoked by Pytest when specified in a test.
4. **Scope**: You can define the scope of a fixture to control its lifetime (e.g., per test, module, or session).

1. Basic Example of a Pytest Fixture:

A fixture can be created using the `@pytest.fixture` decorator, and it is passed as an argument to the test function where it will be used.

```
import pytest

@pytest.fixture
def setup_data():
    data = {"name": "John", "age": 30}
    return data

def test_check_name(setup_data):
    assert setup_data["name"] == "John"

def test_check_age(setup_data):
    assert setup_data["age"] == 30
```

In this example:

- `setup_data` is a fixture that prepares some data (a dictionary in this case).
- The test functions `test_check_name` and `test_check_age` use the fixture by simply including it as an argument.
- The fixture is automatically executed before each test that uses it.

2. Fixture Scope:

By default, fixtures have a **function scope**, meaning they are created and executed for each test function. However, you can change the fixture's scope to one of the following:

- `function` (default): The fixture is executed once per test function.
- `class` : The fixture is executed once per test class.
- `module` : The fixture is executed once per test module.
- `session` : The fixture is executed once per entire test session (useful for long-lived resources like databases).

Example with Different Fixture Scopes:

```
import pytest
```

```python
@pytest.fixture(scope="module")
def setup_module_data():
    print("\nSetting up module data")
    return {"db": "connected"}

@pytest.fixture(scope="function")
def setup_function_data():
    print("\nSetting up function data")
    return {"user": "Alice"}

def test_check_db_connection(setup_module_data):
    assert setup_module_data["db"] == "connected"

def test_check_user_data(setup_function_data):
    assert setup_function_data["user"] == "Alice"
```

In this example:

- `setup_module_data` runs only once for all tests in the module due to its `module` scope.
- `setup_function_data` runs before each individual test due to its `function` scope.

3. Using `yield` for Teardown:

In cases where you need to do **teardown** or cleanup after the test is done (e.g., closing a database connection or file), you can use `yield` in the fixture. The code after the `yield` statement is executed after the test is finished.

Example with Teardown:

```python
import pytest

@pytest.fixture
def setup_teardown():
    print("\nSetting up resources")
    yield "resource"
    print("\nTearing down resources")

def test_using_resource(setup_teardown):
    assert setup_teardown == "resource"
```

In this example:

- The setup part runs before the test.
- The code after `yield` runs after the test for cleanup (teardown).

4. Parameterized Fixtures:

You can **parameterize** fixtures to run the same test multiple times with different data or setups. This is useful for testing a variety of inputs or configurations.

Example with Parameterized Fixture:

```python
import pytest

@pytest.fixture(params=[10, 20, 30])
def setup_numbers(request):
    return request.param

def test_number_is_even(setup_numbers):
    assert setup_numbers % 2 == 0
```

In this example:

- The test will run three times, once for each parameter value (`10` , `20` , and `30`).
- The `request.param` object provides the current parameter value.

5. **Fixture Dependencies:**

Fixtures can **depend on other fixtures**, allowing you to build complex setups by composing multiple fixtures together.

Example with Dependent Fixtures:

```python
import pytest

@pytest.fixture
def setup_db():
    print("\nConnecting to DB")
    return "DB Connection"

@pytest.fixture
def setup_user(setup_db):
    print(f"\nCreating user with {setup_db}")
    return {"username": "user1"}

def test_user_creation(setup_user):
    assert setup_user["username"] == "user1"
```

In this example:

- `setup_user` depends on `setup_db` .
- The database connection (`setup_db`) is created before the user is set up.

6. **Autouse Fixtures:**

You can mark a fixture as **autouse**, meaning it will automatically be invoked for all tests, even if it's not explicitly passed as an argument.

Example with Autouse Fixture:

```python
import pytest

@pytest.fixture(autouse=True)
def auto_setup():
    print("\nAuto setup before each test")

def test_first():
```

```
    assert 1 == 1

def test_second():
    assert 2 == 2
```

In this example:

- The `auto_setup` fixture runs automatically before each test.

Summary:

- **Fixtures** provide a way to set up and clean up resources before and after tests.
- They are reusable, modular, and can have different scopes (function, module, class, session).
- They support **teardown** with `yield` and can be **parameterized** for running tests with different data.
- Fixtures can **depend** on other fixtures, enabling complex setups.
- They can also be **autouse** to automatically run without being explicitly called.

Fixtures enhance code organization, making tests more manageable, reusable, and clean.

@pytest.mark.usefixtures

In **Pytest**, `@pytest.mark.usefixtures` is a decorator that allows you to explicitly specify which **fixtures** should be used in a test function, class, or module. It is an alternative to passing the fixture directly as an argument to the test function. This can be useful when you want to use a fixture for its side effects (such as setting up a database, starting a server, etc.), but you don't need to directly reference the fixture within the test function itself.

Key Use Cases for `usefixtures`:

- When the fixture is needed for **setup or teardown** purposes but is not directly used in the test function.
- When applying fixtures to **classes** or **modules**, affecting multiple test functions at once.
- To make tests **cleaner** by avoiding unused fixture arguments in function signatures.

Syntax:

```
@pytest.mark.usefixtures("fixture_name")
def test_function():
    # Test logic goes here
```

You can apply it to individual test functions, classes, or even the whole module.

Example 1: Applying `usefixtures` to a Test Function

```python
import pytest

@pytest.fixture
def setup_environment():
    print("\nSetting up environment")

@pytest.mark.usefixtures("setup_environment")
def test_sample():
    print("Running test_sample")
    assert True
```

Explanation:

- `setup_environment` is a fixture that prints a message when it is called.
- `@pytest.mark.usefixtures("setup_environment")` ensures that `setup_environment` runs before `test_sample`, even though `setup_environment` is not passed as an argument to `test_sample`.

Output:

```
Setting up environment
Running test_sample
```

Example 2: Applying `usefixtures` to a Class

You can also apply the `usefixtures` decorator to an entire class, and it will be applied to all test methods within that class.

```python
import pytest

@pytest.fixture
def login():
    print("\nLogging in as user")

@pytest.mark.usefixtures("login")
class TestUserActions:

    def test_view_profile(self):
        print("Testing profile view")

    def test_edit_profile(self):
        print("Testing profile edit")
```

Explanation:

- The `login` fixture will run before each test method (`test_view_profile` and `test_edit_profile`), even though it is not passed as an argument to any of these methods.
- The `usefixtures` decorator is applied at the class level, so all test methods in the class will automatically use the fixture.

Output:

```
Logging in as user
Testing profile view

Logging in as user
Testing profile edit
```

Example 3: Applying `usefixtures` to a Module

You can apply `usefixtures` at the module level by putting it at the top of your test module. This will apply the fixture to all tests in the module.

```python
import pytest

@pytest.fixture
def initialize_db():
    print("\nInitializing the database")

@pytest.mark.usefixtures("initialize_db")
def test_insert_data():
    print("Testing data insertion")

@pytest.mark.usefixtures("initialize_db")
def test_fetch_data():
    print("Testing data fetching")
```

Explanation:

- Both `test_insert_data` and `test_fetch_data` will run the `initialize_db` fixture, which sets up the database, even though it's not passed as an argument.

When to Use `usefixtures`:

1. **Fixture with Side Effects**: Use `usefixtures` when the fixture only performs setup or teardown tasks and is not directly required in the test logic.
2. **Cleaner Test Function**: It helps you keep test function signatures clean by avoiding unnecessary fixture arguments.
3. **Class-Level or Module-Level Setup**: When you want to ensure certain setup steps apply to multiple tests, but don't want to pass the fixture to each test function manually.

Summary:

- `@pytest.mark.usefixtures` is used to ensure that a fixture is executed before a test or set of tests without passing the fixture as an argument.
- It is useful for applying fixtures that have side effects or when you want to apply a fixture across multiple test functions, classes, or entire modules.
- It keeps test function signatures cleaner by eliminating the need to include unused fixture arguments.

How fixtures can be configured in Conftest file for better readability

In Pytest, `conftest.py` and `pytest.ini` are configuration files that help in customizing and organizing the test suite. They serve different purposes but work together to make testing more efficient and flexible.

1. `conftest.py`:

The `conftest.py` file is a special configuration file for Pytest that allows you to share fixtures, hooks, and other settings across multiple test files. It's typically used to define **fixtures** or **hooks** that can be used by all test files in a directory or subdirectories without the need for importing them explicitly.

Key Features of `conftest.py`:

- **Local to a Directory**: The configuration and fixtures in `conftest.py` apply to all tests in the same directory and any subdirectories.
- **No Imports Needed**: You don't have to import the fixtures defined in `conftest.py` into each test file; they are automatically available.
- **Hierarchical Behavior**: You can have multiple `conftest.py` files in different directories, and Pytest will use the closest one in the directory hierarchy for that particular test.

Common Use Cases:

- Defining fixtures shared across multiple test modules.
- Adding custom test hooks (e.g., modifying test execution behavior).
- Setting up logging, database connections, or other shared resources.

Example of `conftest.py`:

```python
# conftest.py
import pytest

# A shared fixture for all test files in this directory and
subdirectories
@pytest.fixture
def setup_database():
    print("\nSetting up database connection")
    db = "Connected to DB"
    yield db
    print("\nTearing down database connection")
```

Example Usage in a Test File:

```python
# test_example.py
def test_db_connection(setup_database):
    assert setup_database == "Connected to DB"
```

In this example:

- `setup_database` is a fixture defined in `conftest.py`, and it is automatically available to any test in the directory and subdirectories where `conftest.py` is located.

2. `pytest.ini`:

The `pytest.ini` file is a configuration file that defines project-wide settings for Pytest. It allows you to configure various Pytest behaviors, such as custom markers, test paths, and default command-line options.

Key Features of `pytest.ini`:

- **Global Configuration**: The settings in `pytest.ini` apply to the entire test suite.
- **Custom Markers**: You can define custom markers to label tests.
- **Command-line Options**: Set default Pytest options like verbosity, test paths, and more.
- **Test Discovery**: Configure how Pytest discovers test files (e.g., by setting specific patterns).

Example of `pytest.ini`:

```
# pytest.ini
[pytest]
markers =
    slow: marks tests as slow (use '-m slow' to run)
    database: marks tests as database-related
addopts = -v --maxfail=3
testpaths = tests
```

Key Sections:

1. **markers**:
    - Defines custom markers (e.g., `slow`, `database`) so you can tag and selectively run tests.
2. **addopts**:
    - Specifies additional default command-line options. In this example, `-v` enables verbose output, and `--maxfail=3` stops the test run after 3 failures.
3. **testpaths**:
    - Defines where Pytest should look for tests. Here, it is set to look inside the `tests/` folder.

Example Use Case for `pytest.ini`:

- You want to mark certain tests as "slow" and configure Pytest to stop after 3 test failures. By defining this in `pytest.ini`, all tests in your project will adhere to this behavior without specifying it manually on the command line.

Summary:

- **`conftest.py`**: A Python file used to share fixtures, hooks, and setup logic across multiple test files in a directory or subdirectory. It's automatically discovered by Pytest.
- **`pytest.ini`**: A configuration file to set project-wide Pytest options, including custom markers, default command-line options, and test discovery paths.

Together, `conftest.py` and `pytest.ini` help organize and streamline your test suite, making it easier to manage complex testing setups.

In what scenario one will be forced to pass fixture name though you have declared it globally?

In Pytest, even though fixtures can be defined globally (in `conftest.py` or at the module level), there are certain scenarios where you may still need to pass the fixture name explicitly as an argument in the test function. Here's when and why you'd do that:

1. When the Test Requires the Return Value of the Fixture:

If your test needs to use the return value of a fixture directly, you must pass the fixture name as an argument to the test function. Merely marking it globally (e.g., with `usefixtures`) won't allow you to access the fixture's return value.

Example:

```
import pytest

@pytest.fixture
def db_connection():
    return "Database Connection Established"

def test_query(db_connection):
    assert db_connection == "Database Connection Established"
```

In this case, the test function `test_query` uses the return value of the `db_connection` fixture, so the fixture must be passed as an argument.

2. When the Fixture Is Parameterized:

If a fixture is parameterized, it needs to be passed as an argument to allow the test function to run with different values for each parameter.

Example:

```
import pytest

@pytest.fixture(params=["chrome", "firefox", "safari"])
def browser(request):
    return request.param

def test_browser_compatibility(browser):
    print(f"Testing on {browser}")
```

Here, the `browser` fixture is parameterized to test with different browsers. You need to pass the fixture as an argument to access the parameterized value for each test run.

3. When Using Fixtures in Test Logic:

If your test logic directly depends on the value or output of the fixture, or if you need to assert on the result or output of the fixture, passing it explicitly is necessary.

Example:

```
@pytest.fixture
def user():
    return {"username": "test_user", "email": "user@example.com"}

def test_user_login(user):
    assert user["username"] == "test_user"
```

Here, the test needs the `user` data to validate the login logic, so the fixture is passed as an argument.

4. When Using Multiple Fixtures:

If a test requires multiple fixtures, you must pass them as arguments explicitly in the function definition.

Example:

```
@pytest.fixture
def setup_database():
    return "DB setup complete"

@pytest.fixture
def setup_cache():
    return "Cache setup complete"

def test_application(setup_database, setup_cache):
    assert setup_database == "DB setup complete"
    assert setup_cache == "Cache setup complete"
```

Each fixture that the test function needs must be explicitly included as a parameter.

5. When You Need to Override a Fixture:

In some cases, you may need to override a fixture in certain tests. You must pass the fixture as an argument explicitly in the test function to override or modify its behavior locally.

Example:

```
@pytest.fixture
def user():
    return {"username": "default_user", "email":
"default@example.com"}

def test_override_user(user):
    user["username"] = "custom_user"
    assert user["username"] == "custom_user"
```

Here, `test_override_user` modifies the default fixture value, so the fixture is passed explicitly to the function.

6. When You Want to Control the Fixture Scope:

Fixtures with larger scopes (e.g., module, session) are often used for setting up global resources. However, if a specific test needs to alter or use a different resource, you'll need to pass the fixture to the test function to control how and when the resource is used.

Example:

```
@pytest.fixture(scope="module")
def shared_resource():
    return "Resource for the whole module"

def test_use_shared_resource(shared_resource):
    assert shared_resource == "Resource for the whole module"
```

The fixture is passed because you want to control its usage within the test function, even though it has a global scope.

7. When You Need Access to Fixture Internals (like `request` or `caplog`):

Some fixtures, like `request`, provide special capabilities such as accessing parameters or metadata about the test itself. These fixtures need to be passed as arguments to the test function for accessing their internals.

Example:

```
@pytest.fixture
def my_fixture(request):
    print(f"Running {request.node.name}")
    return "fixture_value"

def test_with_request(my_fixture):
    assert my_fixture == "fixture_value"
```

In this case, the `request` object is only accessible when the fixture is passed explicitly to the test function.

Summary:

You need to pass fixture names explicitly as arguments in the following scenarios:

- When your test function needs to **use the fixture's return value**.
- When your fixture is **parameterized** to run with multiple inputs.
- When you're using **multiple fixtures** in one test.
- When you want to **override** or modify the fixture locally.
- When you need to **control the fixture's scope** or have a fixture with a broader scope.
- When you want to access **special fixture features**, like `request` or `caplog`.

In these cases, the fixture must be passed as an argument so that its result or value can be used directly within the test function logic.
