**I. Running Tests with Pytest -K Options**

refer : https://pytest-with-eric.com/introduction/pytest-k-options/#Running-Tests-with-Pytest-K-Options

***1. Specifying a Substring:***

pytest -v -k checkout

***2. Using Python Expressions:***

This command will run all the tests containing the keyword cart but exclude those containing the keyword core.

pytest -v -k 'cart and not core'

It will run all the tests containing the keyword cart but exclude the test containing the keywords core, logout, and pin.

pytest -v -k "cart and not core and not logout and not pin"

***3. Other Options to Filter Tests***

You can easily exclude unnecessary tests by applying markers like @pytest.mark.skip or @pytest.mark.xfail

@pytest.mark.skip(reason="No need to test core function. SKIPPED!")

def test\_cart\_management\_core():

assert True

@pytest.mark.xfail(reason="Intentionally declaring the test as fail. Shows Xpass if it passes")

def test\_cart\_item\_list():

assert True

***4. Run Test by Node***

pytest -v tests/test\_cart\_access.py::test\_cart\_core\_authentication

5. pytest.ini

[pytest]

adopts =

-k core

[pytest]

addopts =

-k "core and not (test\_api or test\_ui)"

**II. Pytest Markers (refer: https://pytest-with-eric.com/pytest-best-practices/pytest-markers/)**

Pytest markers are a powerful feature that allows you to add metadata or labels to your test functions, making it easier to organize and customize your test suite.

Markers help you categorize and select specific tests to run, especially when dealing with large test suites.

***1. Markers***

import pytest

@pytest.mark.smoke

def test\_homepage\_loads():

# Test to check if the homepage loads quickly

assert ...

@pytest.mark.regression

def test\_login\_successful():

# Test to check if the login process works as expected

assert ...

@pytest.mark.regression

def test\_user\_profile\_update():

# Test to check if user profile updates are saved correctly

assert ...

pytest -m smoke # Run only smoke tests

pytest -m regression # Run only regression tests

***2. Common Built-In Markers***

***- Pytest Skip Test / Skip If***

import pytest

import sys

# A test that will always be skipped.

@pytest.mark.skip(reason="This test is temporarily disabled.")

def test\_example\_skip():

assert 2 + 2 == 4

# A test that will be skipped if it's run on a Python version earlier than 3.8.

@pytest.mark.skipif(sys.version\_info < (3, 8), reason="Requires Python 3.8 or later.")

def test\_example\_skipif():

assert 3 \* 3 == 9

***- Pytest Expected Failure (Xfail)***

# A test that's expected to fail.

@pytest.mark.xfail(reason="Expected to fail until we fix the bug.")

def test\_example\_xfail():

assert 2 \* 3 == 7

- Pytest Parameterization

@pytest.mark.parametrize("arg1, arg2, ...", [(val1, val2, ...), ...])

# Test function demonstrating the parametrize feature.

# It will run 3 times with different inputs.

@pytest.mark.parametrize("test\_input,expected", [(1, 3), (3, 5), (5, 7)])

def test\_addition(test\_input, expected):

assert test\_input + 2 == expected

***- Pytest Fixtures***

import pytest

# A fixture returning a sample database entry.

@pytest.fixture

def database\_data():

return {"username": "Alice", "password": "password123"}

# Test function using the database\_data fixture.

def test\_database\_entry(database\_data):

assert database\_data["username"] == "Alice"

assert database\_data["password"] == "password123"

***- Pytest Timeout***

@pytest.mark.timeout(seconds)

Specifies a maximum execution time for a test. If the test runs longer than the specified timeout, it’s automatically marked as a failure.

This is useful for preventing tests from running indefinitely.

import pytest

import time

# A Slow Running Test that's expected to timeout.

@pytest.mark.timeout(10)

def test\_timeout():

time.sleep(15)

assert 2 \* 3 == 6

***- Pytest Run Order***

Need a plugin to work : pip install pytest-order

@pytest.mark.run(order) or @pytest.mark.run(order=order)

Allows you to control the order in which tests are executed. The order argument specifies the relative execution order of tests.

import pytest

@pytest.mark.order(2)

def test\_foo():

assert True

@pytest.mark.order(1)

def test\_bar():

assert True

- Combining Multiple Markers

import pytest

@pytest.mark.marker1

@pytest.mark.marker2

def test\_combined\_markers():

assert 1 + 1 == 2

pytest -m marker1 -m marker2 tests/test\_combined.py -v -s

***- Grouping Tests***

import pytest

# Define custom markers

pytestmark = [

pytest.mark.login,

pytest.mark.signup

]

# First test for login functionality

@pytest.mark.login

def test\_login\_valid\_user():

username = "valid\_user"

password = "valid\_pass"

assert username == "valid\_user" and password == "valid\_pass"

# Second test for login functionality

@pytest.mark.login

def test\_login\_invalid\_user():

username = "invalid\_user"

password = "valid\_pass"

assert username != "valid\_user" and password == "valid\_pass"

# First test for signup functionality

@pytest.mark.signup

def test\_signup\_new\_user():

new\_username = "new\_user"

new\_password = "new\_pass"

assert new\_username == "new\_user" and new\_password == "new\_pass"

# Second test for signup functionality

@pytest.mark.signup

def test\_signup\_existing\_user():

existing\_username = "existing\_user"

assert existing\_username == "existing\_user"

pytest -v -m login tests/test\_grouping.py -v -s

pytest -v -m signup tests/test\_grouping.py -v -s

3. Define Markers in Pytest.ini File

[pytest]

markers =

development: marks tests as development (deselect with '-m "not development"')

production: marks tests as production (deselect with '-m "not production"')

fast: marks tests as fast (run with '-m fast')

slow: marks tests as slow (run with '-m slow')

custom: custom marker example (run with '-m custom')

asyncio: marks tests requiring asyncio (run with pytest-asyncio plugin)

xfail: marks tests that are expected to fail (handled by pytest itself)

xpass: marks tests that unexpectedly pass after being marked xfail (handled by pytest itself)

parameters: marks parameterized tests (handled by pytest itself)

benchmark: marks tests used for benchmarking (handled by pytest-benchmark plugin)

celery: marks tests related to Celery tasks (custom marker, specifics depend on test implementation)

login: dummy login marker for grouping test

signup: dummy signup marker for grouping test

marker1: combined markers

marker2: combined markers

timeout: test with timeout

III. What Is `pytest.ini` And How To Save Time Using Pytest Config

refer: https://pytest-with-eric.com/pytest-best-practices/pytest-ini/

Config files help you define how you want each program or unit test to behave on execution.

Without it, you would need to specify how the test should work every time you run it, often using several CLI commands.

IV.