**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’.

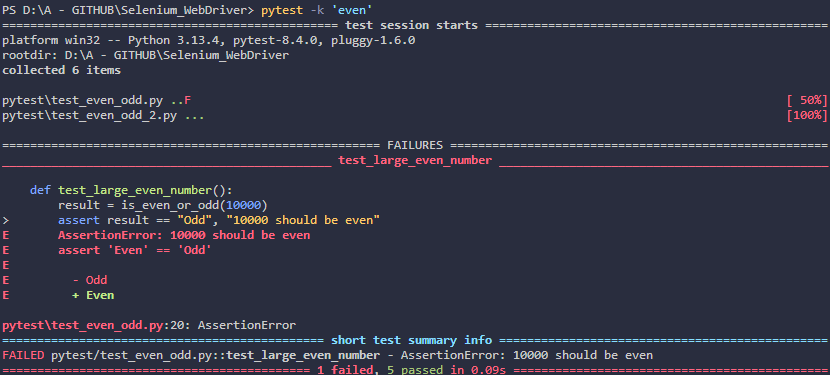
For example, pytest pytest\test\_even\_odd.py::test\_large\_even\_numberclear

**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.