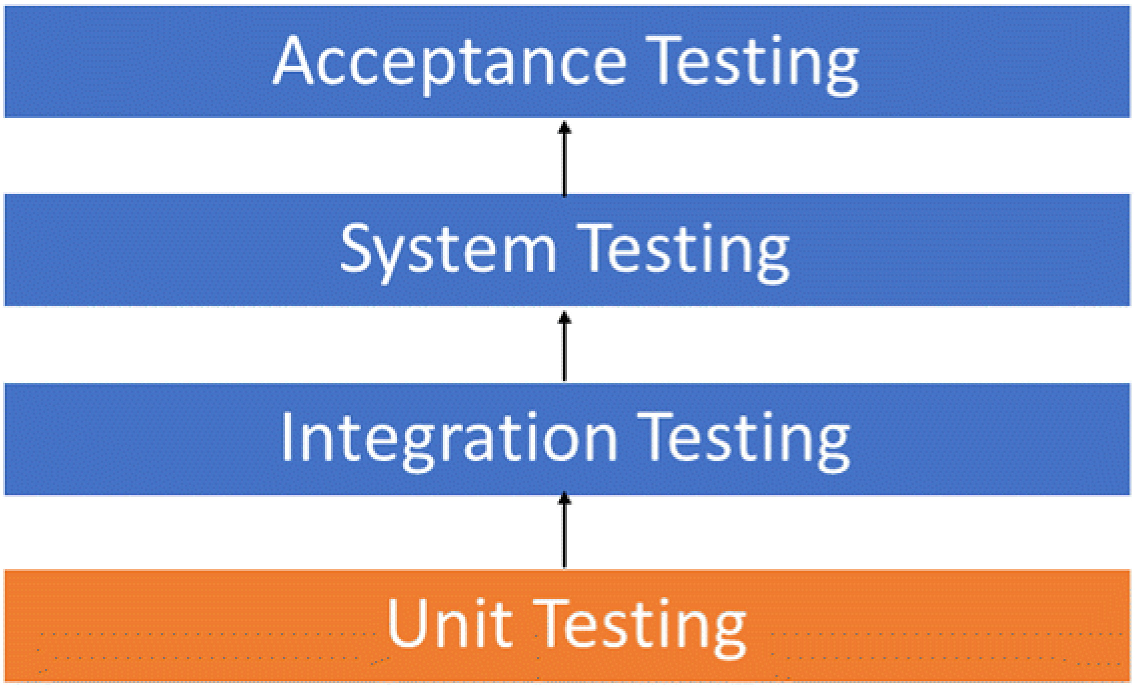
Unit Testing in Python (with pytest)

Software Testing

Software testing can be stated as the process of verifying and validating that a software or application is bug free, meets the technical requirements as guided by it's design and development and meets the user requirements effectively and efficiently with handling all the exceptional and boundary cases.

Levels of software testing:



1. **Unit Testing**: A level of the software testing process where individual units/components of a software/system are tested. The purpose is to validate that each unit of the software performs as designed.
2. **Integration Testing**: A level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.
3. **System Testing**: A level of the software testing process where a complete, integrated system/software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.
4. **Acceptance Testing**: A level of the software testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

Types of software testing:

1. **Manual Testing**: Manual testing includes testing a software manually, i.e., without using any automated tool or any script. In this type, the tester takes over the role of an end-user and tests the software to identify any unexpected behavior or bug.
2. **Automated Testing**: Automated testing is the execution of your test plan by a script instead of a human. It is used to re-run the manually performed test scenarios quickly and repeatedly.

Unit Testing in Python

The three most popular test runners in Python are:

unittest (https://docs.python.org/3/library/unittest.html)

Built into the Python standard library.

2 major requirements of `unittest` :

- Tests are written in a class which inherits from `unittest.TestCase` .
- Forced to use a series of special assertion methods in the `unittest.TestCase` class instead of the built-in assert statement

nose or nose2 (https://docs.nose2.io/en/latest/)

```
$ pip install nose2
```

Nose's tagline is " `nose` extends `unittest` to make testing easier".

`nose` is compatible with any tests written using the `unittest` framework and can be used as a drop-in replacement for the `unittest` test runner. The development of nose as an open-source application fell behind, and a fork called nose2 was created.

pytest (https://docs.pytest.org/en/latest/)

```
$ pip install pytest
```

`pytest` supports execution of `unittest` test cases. `pytest` test cases are a series of functions in a Python file starting with the name `test_` . Despite the fact that it reduces boilerplate code to the minimum, it still remains readable.

Other great features of pytest:

- Support for the built-in assert statement instead of using special `self.assert*()` methods
- Auto-discovery of test modules and functions
- Support for filtering of test cases
- Ability to rerun from the last failing test
- An ecosystem of hundreds of plugins to extend the functionality

PyTest (https://docs.pytest.org/en/latest/)

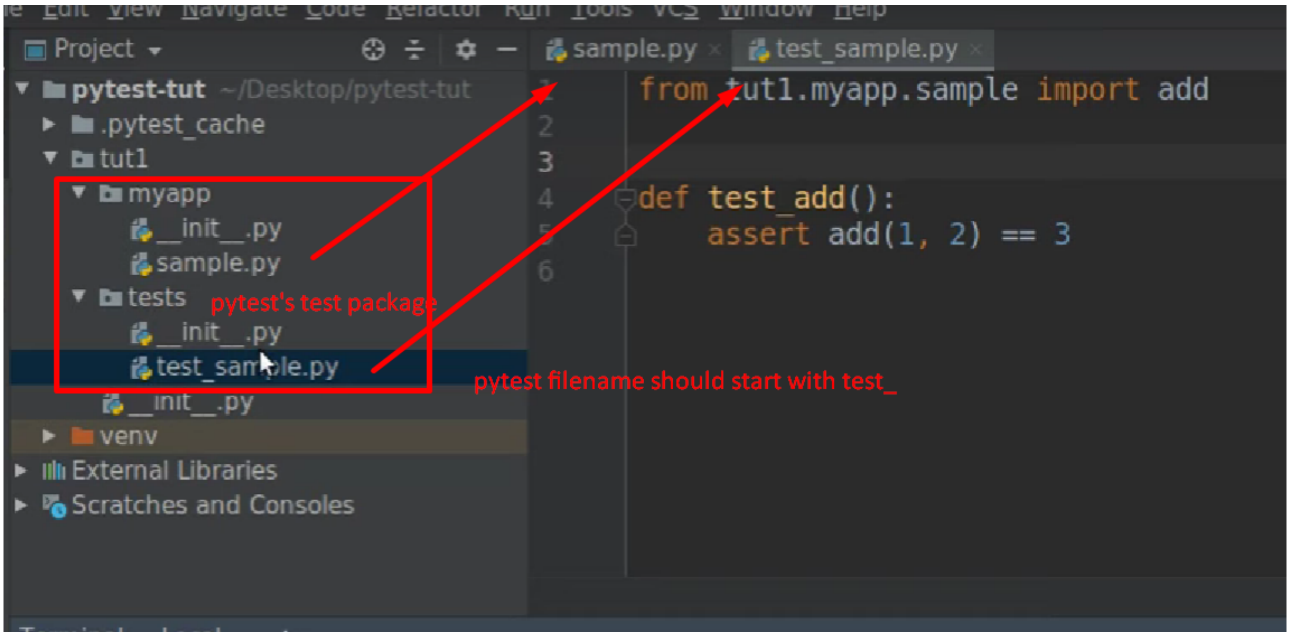
- Tutorial playlist (https://www.youtube.com/watch?v=L6N3BgZh2AA&list=PLyb_C2HpOQSBWGekd7PfhHnb9GnqDgrxS)
- Installation: `pip install pytest`

Points to remember:

1. class name must start with `Test` , like `TestSample` , `TestFunctions` etc..
    - accepted name: **Test, Test_, TestSample, Test_Sample**
    - not accepeted class name like : **testSample, Sample**
2. functions/methods name must start with `test` , like `test_sample()` , `test_login()` etc..

- accepted name: **test(), test_(), testSample(), test_anything()**
- not accepted names: **Test(), TestSample(), testsample **

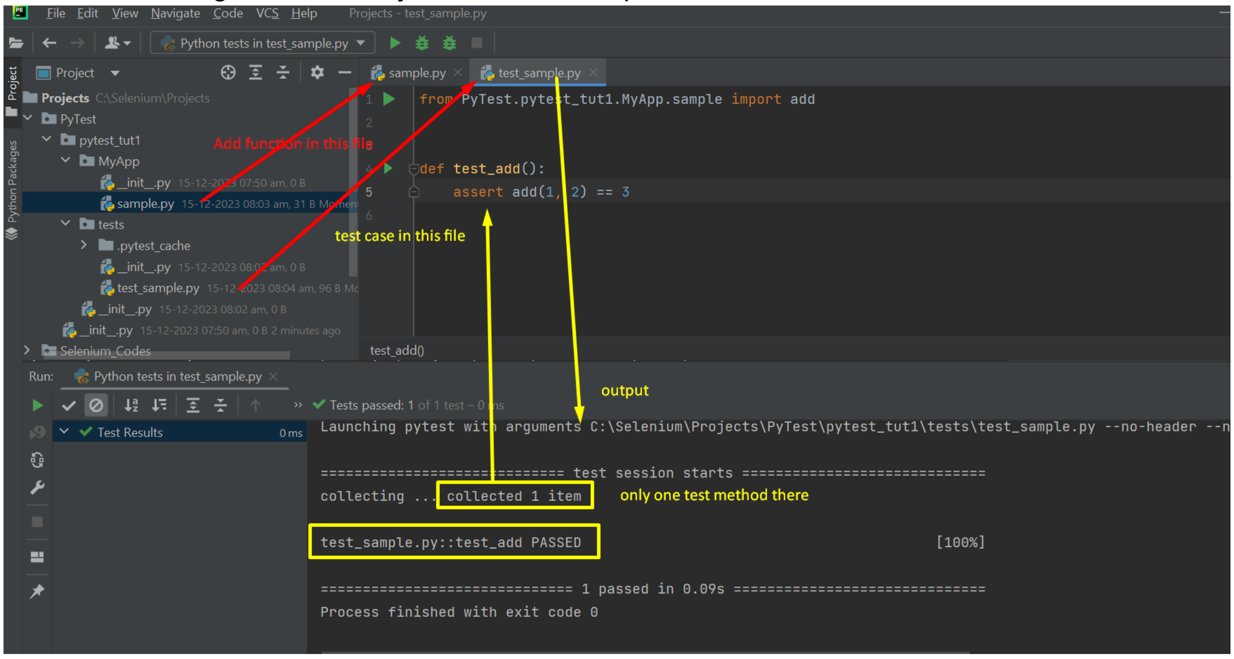
sample and test_sample



```
from tut1.myapp.sample import add


def test_add():
    assert add(1, 2) == 3
```

Run result

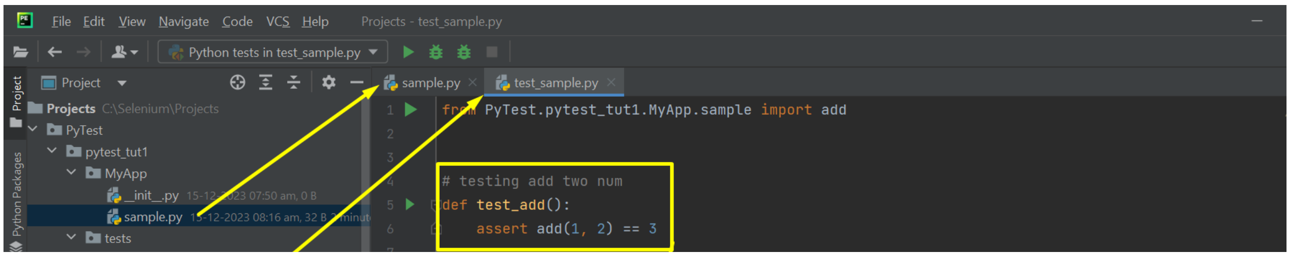
1. Run the test using terminal: just type `pytest` and hit enter
2. Run the test using run shortcut: just click the run option



```
from PyTest.pytest_tut1.MyApp.sample import add


def test_add():
    assert add(1, 2) == 3
```

```
Tests passed: 1 of 1 test – 0 ms
Launching pytest with arguments C:\Selenium\Projects\PyTest\pytest_tut1\tests\test_sample.py --no-header --n

============================ test session starts ============================
collecting ... collected 1 item

test_sample.py::test_add PASSED                                        [100%]

============================= 1 passed in 0.09s =============================
Process finished with exit code 0
```

Run two test cases


File Edit View Navigate Code VCS Help
Projects - test_sample.py
Python tests in test_sample.py
Project
sample.py
test_sample.py
Projects C:\Selenium\Projects
PyTest
pytest_tut1
MyApp
__init__.py 15-12-2023 07:50 am, 0 B
sample.py 15-12-2023 08:16 am, 32 B
tests
Python Packages

```
from PyTest.pytest_tut1.MyApp.sample import add


# testing add two num
def test_add():
    assert add(1, 2) == 3
```

Example:

sample.py

```
def add(a, b):
    return a+b
```

test_sample.py

```
from PyTest.pytest_tut1.MyApp.sample import add


# testing add two num
def test_add_num():
    assert add(1, 2) == 3


# Testing add two string
def test_add_str():
    assert add("Py", "Test") == "PyTest"


class TestSample:

    # testing add two num
    def test_add_num(self):
        assert add(1, 2) == 3

    # Testing add two string
    def test_add_str(self):
        assert add("Py","Test") == "PyTest"
```

Ouput:

```
Launching pytest with arguments C:\Selenium\Projects\PyTest\pytest_tut1\test
s\test_sample.py --no-header --no-summary -q in C:\Selenium\Projects\PyTest
\pytest_tut1\tests

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

test_sample.py::test_add PASSED                                          [ 2
5%]
test_sample.py::test_str PASSED                                          [ 5
0%]
test_sample.py::TestSample::test_add_num PASSED                          [ 7
5%]
test_sample.py::TestSample::test_add_str PASSED                          [10
0%]

============================== 4 passed in 0.05s =============================
===
Process finished with exit code 0
```

Understand the code

Your code seems to be a set of test cases for the `add` function from the `MyApp.sample` module using the pytest framework. Let's break it down:

1. **Importing the `add` function:**

   ```python
   from PyTest.pytest_tut1.MyApp.sample import add
   ```

   This line imports the `add` function from the `MyApp.sample` module.
2. **Testing the `add` function with integers:**

   ```python
   def test_add_num():
       assert add(1, 2) == 3
   ```

   This is a simple test case (`test_add_num`) using the `assert` statement to check if the result of `add(1, 2)` is equal to `3` . If it's not, the test will fail.
3. **Testing the `add` function with strings:**

   ```python
   def test_add_str():
       assert add("Py", "Test") == "PyTest"
   ```

   Similar to the first test, this tests whether the `add` function can concatenate the strings "Py" and "Test" to form "PyTest".
4. **Defining a test class `TestSample` :**

   ```python
   class TestSample:
   ```

   This class contains additional test methods.
5. **Testing the `add` function with integers inside the class:**

   ```python
   def test_add_num(self):
       assert add(1, 2) == 3
   ```

   This is the same as the first test, but it's placed inside a class called `TestSample` . When using classes for tests, each test method should start with the word "test".

6. **Testing the `add` function with strings inside the class:**

    ```
    def test_add_str(self):
        assert add("Py", "Test") == "PyTest"
    ```

    Similar to the second test, this is placed inside the `TestSample` class and checks whether the `add` function can concatenate strings.

In summary, this code is a set of test cases for the `add` function, checking both numerical and string inputs. The tests are organized into individual functions and a test class. To run these tests, you would use the `pytest` command in your terminal in the directory where this file is located. If the assertions hold true for all cases, your tests will pass. If any assertion fails, pytest will report which test failed and what the expected and actual values were.

```
from PyTest.pytest_tut1.MyApp.sample import add


# testing add two num
def test_add_num():
    assert add(1, 2) == 3


# Testing add two string
def test_add_str():
    assert add("Py", "Test") == "PyTest"


class TestSample:

    # testing add two num
    def test_add_num(self):
        assert add(1, 2) == 3

    # Testing add two string
    def test_add_str(self):
        assert add("Py","Test") == "PyTest"


def testSample():
    assert add(2.4, 5) == 7.4


# Wrong test case
def test_add2():
    assert add(3, 7) == 11


def test_add3():
    assert add("Py", "Test") == "Pytest"
```

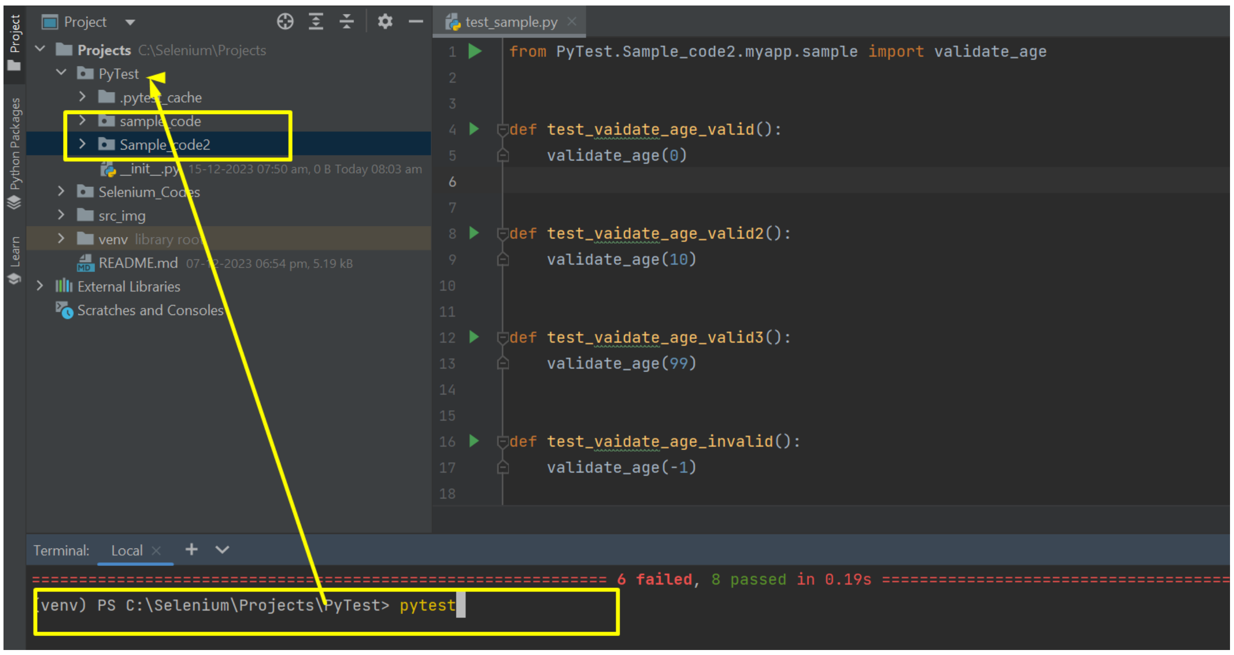
Output

```
Launching pytest with arguments C:\Selenium\Projects\PyTest\pytest_tut1\test
s\test_sample.py --no-header --no-summary -q in C:\Selenium\Projects\PyTest
\pytest_tut1\tests

============================= test session starts ==========================
===
collecting ... collected 7 items

test_sample.py::test_add_num PASSED                                      [ 1
4%]
test_sample.py::test_add_str PASSED                                      [ 2
8%]
test_sample.py::TestSample::test_add_num
test_sample.py::TestSample::test_add_str
test_sample.py::testSample PASSED         [ 42%]PASSED
[ 57%]PASSED                                          [ 71%]
test_sample.py::test_add2 FAILED                                         [ 8
5%]
test_sample.py:29 (test_add2)
10 != 11

11
10
<Click to see difference>

def test_add2():
>       assert add(3, 7) == 11
E       assert 10 == 11
E        +  where 10 = add(3, 7)

test_sample.py:31: AssertionError

test_sample.py::test_add3 FAILED                                         [10
0%]
test_sample.py:33 (test_add3)
'PyTest' != 'Pytest'

'Pytest'
'PyTest'
<Click to see difference>

def test_add3():
>       assert add("Py", "Test") == "Pytest"
E       AssertionError: assert 'PyTest' == 'Pytest'
E         - Pytest
E         ?   ^
E         + PyTest
E         ?   ^

test_sample.py:35: AssertionError


========================= 2 failed, 5 passed in 0.22s ======================
===
Process finished with exit code 1
```

Output from Terminal run

```
(venv) PS C:\Selenium\Projects\PyTest\pytest_tut1> pytest
================================ test session starts ========================
===========
platform win32 -- Python 3.8.2, pytest-7.4.3, pluggy-1.3.0
rootdir: C:\Selenium\Projects\PyTest\pytest_tut1
collected 7 items

tests\test_sample.py .....FF
[100%]
================== FAILURES ======================================
_______________________ test_add2 _____________________________________

    def test_add2():
>       assert add(3, 7) == 11
E       assert 10 == 11
E        +  where 10 = add(3, 7)

tests\test_sample.py:35: AssertionError

===================== short test summary info ==============================
=====
FAILED tests/test_sample.py::test_add2 - assert 10 == 11
FAILED tests/test_sample.py::test_add3 - AssertionError: assert 'PyTest' ==
'Pytest'

======================== 2 failed, 5 passed in 0.25s ==================
(venv) PS C:\Selenium\Projects\PyTest\pytest_tut1>
```

- To run the test case from multiple packages: **run using terminal/CLI**



```
from PyTest.Sample_code2.myapp.sample import validate_age


def test_vaidate_age_valid():
    validate_age(0)


def test_vaidate_age_valid2():
    validate_age(10)


def test_vaidate_age_valid3():
    validate_age(99)


def test_vaidate_age_invalid():
    validate_age(-1)
```

```
============================================================ 6 failed, 8 passed in 0.19s ======================================
(venv) PS C:\Selenium\Projects\PyTest> pytest
```

Assertion Exception/Error handling

app.py

```
def validate_login(username, password):

    if type(username) == str and type(password) == str:
        if username == "admin" and password == "admin123":
            print("Login Success")

        else:
            raise ValueError("Login denied, username or password maybe wron
g.")

    else:
        raise ValueError("Username and password should be string.")
```

test.py

```python
import pytest

from PyTest.Assert_Exception.app.Assertion_app import validate_login


# valid test will pass
def test_validate_login_valid_test():
    validate_login("admin","admin123")


# invalid test will fail with error
def test_validate_login_invalid_test():
    validate_login("admin2", "admin123")


# checking that invalid test is invalid, so it will pass w/o error
def test_validate_login_invalid_test2():
    with pytest.raises(ValueError):
        validate_login("admin2", "admin123")


# invalid test is invalid, so it will pass with error info
def test_validate_login_invalid_test3():
    with pytest.raises(ValueError) as exc_info:
        validate_login("admin2", "admin123")
    print(str(exc_info.value))


# invalid test is invalid, match with expected error; pass
def test_validate_login_invalid_test4():
    with pytest.raises(ValueError) as exc_info:
        validate_login("admin2", "admin123")
    assert str(exc_info.value) == "Login denied, username or password maybe
wrong."


# invalid test is invalid, match with unexpected error: fail
def test_validate_login_invalid_test5():
    with pytest.raises(ValueError) as exc_info:
        validate_login("admin2", "admin123")
    assert str(exc_info.value) == "Login success, username or password maybe
wrong."


# invalid test is invalid, match with expected error: passa
def test_validate_login_invalid_test6():
    with pytest.raises(ValueError, match="Login denied, username or password
maybe wrong."):
        validate_login("admin2", "admin123")
```

Output

```
Launching pytest with arguments C:\Selenium\Projects\PyTest\Assert_Exception
\tests\test_assertions.py --no-header --no-summary -q in C:\Selenium\Project
s\PyTest\Assert_Exception\tests

============================= test session starts ==========================
===
collecting ... collected 7 items

test_assertions.py::test_validate_login_valid_test PASSED                [ 1
4%]Login Success

test_assertions.py::test_validate_login_invalid_test FAILED              [ 2
8%]
test_assertions.py:11 (test_validate_login_invalid_test)
def test_validate_login_invalid_test():
>       validate_login("admin2", "admin123")

test_assertions.py:13:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _

username = 'admin2', password = 'admin123'

    def validate_login(username, password):

        if type(username) == str and type(password) == str:
            if username == "admin" and password == "admin123":
                print("Login Success")

            else:
>               raise ValueError("Login denied, username or password maybe w
rong.")
E               ValueError: Login denied, username or password maybe wrong.

..\app\Assertion_app.py:8: ValueError

test_assertions.py::test_validate_login_invalid_test2 PASSED             [ 4
2%]
test_assertions.py::test_validate_login_invalid_test3 PASSED             [ 5
7%]Login denied, username or password maybe wrong.

test_assertions.py::test_validate_login_invalid_test4 PASSED             [ 7
1%]
test_assertions.py::test_validate_login_invalid_test5 FAILED             [ 8
5%]
test_assertions.py:36 (test_validate_login_invalid_test5)
'Login denied, username or password maybe wrong.' != 'Login success, usernam
e or password maybe wrong.'

'Login success, username or password maybe wrong.'
'Login denied, username or password maybe wrong.'
<Click to see difference>

def test_validate_login_invalid_test5():
        with pytest.raises(ValueError) as exc_info:
            validate_login("admin2", "admin123")
>       assert str(exc_info.value) == "Login success, username or password m
aybe wrong."
E       AssertionError: assert 'Login denied, username or password maybe wro
ng.' == 'Login success, username or password maybe wrong.'
E         - Login success, username or password maybe wrong.
E         ?       ^^^^ ^^
E         + Login denied, username or password maybe wrong.
E         ?       ^ ^^^^

test_assertions.py:40: AssertionError

test_assertions.py::test_validate_login_invalid_test6 PASSED             [10
```

Parametrized Marker

The code you provided is using the `@pytest.mark.parametrize` decorator to parameterize a test function. This allows you to run the same test logic with different sets of input values. Here's a breakdown:

```python
# parametrize: we can create own parameters and pass to pytest function
@pytest.mark.parametrize("a,b,c", [(1, 2, 3), ('Hello ', 'Pytest', "Hello Py
test")])
def test_valid_add(a, b, c):
    assert add(a, b) == c
```

1. **Importing the `pytest` module:**

   ```python
   import pytest
   ```

   This line imports the `pytest` module, which is necessary for using the `@pytest.mark.parametrize` decorator.
2. **Parameterized Test Function:**

   ```python
   @pytest.mark.parametrize("a,b,c", [(1, 2, 3), ('Hello ', 'Pytest', "Hell
   o Pytest")])
   def test_valid_add(a, b, c):
       assert add(a, b) == c
   ```

   - **`@pytest.mark.parametrize("a,b,c", [(1, 2, 3), ('Hello ', 'Pytest', "Hello Pytest")])`:**

     This decorator specifies a set of parameters (`a` , `b` , `c`) and the corresponding values. In this case, the test function will be run twice: once with the values `(1, 2, 3)` and once with `('Hello ', 'Pytest', "Hello Pytest")` .
   - **`def test_valid_add(a, b, c):`:**

     This is the test function. It takes three parameters (`a` , `b` , `c`) corresponding to the parameters specified in the `@pytest.mark.parametrize` decorator.
   - **`assert add(a, b) == c`:**

     This is the assertion within the test function. It checks whether the result of calling the `add` function with the parameters `a` and `b` is equal to the expected result `c` . If the assertion fails for any set of parameters, the test will fail.

In summary, this code defines a test function (`test_valid_add`) that is run multiple times with different sets of parameters specified by the `@pytest.mark.parametrize` decorator. It's a concise way to test the `add` function with various inputs and expected outputs.

```
def test_validate_add_num():
    assert add(5, 10) == 15


def test_validate_add_str():
    assert add('Hello ', 'Pytest') == "Hello Pytest"


def test_validate_add_list():
    assert add([1, 2], [3]) == [1, 2, 3]
```

Output

```
============================= test session starts ==========================
===
collecting ... collected 3 items

test_parametrize.py::test_validate_add_num PASSED                        [ 3
3%]
test_parametrize.py::test_validate_add_str PASSED                        [ 6
6%]
test_parametrize.py::test_validate_add_list PASSED                       [10
0%]

============================== 3 passed in 0.03s ===========================
===
```

Above three test cases is similar to below

```
# parametrize: we can create own parameters and pass to pytest function
@pytest.mark.parametrize("a,b,c", [(1, 2, 3), ('Hello ', 'Pytest', "Hello Py
test"), ([1, 2], [3], [1, 2, 3])])
def test_valid_add(a, b, c):
    assert add(a, b) == c
```

Output

```
============================= test session starts ==========================
===
collecting ... collected 3 items

test_parametrize.py::test_valid_add[with int] PASSED                     [ 3
3%]
test_parametrize.py::test_valid_add[with str] PASSED                     [ 6
6%]
test_parametrize.py::test_valid_add[with list] PASSED                    [10
0%]

============================== 3 passed in 0.03s ===========================
===
```

or

```
# parametrize: we can create own parameters and pass to pytest function
@pytest.mark.parametrize("a,b,c", [(1, 2, 3), ('Hello ', 'Pytest', "Hello Py
test"), ([1, 2], [3], [1, 2, 3])], ids=['with int', 'with str', 'with lis
t'])
def test_valid_add(a, b, c):
    assert add(a, b) == c
```

Output

```
============================= test session starts ==========================
===
collecting ... collected 3 items

test_parametrize.py::test_valid_add[with int] PASSED                     [ 3
3%]
test_parametrize.py::test_valid_add[with str] PASSED                     [ 6
6%]
test_parametrize.py::test_valid_add[with list] PASSED                    [10
0%]

============================== 3 passed in 0.02s ===========================
===
```

Fixtures Marker

The error you're encountering indicates that pytest couldn't find a fixture named `tempdir` when running your test. Fixtures are functions that provide data or setup code to your tests, and they need to be properly defined.

In your case, it seems like you are trying to use a fixture named `tempdir`, but it's not available. To use the `tempdir` fixture, you need to make sure that it's either built-in (pytest provides some default fixtures) or that you've defined it yourself.

Here's an example of how you might define the `tempdir` fixture in a pytest test file:

```
import json
import os
import pytest

from PyTest.Fixtures_Marker.app.sample import save_dict

# Define the tempdir fixture
@pytest.fixture
def tempdir(tmpdir):
    return tmpdir

def test_save_dict(tempdir):
    filepath = os.path.join(tempdir, "test.json")
    d = {1: 'pytest', 2: 'python'}
    save_dict(d, filepath)
    assert json.load(open(filepath, 'r')) == d
```

In this example:

- `@pytest.fixture` is used to define a fixture named `tempdir`.
- The fixture function takes `tmpdir` as an argument. `tmpdir` is a built-in pytest fixture that provides a temporary directory unique to each test.
- The `test_save_dict` function now takes `tempdir` as a parameter, which will automatically use the value provided by the `tempdir` fixture.

Make sure to include `import pytest` at the beginning of your file if you haven't already. If you've already defined a `tempdir` fixture elsewhere in your code, make sure that it's correctly imported or defined in the same file.

If you still encounter issues, you might want to check the pytest documentation or your project's specific fixture definitions to ensure the availability of the `tempdir` fixture.

test-fixtures.py

```
# Fixtures
import json
import os
import pytest

from PyTest.Fixtures_Marker.app.sample import save_dict


# Define the tempdir fixture
@pytest.fixture
def tempdir(tmpdir):
    return tmpdir


def test_save_dict(tempdir, capsys):
    filepath = os.path.join(tempdir, "test.json")
    d = {'1': 'pytest', '2': 'python'}
    save_dict(d, filepath)
    assert json.load(open(filepath, 'r')) == d
    assert capsys.readouterr().out == "Saved\n"
```

output

```
============================= test session starts ==========================
===
collecting ... collected 1 item

test_fixtures.py::test_save_dict PASSED                                 [10
0%]

============================== 1 passed in 0.03s ===========================
===
```

```
In [ ]:
```