44. pytest Framework - 2

pytest Skipping

In pytest, skipping is a way to skip the execution of certain tests based on conditions. This is useful when tests are not relevant in certain environments or when some features are not yet implemented.

Types of Skipping options

1. **@pytest.mark.skip** ⇒ Skip unconditionally.
2. **@pytest.mark.skipif** ⇒ Skip based on conditions.
3. **pytest.skip()** ⇒ Skip inside test execution.
4. **xfail** ⇒ Mark test as expected to fail but still run it.

test_Skipping.py

```
import pytest
import sys
class TestSkipping:
                                    Unconditional skip
  @pytest.mark.skip(reason="Feature not implemented yet")
  def test_unfinished(self):
    assert True
                                Conditional skip based on system platform
  @pytest.mark.skipif(sys.platform == "win32", reason="Doesn't run on Windows")
  def test_non_windows(self):
    assert True
                            Conditional skip inside the test based on runtime condition
  def test_dynamic_skip(self, request):
    if not request.config.getoption("--run-extra-tests", default=False):
      pytest.skip("Skipping because --run-extra-tests is not set")
    assert True
                                Using a fixture to skip based on external condition
  @pytest.fixture
  def skip_if_no_network(self):
                                Imagine this checks if the network is available
    network_available = False  # Simulate no network
    if not network_available:
      pytest.skip("Skipping because no network is available")
  def test_network_feature(self, skip_if_no_network):
    assert True
                                    xfail - Expected to fail
  @pytest.mark.xfail(reason="Known bug in this feature")
  def test_failing_feature(self):
    assert False  # This test will fail, but marked as xfail
```

Commands to Execute from Terminal

1. pytest -s -v **-rs** Day2/test_Skipping.py ⇒ display reasons for skipped tests in the test report

2. pytest -s -v **-rx** Day2/test_Skipping.py ⇒ display reasons for expected failures (xfail) in test report.
3. pytest -s -v **-rxs** Day2/test_Skipping.py ⇒ display reasons for both in test report

pytest Ordering ⇒ **pip install pytest-ordering**

1. By default, pytest runs test functions in the order they are discovered in the test file. Test discovery depends on Python's sorting mechanism, which might not be predictable, especially across different operating systems. This means the tests may not always run in the exact order they are written in the file.
2. To explicitly control the order of test execution, the **pytest-ordering plugin** can be used. It allows you to set the order for test functions, classes, and modules.

test_Ordering.py

```
import pytest
class TestClass:
 @pytest.mark.run(order=3)
 @pytest.mark.third
 def test_methodC(self):
    print("Running method C.. ")
 @pytest.mark.run(order=4)
 @pytest.mark.fourth
 def test_methodD(self):
    print("Running method D.. ")
 @pytest.mark.run(order=5)
 @pytest.mark.fifth
 def test_methodE(self):
    print("Running method E.. ")
 @pytest.mark.run(order=1)
 @pytest.mark.first
 def test_methodA(self):
    print("Running method A.. ")
 @pytest.mark.run(order=2)
 @pytest.mark.second
 def test_methodB(self):
    print("Running method B.. ")
```

Commands to Execute from Terminal

- pytest -s -v Day2/test_Ordering.py

Note

➔ Tests are executed in CDEAB manner without makers.

➔ If we specify markers they are executed in proper order ABCDE.

➔ If we are getting warnings when we execute from the Terminal

- specify markers in the **pytest.ini** file

or

- pytest -p no:warnings -s -v Day2/test_Ordering.py

pytest.ini file ⇒ contains customized markers .if we don't specify orders are not executed in specific order.

```
[pytest]
markers=
    first
    second
    third
    fourth
    fifth
```

pytest Dependancy ⇒ **pip install pytest-dependency**

If the test method fails, whatever the method depends on it should skip instead of executing and failing.

test_Dependency.py

```
import pytest
class TestClass:
  @pytest.mark.dependency()
  def test_openApp(self):
     assert True
  @pytest.mark.dependency(depends=['TestClass::test_openApp'])
  def test_login(self):
     assert True
  @pytest.mark.dependency(depends=['TestClass::test_login'])
  def test_search(self):
     assert False
  @pytest.mark.dependency(depends=['TestClass::test_login','TestClass::test_search'])
  def test_advsearch(self):
     assert True
  @pytest.mark.dependency(depends=['TestClass::test_login'])
  def test_logout(self):
     assert True
```

After execution we can see some additional warnings.To avoid these kind of warnings we can also add dependency marker in the pytest.ini.

44. pytest Framework - 3

Suppose I have multiple test methods when i run all the test methods are executed but i don't want to execute all the methods just group them like sanity regression.After grouping i can run only sanity,only regression ,both sanity and regression,i can run only sanity not belong to regression ,i can run only regression not belong to sanity ,multiple combinations we can run .This is possible by using grouping concept. To achieve grouping we need to add markers in the ini file.

Grouping Tests - @pytest.mark.sanity, @pytest.mark.regression

test_Grouping.py

```
import pytest
class TestClass:
  @pytest.mark.sanity
  def test_LoginByEmail(self):
     print("This is login by email.....")
     assert 1 == 1
  @pytest.mark.sanity
  def test_LoginByFacebook(self):
     print("This is login by facebook.....")
     assert 1 == 1
  @pytest.mark.sanity
  @pytest.mark.regression
  def test_LoginByTwitter(self):
     print("This is login by twitter.....")
     assert 1 == 1
  @pytest.mark.sanity
  @pytest.mark.regression
  def test_signupByEmail(self):
     print("This is signup by email test")
     assert True == True
  @pytest.mark.regression
  def test_signupByFacebook(self):
     print("This is signup by facebook test")
     assert True == True
  @pytest.mark.regression
  def test_signupbytwitter(self):
     print("This is signup by twitter test")
     assert True == True
```

Command to execute :

- pytest -v -s -m "sanity" day37-pytest3\test_Grouping.py
- pytest -v -s -m "sanity and regression" day37-pytest3\test_Grouping.py
- pytest -v -s -m "not sanity" day37-pytest3\test_Grouping.py – to execute only regression not belongs to sanity

- pytest -s -v -m "not sanity" day37-pytest3\test_Grouping.py

pyTest Parallel Test Execution ⇒ **pip install pytest-xdist**

- Pytest parallel testing can significantly speed up the test execution process by running tests in parallel across multiple CPU cores. This is useful for large test suites and can reduce the time taken for test execution.
- The pytest-xdist plugin enables parallel test execution in pytest. It allows tests to be distributed across multiple processors, threads, or machines.

Parallel execution - install package **pytest-xdist**

Command to execute : pytest -n=3 -v -s day27\test_Parallel.py

test_Parallel.py

```
import pytest
from selenium import webdriver
class TestTitle:
  def test_title_chrome(self):
    opt = webdriver.ChromeOptions()
    opt.add_experimental_option("detach", True)
    driver = webdriver.Chrome(options=opt)
    driver.get("https://www.google.com/")
    act_title = driver.title
    exp_title = "Google"
    if act_title == exp_title:
      print("Test passed")
    else:
      print("Test Failed")
    assert act_title == exp_title  # validation
    driver.quit()
  def test_title_edge(self):
    opt = webdriver.EdgeOptions()
    opt.add_experimental_option("detach", True)
    driver = webdriver.Edge(options=opt)
    driver.get("https://www.google.com/")
    act_title = driver.title
    exp_title = "Google"
    if act_title == exp_title:
      print("Test passed")
    else:
      print("Test Failed")
    assert act_title == exp_title  # validation
    driver.quit()
  def test_title_firefox(self):
    opt = webdriver.FirefoxOptions()
    driver = webdriver.Firefox(options=opt)
    driver.get("https://www.google.com/")
```

```
act_title = driver.title
exp_title = "Google"
if act_title == exp_title:
    print("Test passed")
else:
    print("Test Failed")
assert act_title == exp_title  # validation
driver.quit()
```