Pytest

INDEX:

1. Why we need a framework

- Structure our test
- Added helpul features

2. About Pytest

pytest is a popular testing framework for Python. It is used to write simple and scalable test cases, making it easy to test and debug Python code. Here are some key features and benefits of pytest:

- Pytest framework is based on python language
- Easy to write, execute and generate test reports
- Different types and levels of testing
- Used by developers and QA team
- Auto detect tests
- Grouping/Marking Tests
- Uses python’s assert keyword
- Features – fixtures,parameterize,etc

3. Creation and Activation of Virtual Environment

venv is a module that provides support for creating lightweight, isolated Python environments. Each environment has its own installation directories and does not share libraries with other venv environments, nor with the global Python installation. This is particularly useful for managing dependencies for different projects separately, avoiding conflicts between packages required by different projects.

3.1 Key Features and Benefits of venv

Isolation: Each virtual environment is isolated from others, allowing you to work on multiple projects with different dependencies simultaneously without conflicts.

Dependency Management: You can maintain a separate set of dependencies for each project. This ensures that your project runs with the exact versions of libraries it was developed and tested with.

Reproducibility: By using a requirements.txt file, you can ensure that other developers or deployment environments can recreate the same environment.

Compatibility: It allows using different versions of Python for different projects. This is especially useful when you need to support multiple versions of Python.

To create Virtual Environment

```
PS C:\Users\vlab\Desktop\Pytest> python -m venv myenv
```

To Activate Virtual Environment

```
PS C:\Users\vlab\Desktop\Pytest> myenv\Scripts\activate
```

Installation of Pytest

```
(myenv) PS C:\Users\vlab\Desktop\Pytest> pip install pytest
```

To view the installed packages:

```
(myenv) PS C:\Users\vlab\Desktop\Pytest> pip list
```

To save the installed packages in a file

Use **pip freeze > requirements.txt**

To see the file

Use **cat requirements.txt**

Write First Test

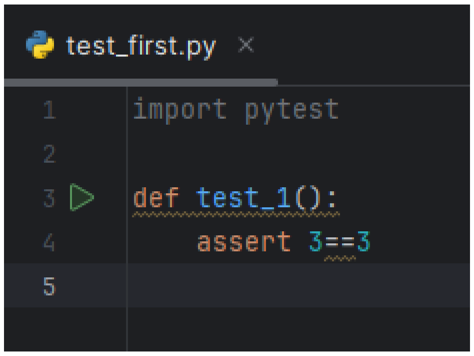
4. Naming conventions for test file

test_<name>.py or <name>**_test**.py

So while running pytest is going to search for the test files in current directory and sub-directories!

Naming conventions for test Function

```
def test_name():
      --------
      --Code--
      --------
```



```python
import pytest

def test_1():
    assert 3==3
```

Assert is part of python library
assert keyword lets you test if a condition in your code returns True

Run the test

```
(myenv) PS C:\Users\vlab\Desktop\Pytest> pytest
===========================================    test    session    starts
============================================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0
rootdir: C:\Users\vlab\Desktop\Pytest
collected 1 item

test_first.py                                                                 .
[100%]
```

```
============================================ 1 passed in 0.23s
=============================================
```

Do not add multiple assert statements in a single test function

```
    import pytest

    def test_1():
        assert 3==3
        assert 3-3 == 0
```

We can also give the message comment

```
    def test_2():
        assert 5-5 == 5 , "failed intentionally"
```

The Output:

```
(myenv) PS C:\Users\vlab\Desktop\Pytest> pytest
========================================= test session starts
=========================================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0
rootdir: C:\Users\vlab\Desktop\Pytest
collected 2 items

test_first.py                                                               .F
[100%]

============================================== FAILURES
===============================================
_______________________________________________ test_2
________________________________________________

    def test_2():
>       assert 5-5 == 5 , "failed intentionally"
E       AssertionError: failed intentionally
```

```
E       assert (5 - 5) == 5

test_first.py:8: AssertionError
======================================= short test summary info
=======================================
FAILED test_first.py::test_2 - AssertionError: failed intentionally
===================================== 1 failed, 1 passed in 0.12s
=====================================
```

We can use the pytest verbose mode which clearly gives the about about which test has passed and which has failed

```
(myenv) PS C:\Users\vlab\Desktop\Pytest> pytest -v
========================================= test session starts
=========================================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0 --
C:\Users\vlab\Desktop\Pytest\myenv\Script
s\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\vlab\Desktop\Pytest
collected 2 items

test_first.py::test_1                                          PASSED
[ 50%]
test_first.py::test_2                                          FAILED
[100%]
```

5. About Pytest__cache__

We can also see the pytest cache contents which stores the first failed tests, last failed tests and etc...

Commands : pytest --lf
pytest --ff
pytest -cache-show

6. Advantages of python package

Creating a package gives you __init__.py
We can name same file names in different folders

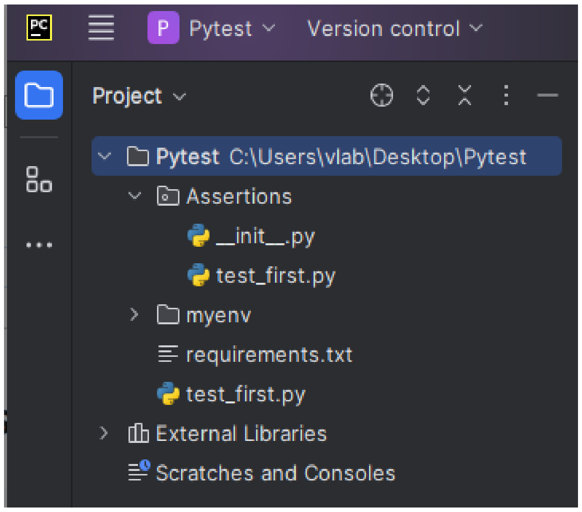
7. Test class

Naming convention : must starts with Test<classname>:

```
    class TestMycode:
        def test_type(self):
            assert type(1) == int
        def test_strs(self):
            assert str.upper("sri") == "SRI"
```

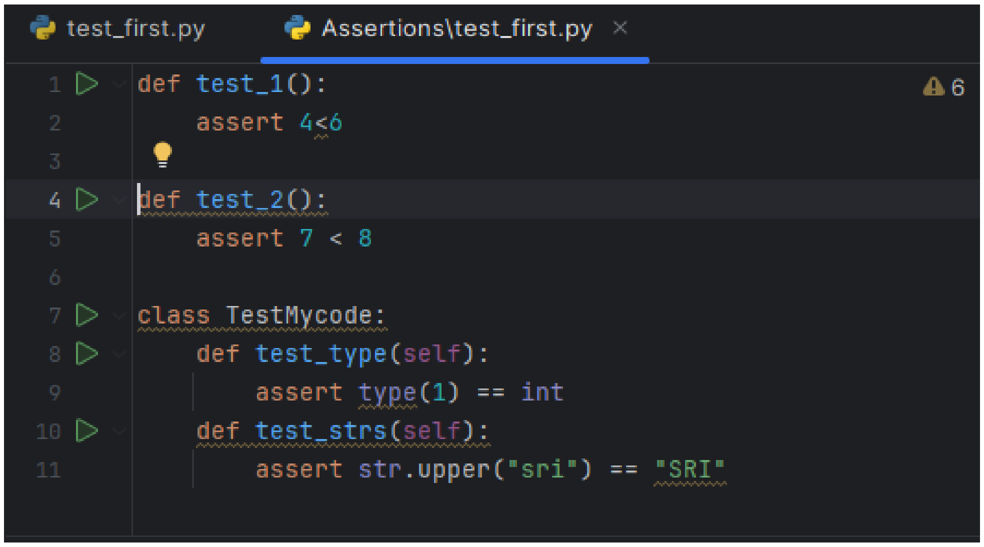
8. RUN methods

Here is the projectfile structure



```
Pytest C:\Users\vlab\Desktop\Pytest
  Assertions
    __init__.py
    test_first.py
  myenv
  requirements.txt
  test_first.py
External Libraries
Scratches and Consoles
```

In Assertions
test_first.py ---> **4 tests 2 seperate,2 in a class**

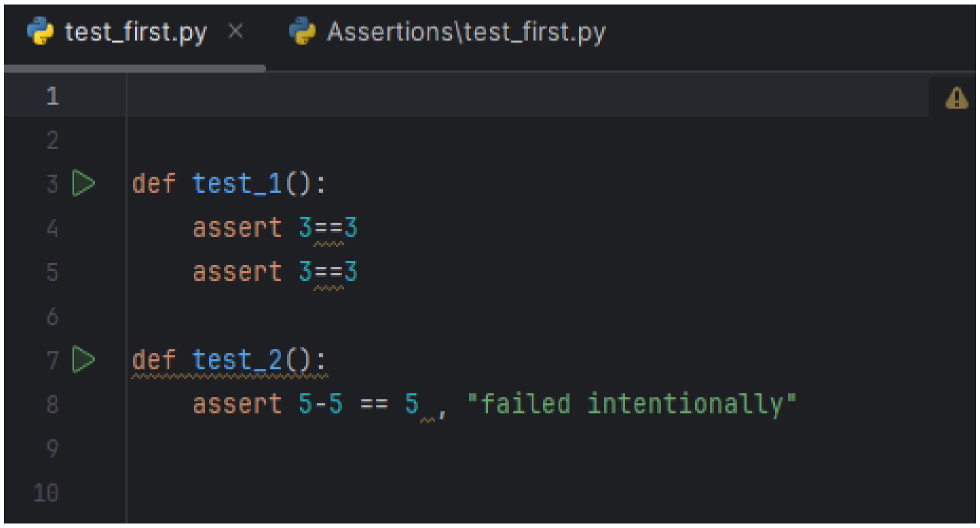


```python
def test_1():
    assert 4<6

def test_2():
    assert 7 < 8

class TestMycode:
    def test_type(self):
        assert type(1) == int
    def test_strs(self):
        assert str.upper("sri") == "SRI"
```

```
In pytest
Test_first.py --->  **2 tests**
```



```python


def test_1():
    assert 3==3
    assert 3==3

def test_2():
    assert 5-5 == 5 , "failed intentionally"


```

Total 6 tests

- **Running pytest will check for all the tests in directory and subdirectories**

```
(myenv) C:\Users\vlab\Desktop\Pytest>pytest
```

```
======================================================== test session
starts ========================================================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0 --
C:\Users\vlab\Desktop\Pytest\myenv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\vlab\Desktop\Pytest
collected 6 items
```

- Running pytest Aseertions/test_first.py will run the tests inside test_first.py file

```
(myenv) C:\Users\vlab\Desktop\Pytest>pytest Assertions/test_first.py
======================================================== test session
starts ========================================================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0
rootdir: C:\Users\vlab\Desktop\Pytest
collected 4 items
```

- Running pytest test_first.py will run the tests inside test_first.py file

```
(myenv) C:\Users\vlab\Desktop\Pytest>pytest test_first.py
======================================================== test session
starts ========================================================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0
rootdir: C:\Users\vlab\Desktop\Pytest
collected 2 items
```

- We can also run the class seperately by specifying classname like using :: after test file name.

```
(myenv)C:\Users\vlab\Desktop\Pytest>pytest
Assertions/test_first.py::TestMycode
======================================================== test session
starts ========================================================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0
rootdir: C:\Users\vlab\Desktop\Pytest
collected 2 items
```

- We can also run the single test in a class seperately by specifying testcase name after the classname using :: after class name.

```
(myenv)C:\Users\vlab\Desktop\Pytest>pytest
Assertions/test_first.py::TestMycode::test_strs
==========================================================        test
session                                                            starts
==========================================================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0
rootdir: C:\Users\vlab\Desktop\Pytest
collected 1 item
```

```
(myenv) C:\Users\vlab\Desktop\Pytest>pytest test_first.py::test_1
==========================================================        test
session                                                            starts
==========================================================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0
rootdir: C:\Users\vlab\Desktop\Pytest
collected 1 item
```

About __init__.py

If __init__.py is present it will allow us to create same file names in multiple directories and sub-directories

To understand remove the init file and try to run the tests using pytest

It will gives you a error

```
import file mismatch:
imported module 'test_first' has this __file__ attribute:
  C:\Users\vlab\Desktop\Pytest\Assertions\test_first.py
which is not the same as the test file we want to collect:
  C:\Users\vlab\Desktop\Pytest\test_first.py
```

HINT: remove __pycache__ / .pyc files and/or use a unique basename for your test file modules

8.1 Run test by test name using –k

-k option enables us to pass in an expression to run tests.
Allows and, or and not etc in the expression
We can also use -k option for the module names.

8.2 --tb=no

Traceback =no means it will not show the traceback full information means reason for failure it will only show which test got passed and which failed.
Pytest -v -k "module or testname " --tb =no

9. Pytest.raises()

pytest.raises() is a function provided by the pytest framework to assert that a block of code raises a specified exception. It is commonly used in unit tests to verify that code behaves as expected when it encounters error conditions.

9.1 Basic Usage

Here is a basic example demonstrating the use of pytest.raises():

```
def test_zero_division():
    with pytest.raises(ZeroDivisionError):
        1/0
```

In this example, the test will pass if dividing by zero raises a ZeroDivisionError exception.

Run the above test by commenting with pytest.raises()
It will gives you a zero division error and execution is aborted

So, simple pytest.raises is used to handle exceptions.

9.2 To print the exception:

```
import pytest

def test_zero_division():
    with pytest.raises(Exception) as excinfo:
        assert (1,2,3) == (1,2,4)
    print(str(excinfo))
```

10. Markers

Markers are basically kind of grouping uour test or making your test, and you can seperately run those group or the mark test seperately using -m option.

- Test can have multiple markers.
- A marker can be on multiple tests.

File-name : test_markers.py

```
import pytest



@pytest.mark.str
@pytest.mark.sanity
def test_1():
    assert str.capitalize("sri") == "Sri"



@pytest.mark.smoke
def test_2():
    assert str.center("sri", 5, "*") == "*sri*"

@pytest.mark.digit
@pytest.mark.smoke
def test_3():
    assert str.isdigit("2") == True




@pytest.mark.sanity
def test_4():
    assert str.isalpha("sri") == True
```

```python
@pytest.mark.regression
def test_5():
    assert str.lower("SRI") == "sri"

@pytest.mark.str
@pytest.mark.sanity
def test_6():
    assert str.upper("sri") == "SRI"
```

```
(myenv) PS C:\Users\vlab\Desktop\Pytest> pytest -m regression
======================================      test      session      starts
=======================================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0
rootdir: C:\Users\vlab\Desktop\Pytest
configfile: pytest.ini
collected 13 items / 12 deselected / 1 selected

(myenv) PS C:\Users\vlab\Desktop\Pytest> pytest -m smoke
======================================      test      session      starts
=======================================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0
rootdir: C:\Users\vlab\Desktop\Pytest
configfile: pytest.ini
collected 13 items / 11 deselected / 2 selected
```

Like this it will run all the tests which are mentioned with specified marker

Markers support AND , OR , NOT operators

```
(myenv) PS C:\Users\vlab\Desktop\Pytest> pytest -m "sanity and str"
======================================      test      session      starts
=======================================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0
rootdir: C:\Users\vlab\Desktop\Pytest
configfile: pytest.ini
collected 13 items / 11 deselected / 2 selected
```

```
(myenv) PS C:\Users\vlab\Desktop\Pytest> pytest -m "sanity or str"
======================================      test      session      starts
=======================================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0
rootdir: C:\Users\vlab\Desktop\Pytest
configfile: pytest.ini
collected 13 items / 10 deselected / 3 selected
```

You will get warning if you dont define markers in pytest.ini file
File-name pytest.ini

```
[pytest]
markers =
    sanity
    smoke
    regression
    str
    digit
```

Defining markers at Module level

```
pytestmark = [pytest.mark.smoke]
```

So when we run using **pytest -m “smoke”** it will run all the tests inside the file.

11. xfail

it means you expect a test to fail for some reason e.g known bug

@pytest.mark.xfail(reason=”known issue”)

```
import pytest

@pytest.mark.xfail
def test_str2():
    str2 = "srijyothsna"
    assert str2[15] == "a"
```

```python
@pytest.mark.xfail
def test_str3():
    str3 = "sri"
    num = 1234
    assert str3 + num == "sri1234"

def test_strjoin():
    str1 = "balla,sri and jyothsna"
    l1 = ["balla,sri", "and","jyothsna"]
    assert ' '.join(l1) == str1
```

If you run this code without pytest.mark.xfail it will through u an error and reports as 2 failed tests.
By using xfail those tests will be ignored and didnt through an error called failed tests.

Output:

```
(myenv) PS C:\Users\vlab\Desktop\Pytest> pytest -v test_xfail
.py
=================== test session starts ====================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0 -
- C:\Users\vlab\Desktop\Pytest\myenv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\vlab\Desktop\Pytest
configfile: pytest.ini
collected 3 items

test_xfail.py::test_str2 XFAIL                      [ 33%]
test_xfail.py::test_str3 XFAIL                      [ 66%]
test_xfail.py::test_strjoin PASSED                  [100%]

=============== 1 passed, 2 xfailed in 0.28s ===============

Xpass : test passes despite being expected to fail
```

Xpass will come if we run the test which is going to be pass marked as xfail then xpass will come in the output.

```
@pytest.mark.xfail
def test_str2():
    str2 = "srijyothsna"
    #with pytest.raises(Exception) as excinfo:
    assert str2[10] == "a"
    #print(excinfo)
```

Output:

```
(myenv) PS C:\Users\vlab\Desktop\Pytest> pytest -v test_xfail
.py
=================== test session starts ====================
platform win32 -- Python 3.11.5, pytest-8.2.2, pluggy-1.5.0 -
- C:\Users\vlab\Desktop\Pytest\myenv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\vlab\Desktop\Pytest
configfile: pytest.ini
collected 3 items

test_xfail.py::test_str2 XPASS                         [ 33%]
test_xfail.py::test_str3 XFAIL                         [ 66%]
test_xfail.py::test_strjoin PASSED                     [100%]

========= 1 passed, 1 xfailed, 1 xpassed in 0.12s ==========
```

11.1 Xfail with conditions

Import sys
@pytest.mark.xfail(sys.platform == “win32”, reason = “works only in win32”)

```
import pytest
import sys

@pytest.mark.xfail(raises=IndexError,reason="known issue")
def test_str2():
    str2 = "srijyothsna"
    #with pytest.raises(Exception) as excinfo:
    assert str2[15] == "a"
    #print(excinfo)

@pytest.mark.xfail(sys.platform == "win32",reason = "works only in windows")
```

```
def test_str3():
    str3 = "sri"
    num = 1234
    #with pytest.raises(Exception) as excinfo:
    assert str3 + num == "sri1234"
    #print(excinfo)

def test_strjoin():
    str1 = "balla,sri and jyothsna"
    l1 = ["balla,sri", "and","jyothsna"]
    assert ' '.join(l1) == str1
```

Run the code it will work fine if you change the raises for 1st test case from index error to type error then this test gonna fail.
Because it will through an error called index error but in raises if you mention type error then it will fail
same for win32 if you changed to linux then it will fail because it will run only on linux specified but ur running on windows.

Pytest Command line options

To see the options pytest –help/pytest -h

Known options:

-v verbose mode
-k run tests based on given string
-m to run markers
--tb =no no traceback
-s to output the text in print statements in terminal
-x stop after first failure
--maxfail stop after specified maxfail
Ex –maxfail = 2 means it will stop executing after 2 failed tests
-q quiet execution

```
(myenv) PS C:\Users\vlab\Desktop\Pytest> pytest -q test_mar
kers.py
......                                                     [100%]
6 passed in 0.02s
```

--collect-only or --co dont want to run the tests, only collect.

--lf last failed tests
--ff it will run the failed tests first then other tests
--disable-warnings to disable warnings for ex when we not specified markers in pytest.ini file

12. Test outputs of pytest

```
PASSED(.)  : The test ran successfully
FAILED(F)  : The test did not run successfully
SKIPPED(s) : The test was skipped
XFAIL(x)   : The test was not supposed to pass, ran, and failed
XPASS(X)   : The test was not supposed to pass, ran and passed
ERROR(E)   : An exception happened outside of the test function
```

13. Pytest parametrization

In pytest, parameterization allows you to run the same test function with different sets of input parameters. This is particularly useful when you want to test a function or a piece of code with multiple inputs or conditions. Parameterization helps in writing concise and maintainable tests by reducing code duplication.

Syntax: @pytest.mark.parametrize("arg1, arg2, ..., expected", [(val1_1, val1_2, ..., expected1), (val2_1, val2_2, ..., expected2), ...])

When you want to find list of numbers that are less than 50 then we can use parametrization

File name: test_parametrization.py

```
@pytest.mark.parametrize("input",[10,60,30,40])
def test_param1(input):
    assert input<50
```

Output:

```
============================== test session starts ==============================
collecting ... collected 4 items
```

```
test_parametrization.py::test_param1[10]
test_parametrization.py::test_param1[60]
test_parametrization.py::test_param1[30]
test_parametrization.py::test_param1[40]

=========================    1    failed,    3    passed    in    0.15s
=========================
PASSED                                                       [  25%]FAILED
[ 50%]
test_parametrization.py:2 (test_param1[60])
60 != 50

Expected :50
Actual   :60
<Click to see difference>

input = 60

    @pytest.mark.parametrize("input",[10,60,30,40])
    def test_param1(input):
>       assert input<50
E       assert 60 < 50

test_parametrization.py:5: AssertionError
PASSED                                                       [  75%]PASSED
[100%]
```

13.1 Passing multiple arguements

```
@pytest.mark.parametrize("input,output",[(2,5),(3,27),(4,256)])
def test_param2(input,output):
    assert (input**input) == output
```

```
Output:
test_parametrization.py::test_param2[2-4]                          PASSED
[ 33%]
```

```
test_parametrization.py::test_param2[3-27]                      PASSED
[ 66%]
test_parametrization.py::test_param2[4-256]                     PASSED
[100%]
=================================================== 3 passed in 0.04s
===================================================
```

We can also define the arguements outside parametrize marker

```
data = [
    ("sri",2),
    ("deepu",5),
]

@pytest.mark.parametrize("name,length",data)
def test_param3(name,length):
    assert len(name) == length
```

14. Fixtures

To setup and teardown

In pytest, fixtures are functions that provide a baseline or setup for your tests. They can be used to initialize objects, prepare the environment, or set up dependencies needed by multiple tests.

Fixtures are functions that are run by pytest before (and sometimes after) the actual test functions.
 e.g initialize webdriver.

Fixtures can be put in individual file called conftest.py for making fixtures available in multiple test files.

14.1 Fixture in the same file

Two ways of calling a fixture

1.from test function (passing fix as arguement)

```
import pytest

@pytest.fixture()
```

```
def setup_list():
    print("setting list in fixture\n")
    friends = ["deepu","lucky","sarika","sri"]
    return friends

def test_getitem(setup_list):
    assert setup_list[0] == "deepu"
    assert setup_list[0]+setup_list[1] == "deepulucky"
```

2.from mark decorator (@pytest.mark.usefixtures(fix name))
We can use the decorator function @pytest.mark.usefixtures()

```

@pytest.mark.usefixtures("setup_list")
def test_reverselist():
    assert setup_list[::-1] == ["sri","sarika","lucky","deepu"]

```

But here we cannot access the return items in fixture just it will calls the fixture and we cannot access the return items.
It will gives u an error called.
TypeError: 'function' object is not subscriptable

14.2 Setup and teardown

```
import pytest

my_wishlist = ["watch","earphones","scooty"]
sis_wishlist = ["dress","earrings","slippers"]
full_list = ["watch","earphones","scooty","phone","dress","earrings","slippers"]
@pytest.fixture()
def setup_list1():
    print("\nsetting the list in fixture....")
    cpy1 = my_wishlist.copy()
    cpy1.append("phone")
    yield cpy1
    print("\ntearing down the copy after yield in fixture\n")
    cpy1.clear()

def test_get_full_list(setup_list1):
    setup_list1.extend(sis_wishlist)
    assert setup_list1 == full_list
```

14.3 Difference between return and yield

Execution of function stops after return statement
Execution of function continues after the yield statement

Output:

```
collected 1 item
test_fixtures2.py::test_get_full_list
setting the list in the fixture....
PASSED
tearing down the copy after yield in fixture
```

Like this way you can initialize means setup the driver and teardown means closing the driver after usage in web testing can be done using fixtures.

14.4 Multiple fixtures in same file

```python
import pytest

my_wishlist = ["watch","earphones","scooty"]
sis_wishlist = ["dress","earrings","slippers"]
full_list = ["watch","earphones","scooty","phone","dress","earrings","slippers"]
@pytest.fixture()
def setup_list1():
    print("\nsetting the list in fixture....")
    cpy1 = my_wishlist.copy()
    cpy1.append("phone")
    yield cpy1
    print("\ntearing down the copy after yield in fixture\n")
    cpy1.clear()
@pytest.fixture()
def setup_list2():
    print("\nsetting the list in fixture....")
    cpy1 = sis_wishlist.copy()
    cpy1.append("laptop")
    yield cpy1
    print("\ntearing down the copy after yield in fixture\n")
    cpy1.clear()

def test_get_full_list(setup_list1,setup_list2):
    setup_list1.extend(sis_wishlist)
    assert setup_list1 == full_list
    assert setup_list2[-1] == "laptop"
```

Output:

```
collected 1 item
test_fixtures2.py::test_get_full_list
setting the list in fixture....
setting the list in fixture....
PASSED
tearing down the copy after yield in fixture
tearing down the copy after yield in fixture
==================================================== 1 passed in 0.04s
====================
```

14.5 Creating file and removing using fixture

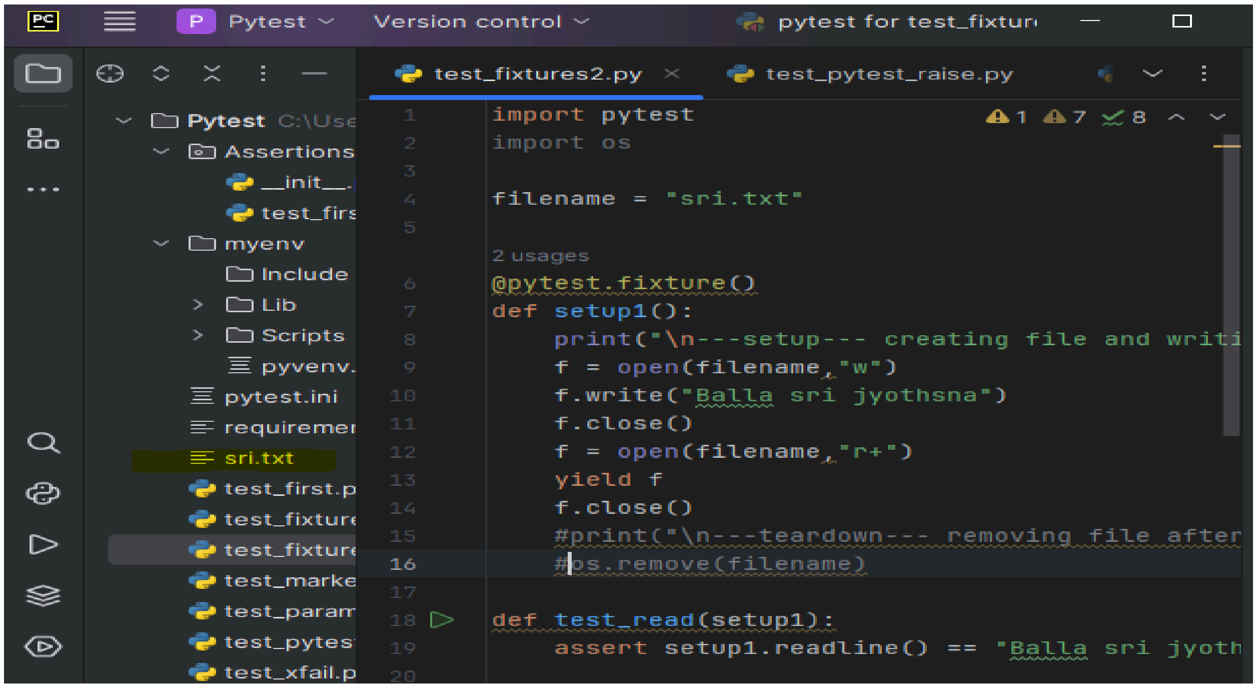
```
import pytest
import os

filename = "sri.txt"

@pytest.fixture()
def setup1():
    print("\n---setup--- creating file and writing text\n")
    f = open(filename,"w")
    f.write("Balla sri jyothsna")
    f.close()
    f = open(filename,"r+")
    yield f
    f.close()
    #print("\n---teardown--- removing file after usage\n")
    #os.remove(filename)

def test_read(setup1):
    assert setup1.readline() == "Balla sri jyothsna"
```

run the following code a file named sri.txt will be created and it will not be deleted after execution.



```
import pytest
import os

filename = "sri.txt"

@pytest.fixture()
def setup1():
    print("\n---setup--- creating file and writi
    f = open(filename,"w")
    f.write("Balla sri jyothsna")
    f.close()
    f = open(filename,"r+")
    yield f
    f.close()
    #print("\n---teardown--- removing file after
    #os.remove(filename)

def test_read(setup1):
    assert setup1.readline() == "Balla sri jyoth
```

Now un comment the code after yield statement run the code then the file sri.txt will be deleted after reding its content.

14.6 fixtures in different file named conftest.py

Conftest.py : share fixtures across multiple tests. Can have single conftest.py in centralized directory for all test to access the fixture. Also can have other conftest.py files in subdirectories.

Fixtures in conftest.py

```
import pytest
import os


def pytest_configure():
    pytest.my_wishlist = ["watch","earphones","scooty"]
    pytest.sis_wishlist = ["dress","earrings","slippers"]
                                                          pytest.full_list          =
["watch","earphones","scooty","phone","dress","earrings","slippers"]
    pytest.filename = "sri.txt"


@pytest.fixture()
def setup1():
    print("\n---setup--- creating file and writing text\n")
    f = open(pytest.filename,"w")
    f.write("Balla sri jyothsna")
    f.close()
    f = open(pytest.filename,"r+")
    yield f
    f.close()
    print("\n---teardown--- removing file after usage\n")
    os.remove(pytest.filename)
@pytest.fixture()
def setup_list1():
    print("\nsetting the list in fixture....")
    cpy1 = pytest.my_wishlist.copy()
    cpy1.append("phone")
    yield cpy1
    print("\ntearing down the copy after yield in fixture\n")
    cpy1.clear()
@pytest.fixture()
def setup_list2():
    print("\nsetting the list in fixture....")
    cpy1 = pytest.sis_wishlist.copy()
    cpy1.append("laptop")
    yield cpy1
    print("\ntearing down the copy after yield in fixture\n")
    cpy1.clear()
```

Here def pytest_configure(): is used to make the object accessable across all the test files so the objects can be accessed in any file using pytest.objectname

Test file using fixtures in conftest.py

```
import pytest

def test_read(setup1):
    assert setup1.readline() == "Balla sri jyothsna"

def test_get_full_list(setup_list1,setup_list2):
    setup_list1.extend(pytest.sis_wishlist)
    assert setup_list1 == pytest.full_list
    assert setup_list2[-1] == "laptop"
```

You can see where fixture has been used using --setup=show opion
Output:

```
C:\Users\vlab\Desktop\Pytest>pytest -vs test_fixtures2.py --setup-show
```

```
================================================= test session starts =================================================
collected 2 items
test_fixtures2.py::test_read
---setup--- creating file and writing text
        SETUP    F setup1
     test_fixtures2.py::test_read (fixtures used: setup1)PASSED
---teardown--- removing file after usage
        TEARDOWN F setup1
test_fixtures2.py::test_get_full_list
setting the list in fixture....
        SETUP    F setup_list1
setting the list in fixture....
        SETUP    F setup_list2
     test_fixtures2.py::test_get_full_list (fixtures used: setup_list1, setup_list2)PASSED
tearing down the copy after yield in fixture
        TEARDOWN F setup_list2
tearing down the copy after yield in fixture
        TEARDOWN F setup_list1

================================================== 2 passed in 0.05s ==================================================
```

Here TEARDOWN F F specifies it is function level
M means module level
Observe the output by changing the scope of one fixture to module level and note the changes in teardown.

```
@pytest.fixture(scope="module")
def setup1():
    print("\n---setup--- creating file and writing text\n")
    f = open(pytest.filename,"w")
    f.write("Balla sri jyothsna")
    f.close()
    f = open(pytest.filename,"r+")
    yield f
    f.close()
    print("\n---teardown--- removing file after usage\n")
    os.remove(pytest.filename)
```

Here teardown of setup1 will be last after all the function level firxtures teardown happend. Because scope is module level.

Understand the output by running the two codes using fixture level and module level.

15. Request

By using request we can know

Which function is calling the fixture
Which module is calling the fixture
Scope of a fixture
Objects in the test files can also be used in the fixture using request

Filename - Conftest.py

```
@pytest.fixture()
def setup1(request):

    print("\n---setup--- creating file and writing text\n")
    f = open(pytest.filename,"w")
    f.write("Balla sri jyothsna")
    f.close()
    f = open(pytest.filename,"r+")
    print("\n --- fixture scope-----"+str(request.scope))
    print("\n --- calling function---"+str(request.function.__name__))
    yield f
    f.close()
    print("\n---teardown--- removing file after usage\n")
    os.remove(pytest.filename)
```

Test file - test_fixtures2.py

```
import pytest

def test_read(setup1):
    assert setup1.readline() == "Balla sri jyothsna"
```

output:

```
collected 2 items
test_fixtures2.py::test_read
---setup--- creating file and writing text
```

```
--- fixture scope-----function
--- calling function---test_read
PASSED
---teardown--- removing file after usage
test_fixtures2.py::test_get_full_list
setting the list in fixture....
setting the list in fixture....
PASSED
tearing down the copy after yield in fixture
tearing down the copy after yield in fixture
```

We can also use the objects in the test file in fixture(conftest) file using request

Example:

Consider weekdays object is present in test file you have to use it in fixture that is present in conftest file.
Then we will access it through request

File name – test_fixtures2.py

```python
import pytest

weekdays = ["mon","tue","wed","thur","fri","sat","sun"]

def test_check_request(use_by_request):
    assert "noday" in use_by_request
    print(use_by_request)
```

Conftest.py

```python
@pytest.fixture()
def use_by_request(request):
    days = getattr(request.module,"weekdays")
    days.append('noday')
    yield days
```

16. Fixture levels

16.1 function level

A function-level fixture is created and destroyed once per test function. This is useful for scenarios where you need a fresh setup for each test.

Conftest.py

```python
import pytest

@pytest.fixture
def function_fixture():
    print("\nSetup for function-level fixture")
    yield "Function-level fixture data"
    print("\nTeardown for function-level fixture")
```

Test_fixture_level.py

```python
def test_one(function_fixture):
    assert function_fixture == "Function-level fixture data"

def test_two(function_fixture):
    assert function_fixture == "Function-level fixture data"
```

Output for fixture level

```
collected 2 items
test_fixture_level.py::test_one
Setup for function-level fixture
        SETUP    F function_fixture
                    test_fixture_level.py::test_one   (fixtures   used:
function_fixture)PASSED
Teardown for function-level fixture
        TEARDOWN F function_fixture
test_fixture_level.py::test_two
Setup for function-level fixture
        SETUP    F function_fixture
                    test_fixture_level.py::test_two   (fixtures   used:
function_fixture)PASSED
Teardown for function-level fixture
        TEARDOWN F function_fixture
```

```
==================================================  2 passed in 0.04s
==================================================
```

16.2 module level

A module-level fixture is created and destroyed once per module. All test functions in the module share the same fixture instance. This is useful for expensive setup operations that you want to perform only once per module.

Conftest.py

```python
import pytest

@pytest.fixture(scope="module")
def module_fixture():
    print("\nSetup for module-level fixture")
    yield "Module-level fixture data"
    print("\nTeardown for module-level fixture")
```

Test_fixture_level.py

```python
def test_one(module_fixture):
    assert module_fixture == "Module-level fixture data"

def test_two(module_fixture):
    assert module_fixture == "Module-level fixture data"
```

Output for module level:

```
test_fixture_level.py::test_one
Setup for module-level fixture
    SETUP    M module_fixture
                     test_fixture_level.py::test_one (fixtures used: module_fixture)PASSED
test_fixture_level.py::test_two
                     test_fixture_level.py::test_two (fixtures used: module_fixture)PASSED
Teardown for module-level fixture
    TEARDOWN M module_fixture
```

17. Factories as Fixtures

The “factory as fixture” pattern can help in situations where the result of a fixture is needed multiple times in a single test. Instead of returning data directly, the fixture instead returns a function which generates the data. This function can then be called multiple times in the test.

File name – test_factory_fix.py

```python
class User:
    def __init__(self,username,email):
        self.username = username
        self.email = email

def test_fact_fix(user_factory_fix):
    user1 = user_factory_fix("sri","sripilla94@gmail.com")
    assert user1.username == "sri"
```

Conftest.py

```python
import pytest
import os

from test_factory_fix import User

@pytest.fixture()
def user_factory_fix():
    def create_user(username,email):
        return User(username=username,email= email)
    return create_user
```

By using factory fixture, we can create multiple objects for user class.

18. Parametrization from fixtures

```python
@pytest.fixture(params=[(2,5),(3,27),(4,256)],ids = ["(2,5)","(3,27)","(4,256)"])
def fixture01(request):
    return request.param

def test_param1(fixture01):
    assert fixture01[0]**fixture01[0] == fixture01[1]
```

```
Output:
collecting ... collected 3 items
test_parametrization.py::test_param1[(2,5)]
test_parametrization.py::test_param1[(3,27)]
test_parametrization.py::test_param1[(4,256)]
=========================    1    failed,    2    passed    in    0.14s
=========================
FAILED                             [ 33%]
test_parametrization.py:16 (test_param1[(2,5)])
4 != 5
Expected :5
Actual   :4
<Click to see difference>

fixture01 = (2, 5)
    def test_param1(fixture01):
>       assert fixture01[0]**fixture01[0] == fixture01[1]
E       assert (2 ** 2) == 5

test_parametrization.py:18: AssertionError
PASSED                            [ 66%]PASSED                    [100%]
Process finished with exit code 1
```

19. Passing arguments in pytest command line

Conftest.py

```
import pytest

def pytest_addoption(parser):
    parser.addoption("--cmdopt",default="sri")

@pytest.fixture()
def cmd_fixture(pytestconfig):
    opt = pytestconfig.getoption("cmdopt")
    if opt == "unknown":
        f = open("unknown","r")
    else:
        f = open("sri","r")
```

```
  yield f
```

- **pytest_addoption**: This is a pytest hook function that allows you to add custom command-line options.
- **parser**: An argument passed to the pytest_addoption function, which is used to add command-line options.

20. parser

- **Role**: The parser object in pytest_addoption is used to add custom command-line options to pytest.
- **Methods**:
    - addoption(name, ...):
        - **name**: The name of the command-line option (e.g., --cmdopt).
        - **action**: The type of action to be taken when the option is encountered (e.g., store to store a value).
        - **default**: The default value to be used if the option is not specified by the user.
        - **help**: A description of what the option does, which will be shown in the help message.

Example:

python

Copy code

```
def pytest_addoption(parser):
        parser.addoption("--cmdopt", action="store", default="sri",
help="Custom command-line option”)
```

- **parser.addoption("--cmdopt", default="sri")**: This line adds a new command-line option --cmdopt with a default value of "sri". Users can specify this option when running pytest to modify the behavior of the tests.
- **pytestconfig**: A built-in pytest fixture that provides access to configuration values, including command-line options.

- **opt = pytestconfig.getoption("cmdopt"):** This line retrieves the value of the custom command-line option --cmdopt that was added earlier. If the option is not specified by the user, it will use the default value "sri".

Test_cmdline.py

```
def test_cmdline(cmd_fixture):
    print("content in the file---"+cmd_fixture.readline())
```

Run test without arguement

Output:

```
C:\Users\vlab\Desktop\Pytest>pytest -sk "cmdline"
================================================= test session starts
=================================================
platform win32 -- Python 3.12.2, pytest-8.2.0, pluggy-1.5.0
rootdir: C:\Users\vlab\Desktop\Pytest
configfile: pytest.ini
plugins: html-4.1.1, metadata-3.1.1
collected 33 items / 32 deselected / 1 selected

test_cmdline.py content in the file---balla sri jyothsnaaa
.

========================================== 1 passed, 32 deselected in
0.06s ============================================
```

Run test with arguement

```
C:\Users\vlab\Desktop\Pytest>pytest -sk "cmdline" --cmdopt=unknown
================================================= test session starts
=================================================
platform win32 -- Python 3.12.2, pytest-8.2.0, pluggy-1.5.0
rootdir: C:\Users\vlab\Desktop\Pytest
configfile: pytest.ini
plugins: html-4.1.1, metadata-3.1.1
collected 33 items / 32 deselected / 1 selected
```

test_cmdline.py content in the file----------------unknown file ------
.

```
========================================== 1 passed, 32 deselected in
0.05s ===========================================
```

When we run out from the directory then it will gives u no such file error then use os module and join the path.

21. Configuring pytest.ini file

- We can specifty the start and end of file name rather than test_ or _test which pytest will automatically run using

Pytest.ini

```
python_files = sri_*.py
```

- We can specify the directory name to run al the tests within the directory

Pytest.ini

```
testpaths = directory_name
```

22. Behavioral driven development framework

BDD is a framework or technique of software development

In the context of pytest, BDD (Behavior-Driven Development) is typically implemented using a combination of pytest and plugins/extensions that support BDD-style testing. Here's a breakdown of how BDD concepts can be integrated into pytest:

22.1 BDD Concepts in pytest:

1. **Feature Files and Scenarios**:
   - BDD encourages writing scenarios in a human-readable format using Given-When-Then steps. These scenarios are typically written in feature files using a language like Gherkin.
   -
2. **pytest-bdd Plugin**:
   - **pytest-bdd** is a popular plugin for pytest that allows you to write BDD-style tests using Gherkin syntax and execute them using pytest.
   - It integrates Gherkin syntax (Given-When-Then) with pytest's testing framework, enabling developers to write tests that are easily readable and understandable by non-technical stakeholders.

Sample code on bdd

File name: test_transactions.feature

```
Feature: Bank Transactions
    Tests performed on bank transactions like withdrawal, deposit

  Scenario: Withdrawal of money
    Given the account balance is 4000
    When the account holder withdraws 1000
    Then the account balance is 3000
```

File name : test_bdd.py

```
import pytest

from pytest_bdd import scenario,then,when,given

def pytest_configure():
    pytest.amount = 0

@scenario("test_addition.feature","Withdrawal of money")
def test_withdrawal():
    pass

@given("the account balance is 4000")
def current_balance():
    pytest.amount = 4000

@when("the account holder withdraw 1000")
def withdraw_amount():
    pytest.amount = pytest.amount - 1000

@then("the account balance is 3000")
def overall_balance():
    assert pytest.amount == 3000
```

22.2 Multiple scenarios in a single feature file

```
Feature: Bank Transactions
    Tests performed on bank transactions like withdrawal, deposit

  Scenario: Withdrawal of money
    Given the account balance is 4000
    When the account holder withdraw 1000
    Then the account balance is 3000

   Scenario: removing numbers from a list
    Given the user have list of 2 numbers
    When the user removes one number from a list
    Then the length of list is 1
```

Test file : test_bdd.py

```
import pytest

from pytest_bdd import scenario,then,when,given

def pytest_configure():
    pytest.amount = 0

@scenario("test_addition.feature","Withdrawal of money")
def test_withdrawal():
    pass

@given("the account balance is 4000")
```

```python
def current_balance():
    pytest.amount = 4000

@when("the account holder withdraw 1000")
def withdraw_amount():
    pytest.amount = pytest.amount - 1000

@then("the account balance is 3000")
def overall_balance():
    assert pytest.amount == 3000

@scenario("test_addition.feature","removing numbers from a list")
def test_add():
    pass

@given("the user have list of 2 numbers",target_fixture="l1")
def list():
    l1 = [50,100]
    return l1



@when("the user removes one number from a list")
def added_numbers(l1):
    l1.pop()
    print(l1)

@then("the length of list is 1")
def list_length(l1):
    print(len(l1))
```

Target_fixture: available for other given,when,then step definitions in the same test scenarios.

Using scenarios decorator instead of scenario
Consider the above feature file with 2 scenarios

Test file :

```python
import pytest
from pytest_bdd import scenarios,scenario,then,when,given

scenarios("test_addition.feature")
```

```
def pytest_configure():
    pytest.amount = 0
@given("the account balance is 4000")
def current_balance():
    pytest.amount = 4000
@when("the account holder withdraw 1000")
def withdraw_amount():
    pytest.amount = pytest.amount - 1000
@then("the account balance is 3000")
def overall_balance():
    assert pytest.amount == 3000


@given("the user have list of 2 numbers",target_fixture="l1")
def list():
    l1 = [50,100]
    return l1
@when("the user removes one number from a list")
def added_numbers(l1):
    l1.pop()
    print(l1)
@then("the length of list is 1")
def list_length(l1):
    print(len(l1))
```

Observe the outpt the pytest generates reports and take the test names as the test_scenario name we have given
Ex: test_withdrawal_of_money

22.3 Background in pytest-bdd

Background: All the steps from the background will be executed before all the scenarios own given steps

Put some common setup functions petaining to all the tests in the feature file.

There is only sten “Given” should be used in “background” section , steps “when” and “then” are prohibited, because their purpose are related to actions and consuming outcomes.

```
Feature: some practise
    Tests performed on bdd

  Background: Setting the data for test
    Given the list is not empty

  Scenario: removing numbers from a list
    Given the user have list of 2 numbers
    When the user removes one number from a list
    Then the length of list is 1
```

Here background is used for testing the list is not empty

```
import pytest

from pytest_bdd import scenarios,scenario,then,when,given

scenarios("test_addition.feature")



@pytest.fixture()
def setup_list():
    l1 = [100,50]
    return l1

@given("the list is not empty")
def check_not_empty(setup_list):
    print("\n In background checking list is not empty")
    if len(setup_list) == 0:
        pytest.xfail("the list is empty")

@given("the user have list of 2 numbers",)
def list(setup_list):
    print("user have list of two number",setup_list)

@when("the user removes one number from a list")
def added_numbers(setup_list):
    setup_list.pop()
    #print(setup_list)

@then("the length of list is 1")
```

```python
def list_length(setup_list):
    print("length of list",len(setup_list))
```

22.4 Tags in BDD:

3. **@bddscenario**:
   - This tag could be used to annotate a scenario in your feature file (usually written in Gherkin syntax) to mark it as a BDD scenario. For example:

```
@bddscenario
Scenario: Addition of two numbers
  Given I have entered 50 into the calculator
  And I have entered 70 into the calculator
  When I press add
  Then the result should be 120 on the screen
```

   - In pytest-bdd, you might use this tag to filter scenarios based on their purpose or category during test execution.
4. **@bddparan**:
   - The @bddparan tag isn't a standard BDD tag and might be specific to your project or framework conventions. Typically, tags like @param, @parameter, or similar are used to denote scenarios or steps that involve parameterized testing or varying inputs.