**What is pytest?**

• Pytest is identified as one of the most popular testing frameworks for the Python programming language.

• It is recognized for its simplicity, scalability, and powerful features.

• Pytest is widely used for writing both simple unit tests and more complex functional and integration tests.

**Pytest framework:**

• Simple Syntax: Tests are written as regular Python functions, and there is no requirement to inherit any specific class.

• Automatic Discovery: Pytest automatically discovers test files and functions. This works by searching for test files that start with the test\_ keyword, followed by an underscore, or end with \_test. Similarly, test functions should begin with the keyword test\_ followed by an underscore.

• Parameterization: This feature enables running a single test multiple times with different sets of data.

• Fixtures: Fixtures are used for the setup and tear-down phases of the scripts.

• Detail Assertions: Pytest provides numerous assertions that enable proper testing of your application.

Regarding the question of why Pytest is needed, the source states that features specific to testing, such as automatic discovery of tests, parameterization, and fixtures, are not as robust in some other frameworks. It suggests the need for something more powerful in terms of automation testing.

**Writing First Test Using Pytest:**

File name: test\_even\_odd.py

def is\_even\_or\_odd\_new(n):

    if n % 2 == 0:

        return "Even"

    else:

        return "Odd"

def test\_even\_number\_new():

    result = is\_even\_or\_odd\_new(4)

    print(f"Test for 4: {result}")

def test\_odd\_number\_new():

    result = is\_even\_or\_odd\_new(3)

    print(f"Test for 3: {result}")

def test\_large\_even\_number\_new():

    result = is\_even\_or\_odd\_new(10000)

    assert result == "Even", "10000 should be even"

To execute the code, type pytest in the terminal. Which shows output like this:

A computer screen shot of a program code

AI-generated content may be incorrect.

**How to Execute Test Cases:**

To run a specific file in a folder, in the terminal type: pytest ‘relative path of that file’.  
For example, pytest pytest\test\_even\_odd.py

**A computer screen shot of a program

AI-generated content may be incorrect.**

To run a specific function on a specific file in a folder, in the terminal type: pytest ‘relative path of that file::specif function name’.

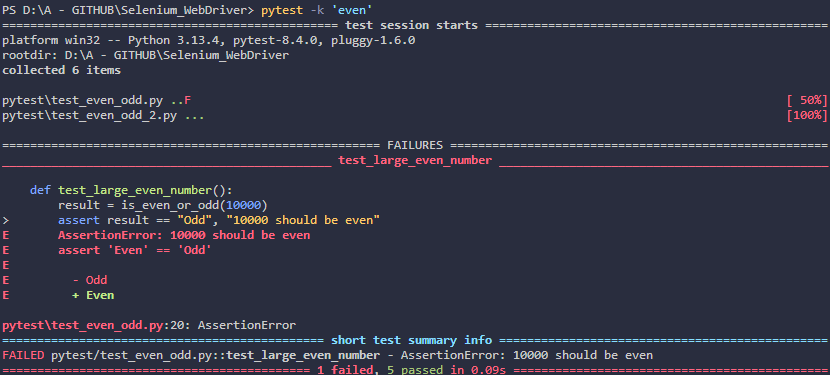
For example, pytest pytest\test\_even\_odd.py::test\_large\_even\_numberclear

**A computer screen shot of a program

AI-generated content may be incorrect.**

To search for a specific keyword in a folder, in the terminal type: pytest -k ‘The search keyword’

For example, pytest -k ‘even’



To get more information from a specific file, in the terminal type: pytest -v ‘relative path of the file’. ‘v’ stands for verbose.

For example, pytest -v pytest\test\_even\_odd.py

**A computer screen shot of a program

AI-generated content may be incorrect.**

To run the last failed files, in the terminal type: pytest –lf

A computer screen shot of a program

AI-generated content may be incorrect.

To stop the code from running after the first failure, in the terminal type: pytest -x ‘relative path of the file’.

For example, pytest -x pytest\test\_even\_odd.py

A computer screen shot of a program

AI-generated content may be incorrect.

**Markers in Pytest:**

Markers allow labeling test cases with specific names, which enables grouping, categorizing, and running a particular set of tests. Markers are essentially just labels or tags.

**Why Group Tests Using Markers?**

The primary reason for grouping tests with markers is that when executing automation scripts, there's no need to run every test case every time. For instance, when a new build is available, only sanity or smoke tests might require execution. In other cases, regression tests might need to be run. Markers allow the execution of only a specific set of test cases, such as smoke or regression, so everything doesn't have to run.

**Types of Markers**

The sources mention two types of markers: custom markers and built-in markers.

Custom Markers:

Custom markers first import pytest into the code and add @pytest.mark.‘custom marker keyword’ before the desired function.

For example, @pytest.mark.smoke

A computer screen shot of a code

AI-generated content may be incorrect.

Output:

To run the code, in the terminal type: pytest -x ‘custom marker name.

For example, pytest -m smoke

A screenshot of a computer screen

AI-generated content may be incorrect.

To combine markers in the code, in the terminal type:

pytest -x ‘custom marker name or custom marker name’

pytest -x ‘custom marker name and not custom marker name’

For example, pytest -m “smoke or regression”, pytest -m “smoke and not regression”.

**Skip and SkipIf Markers:**

Pytest provides **Skip** and **SkipIf** markers, used to control test execution. These markers help manage scenarios where certain tests should not run under specific conditions.

Skip Marker:

**Purpose:** The skip marker is used to unconditionally skip a test. No condition is required; the test is skipped for any reason.

**Use Case:** This is helpful when it's known beforehand that a particular test is irrelevant or incomplete and should not run.

**Syntax:** It's defined using the @pytest.mark.skip decorator above a test function.

**Example:** @pytest.mark.skip, or @pytest.mark.skip(reason="functionality not developed")

A computer screen shot of numbers

AI-generated content may be incorrect.

Output:

A computer screen shot of a program

AI-generated content may be incorrect.\

SkipIf Marker:

The skipif marker is used to conditionally skip a test. Test execution is skipped if a specified condition is met.

**Use Case:** This is useful in scenarios where skipping depends on a runtime check. For example, a test might need to:

* Execute Selenium scripts on a specific browser version and check its availability.
* Check the operating system (e.g., execute only on Windows or Linux).
* Ensure a particular test only runs if a prerequisite test has executed successfully.

**Syntax:** It's defined using the @pytest.mark.skipif decorator.

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

* Condition: The first argument required for skipif is the condition. If this condition evaluates to True, the test will be skipped. For instance, using a variable feature\_available = False, the condition not feature\_available (which is True in this case) would cause the test to skip.
* Reason: An optional reason argument can be provided, similar to the skip marker, to explain the condition for skipping. Example: @pytest.mark.skipif(not feature\_available, reason="feature not available").

A computer screen shot of a computer code

AI-generated content may be incorrect.

Output:

A computer screen shot of a computer program

AI-generated content may be incorrect.

**Parametrize Marker:**

Pytest offers a powerful feature called **parameterization**. The @pytest.mark.parametrize marker is specifically designed to implement this feature. Its primary purpose is to run the same test logic multiple times with different sets of data. This allows testing a function or piece of code thoroughly using various inputs without writing separate test functions for each case.

**Syntax:**

The Parametrize marker is applied as a decorator using @pytest.mark.parametrize directly above a test function. This decorator takes at least two main arguments:

1. argnames (Parameter Names): A string containing comma-separated names for the variables that will receive data in the test function. For example, "a, b, expected".
2. argvalues (Parameter Values): A list of tuples. Each tuple in the list represents one set of data for a single execution of the test. The values within each tuple must correspond to the variable names listed in argnames. For example, [(1, 2, 3), (4, 5, 9)].

For example:

A computer screen shot of a program code

AI-generated content may be incorrect.

Output:

A computer screen with white text

AI-generated content may be incorrect.

**Parametrize Marker:**

The **xfail** marker indicates that a test is expected to fail. This marker is particularly useful for tests known to be failing due to specific reasons, such as a bug or a feature not yet being implemented.

**Syntax**

The **xfail** marker is applied as a decorator using @pytest.mark.xfail(condition) above a test function, similar to other Pytest markers.

For example:

A computer screen shot of a code

AI-generated content may be incorrect.

Output:

A computer screen shot of a program

AI-generated content may be incorrect.

**Fixtures:**

In Pytest, **fixtures** are functions that serve as a robust and flexible mechanism for setting up and tearing down the environment or resources required by tests. They provide a defined, consistent, and reusable baseline for test execution.

**Why Use Fixtures?**

Using fixtures offers several advantages:

* The same setup code can be shared across multiple tests.
* Test dependencies are isolated.
* Tests remain focused on the actual test logic.

**Syntax**

A fixture is defined using the @pytest.fixture decorator above a Python function.

A computer screen shot of a program code

AI-generated content may be incorrect.

Output:

A computer screen shot of a program

AI-generated content may be incorrect.

**Parameterized fixtures:**

Parameterized fixtures in Pytest offer a way to run the same test with multiple sets of data. This feature combines the power of fixtures and parameters to make tests more flexible, modular, and data-driven.

**Why Use Parameterized Fixtures?**

In automation testing, scenarios often arise requiring functionality to be tested with different inputs. For example, testing a login feature with various user credentials or a registration form with different data sets. Parameterized fixtures address this by allowing reuse of test logic across these different data sets.

**Syntax and Usage**

A parameterized fixture is defined using the @pytest.fixture decorator, similar to regular fixtures. To parameterize it, the params argument is included, providing a sequence (like a list or tuple) of the data sets for use.

A screen shot of a computer program

AI-generated content may be incorrect.

Output:

A computer screen shot of a program

AI-generated content may be incorrect.

**Xdist Plugin:**

The **Pytest Xdist plugin** is a popular plugin for Pytest. Its primary function is to enable **parallel test execution**. This means tests can be run simultaneously across multiple CPU cores on a single machine or even across different machines.

**Why Use the Xdist Plugin?**

The main reason to use the Xdist plugin is to significantly **reduce test execution time**. This is particularly beneficial for large test suites containing many test cases. Without this plugin, tests execute one after another, increasing total time. Using Xdist allows tests to run in parallel, thereby reducing overall execution time.

**How to Use the Xdist Plugin?**

1. **Installation:** First, the plugin needs to be installed. This is done using pip:

pip install pytest-xdist

1. **Running Tests in Parallel:** After installation, tests can be run in parallel using the pytest command with the -n option, followed by the desired number of parallel executions (often referred to as "workers").
2. For example, to run tests using 4 parallel workers, the following command is used:

pytest -n 4 filepath

1. When this command executes, Pytest will show the number of active workers. A particular file can also be specified for parallel execution, for instance:
2. If a number of workers larger than the number of collected test items is specified, Pytest will simply use the number of collected items.

A computer screen shot of a program

AI-generated content may be incorrect.

**HTML Reports:**

Generating HTML reports in Pytest is a common practice, particularly useful for sharing test execution results with stakeholders like management. These reports help understanding the status of automation tests, including the number of passed, failed, or skipped test cases, providing insight into application quality.

The benefits of HTML reports include:

* Improved visual understanding of test results.
* Easy sharing with team members.
* Detailed information, including the status of passed, failed, and skipped tests.

To generate an HTML report in Pytest, a plugin is typically used. The specific plugin for this purpose is **pytest-html**.

**How to Use the pytest-html Plugin**

1. **Installation:** First, install the plugin using pip:

pip install pytest-html (This command installs the pytest-html plugin into the project)

1. **Generation:** After installation, generate the HTML report by running the Pytest command with the --html argument, specifying the desired output filename:

pytest --html=report.html

An example for generating a report for a specific file named html.py is:

pytest html.py --html=report.html (When this command runs, Pytest will execute the tests and generate an HTML file (e.g., report.html) in the project directory.)

1. **Viewing the Report:** The generated HTML file can then be opened to view the report. It can be opened in a web browser or a built-in preview tool. The report includes information such as the Python version, platform, and packages used, along with a summary of the test results. It also allows filtering results (e.g., by failed, error, or passed tests) and shows detailed information for each test.

A screen shot of a computer

AI-generated content may be incorrect.