Unit Testing Framework for Python [PyTest]

What is PyTest?

> *Unit testing framework for Python (UnitTest or Nose)*
>
> *Options to conditionally execute cases*
>
> *Can setup pre-requisite and post script*
>
> *Can Add Assertions* *compares actual result with expected result*
>
> *Options to generate report*

Install PyTest

Command: pip install pytest

Write TestCase in PyTest Format

Note: *To execute test cases simultaneously, ensure that the file name and method names start with `**tests**`.*

- **Create a New Project:** Begin by creating a new directory to serve as the root of your project. Let's name it "PytestLearningt".
- **Create Python File:** Inside your project directory, create a Python file where you'll write your test cases. Name it something descriptive, like "test_TC001_First.py".
- **Write Test Case Method:** In "test_suite.py", write your test case methods using PyTest, which is known for its simplicity and readability. Here's an example of a basic PyTest test case:

test_TC001_First.py

```
import pytest

def tc_test_addition():
    assert 2 + 2 == 4
```

- **Change Method Name for PyTest Compatibility:** Adjust the method name to conform to PyTest's naming convention. PyTest identifies test functions based on their name prefixes. For instance, rename `tc_test_addition` to `test_addition`:

test_TC001_First.py

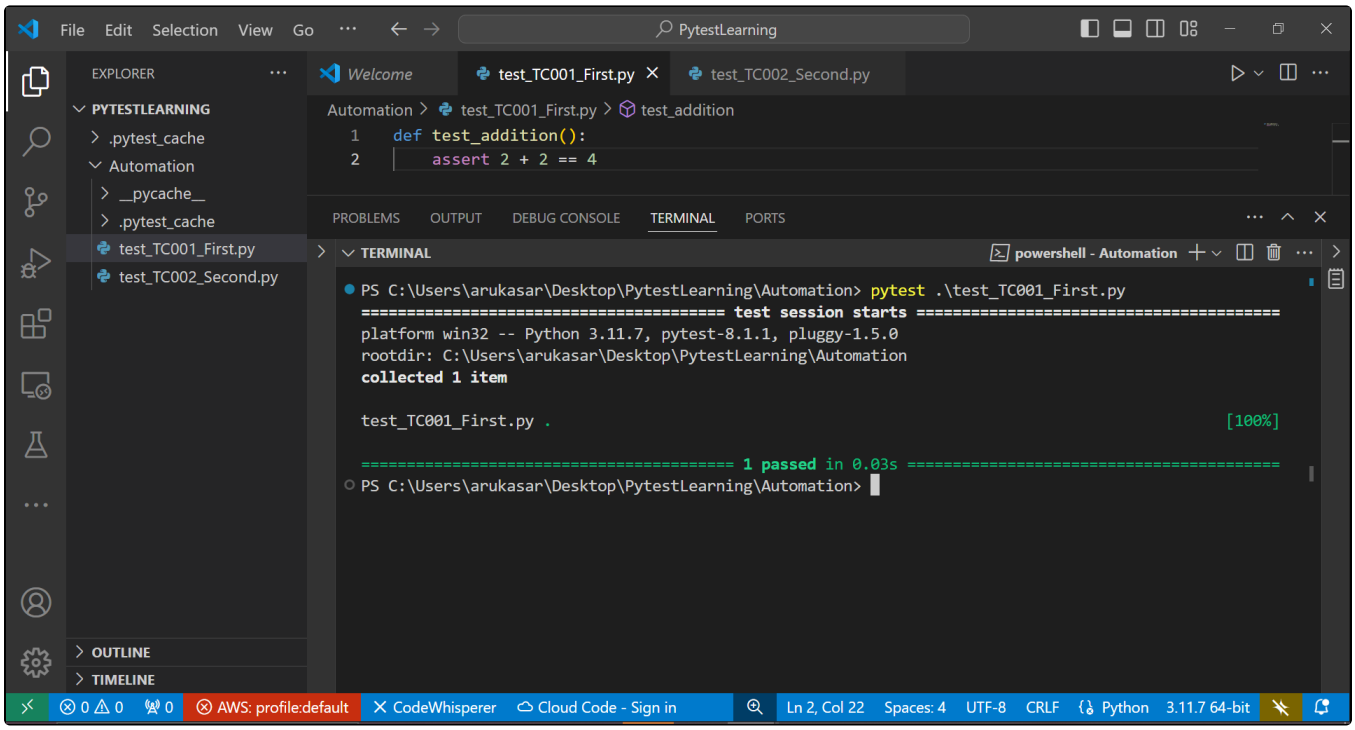
```
//Test case code must be written inside a method
//Method name must be started with test

import pytest

def test_addition():
    assert 2 + 2 == 4
```

- **Run Test Case using PyTest:** Navigate to your project directory in the terminal and execute your test cases using PyTest:

command to run single test case

```
pytest test_TC001_First.py
```

Results:

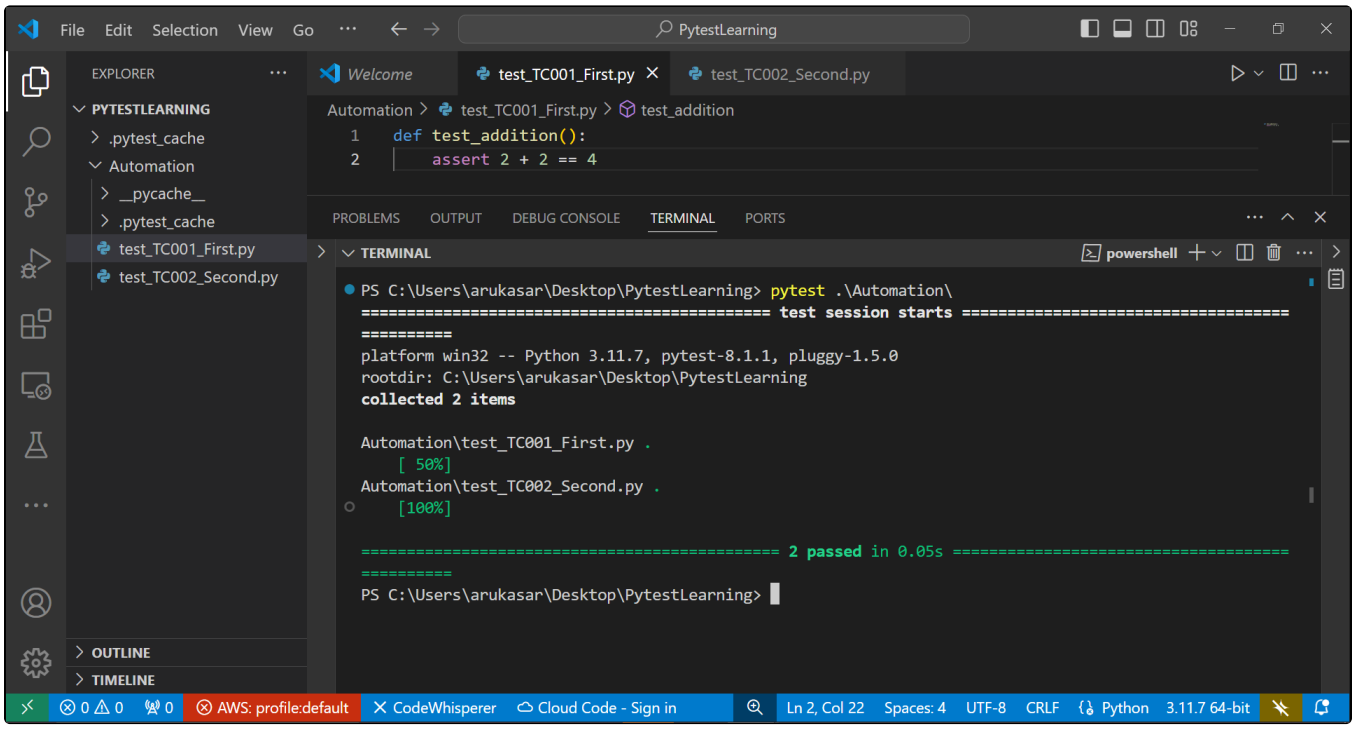


- **Write Test Cases Inside a Folder:** To maintain better project structure and organization, create a folder named "tests" within your project directory. Then, move "test_suite.py" into this folder.
- **Execute All Test Cases Inside the Folder:** Run all test cases located within the "tests" folder by providing the folder path to PyTest:

command to run all the test case inside the given folder

```
pytest Automation/
```

Results:



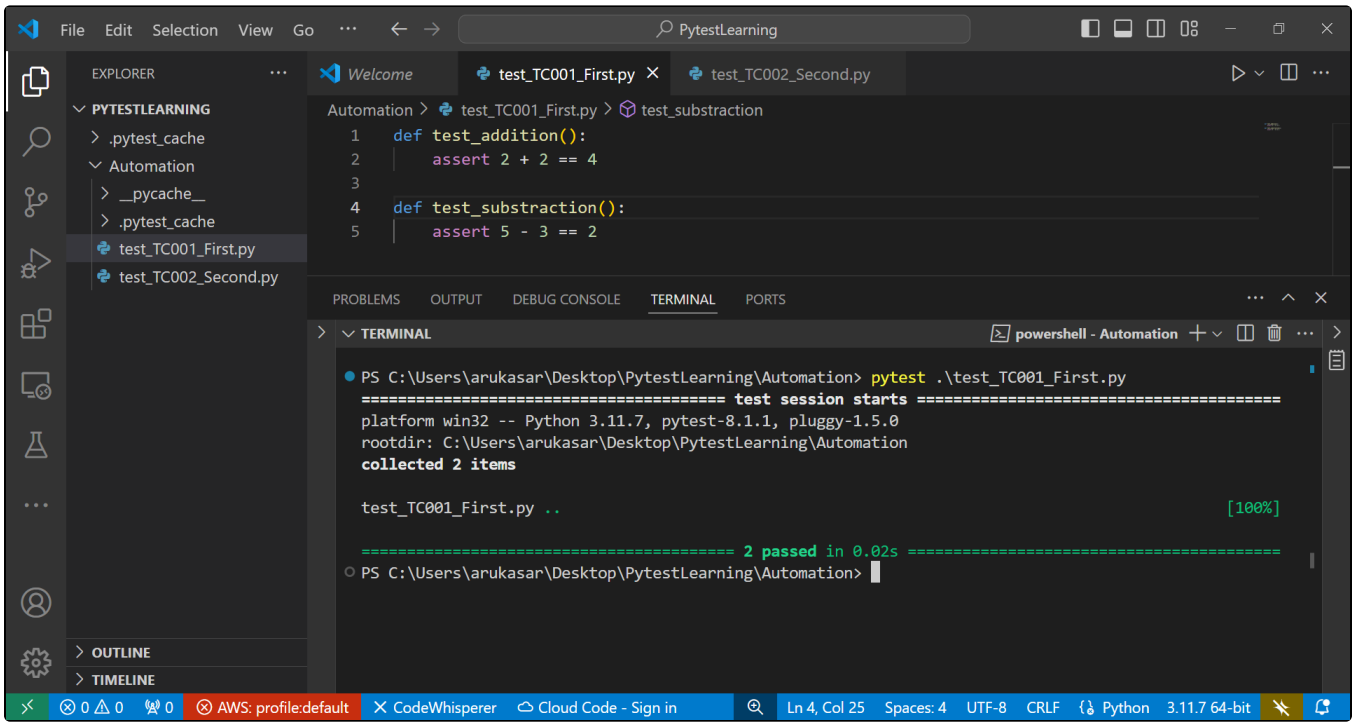
Write Multiple Test Cases in a File: Check Execution Options

test_TC001_First.py

```
def test_addition():
    assert 2 + 2 == 4

def test_substraction():
    assert 5 - 3 == 2
```

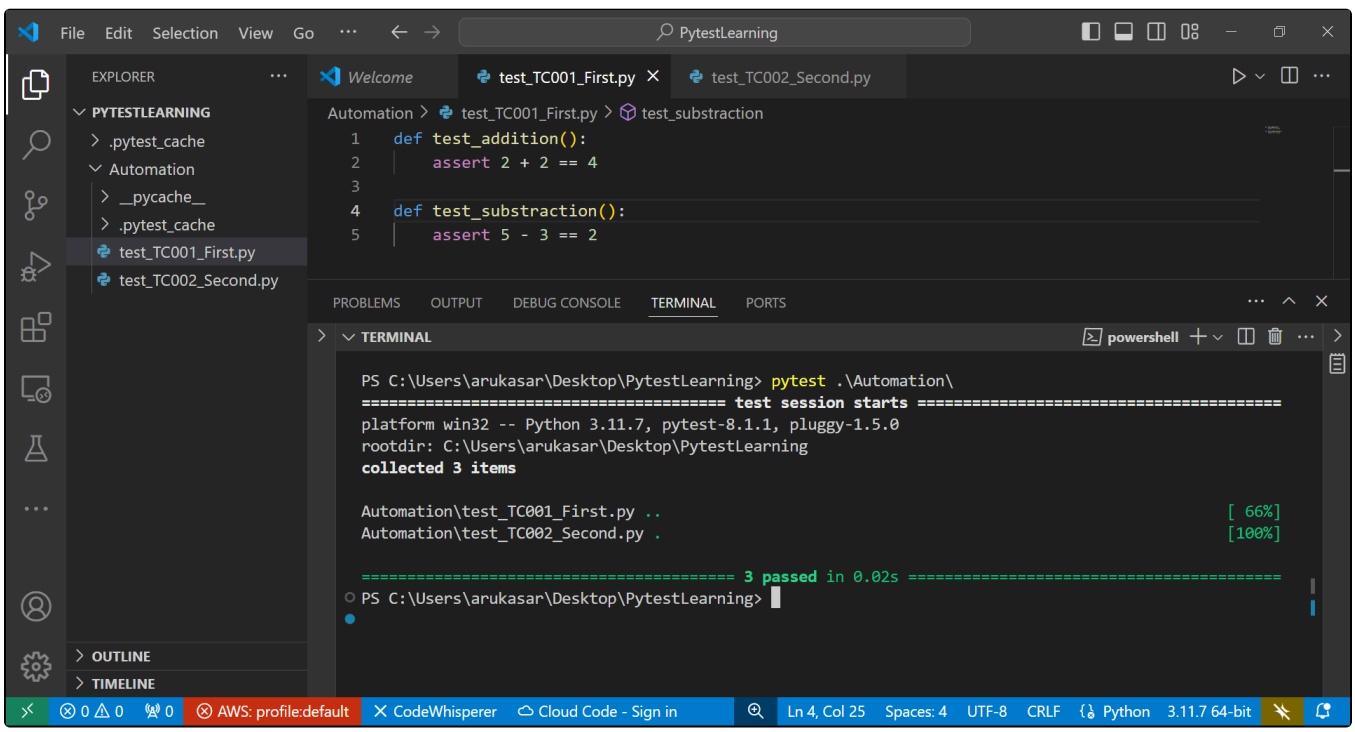
command to run multiple test cases in a single file

```
pytest <file_name>
```



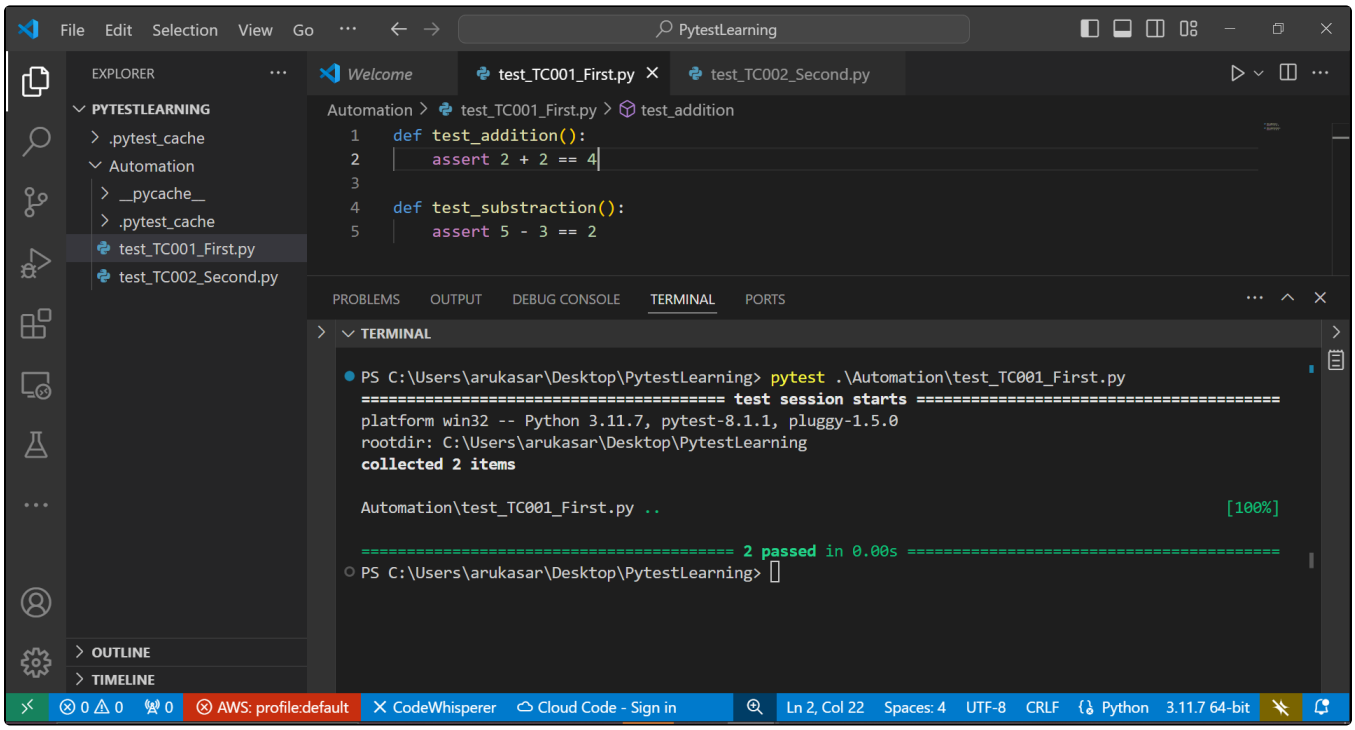
command to run multiple test cases in a single file

```
pytest <folder_name>
```



command to run multiple test cases in a single file

```
pytest <folder_name><file_name>
```



Execute Test Case using verbos(-v)

Here's what each option does:

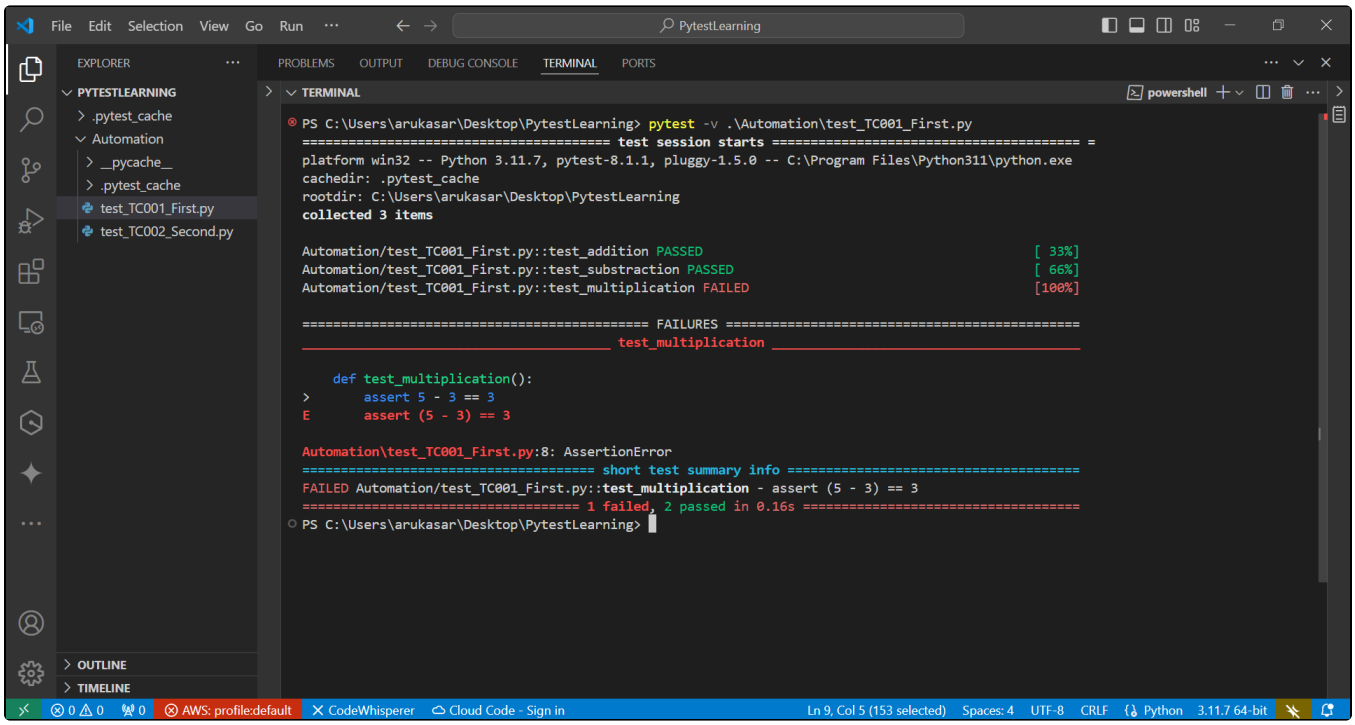
-s: Allows printing output to the console. This option is useful for displaying print statements within your test code.
-v: Enables verbose mode, which provides more detailed output about the test execution process, including the names of the tests being run and their outcomes.

test_TC001_First.py

```
def test_addition():
    assert 2 + 2 == 4

def test_substraction():
    assert 5 - 3 == 2

def test_multiplication():
    assert 5 - 3 == 3
```

verbos command

```
pytest -v <file_name>
```

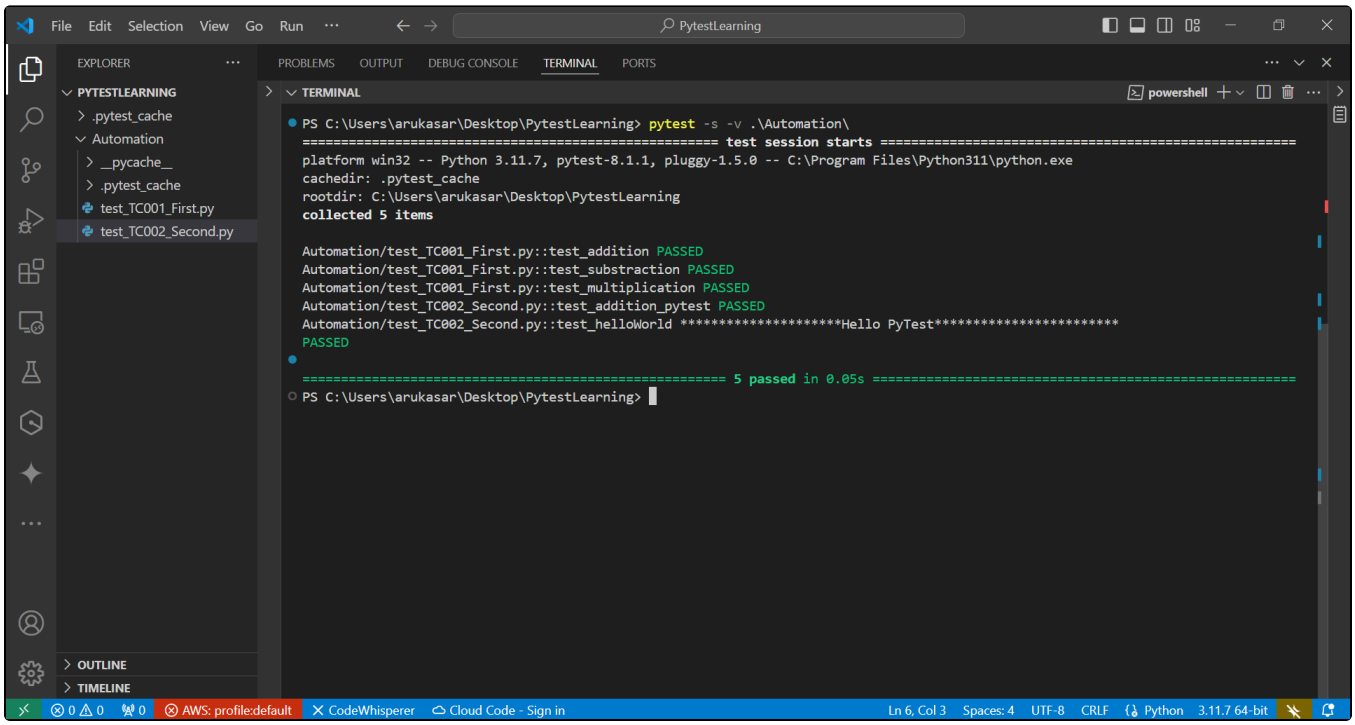


test_TC002_Second.py

```
def test_addition_pytest():
    assert 2 + 6 == 8

def test_helloWorld():
    print("********************Hello PyTest************************")
```

print statement command

```
pytest -s -v <file_name>
```



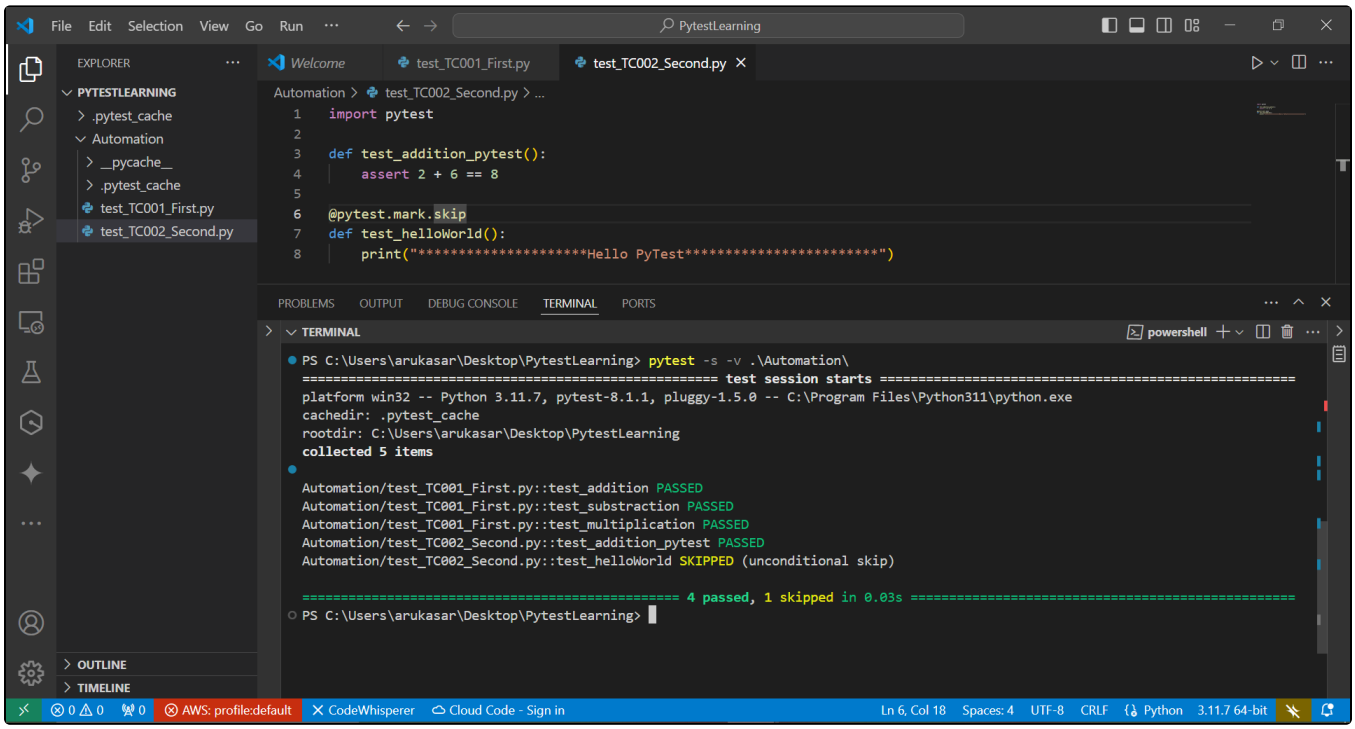
Skip execution of any specific test cases

test_TC002_Second.py

```
import pytest

def test_addition_pytest():
    assert 2 + 6 == 8

@pytest.mark.skip
def test_helloWorld():
    print("**********Hello PyTest**********")
```

print statement command

```
pytest -s -v <file_name>
```

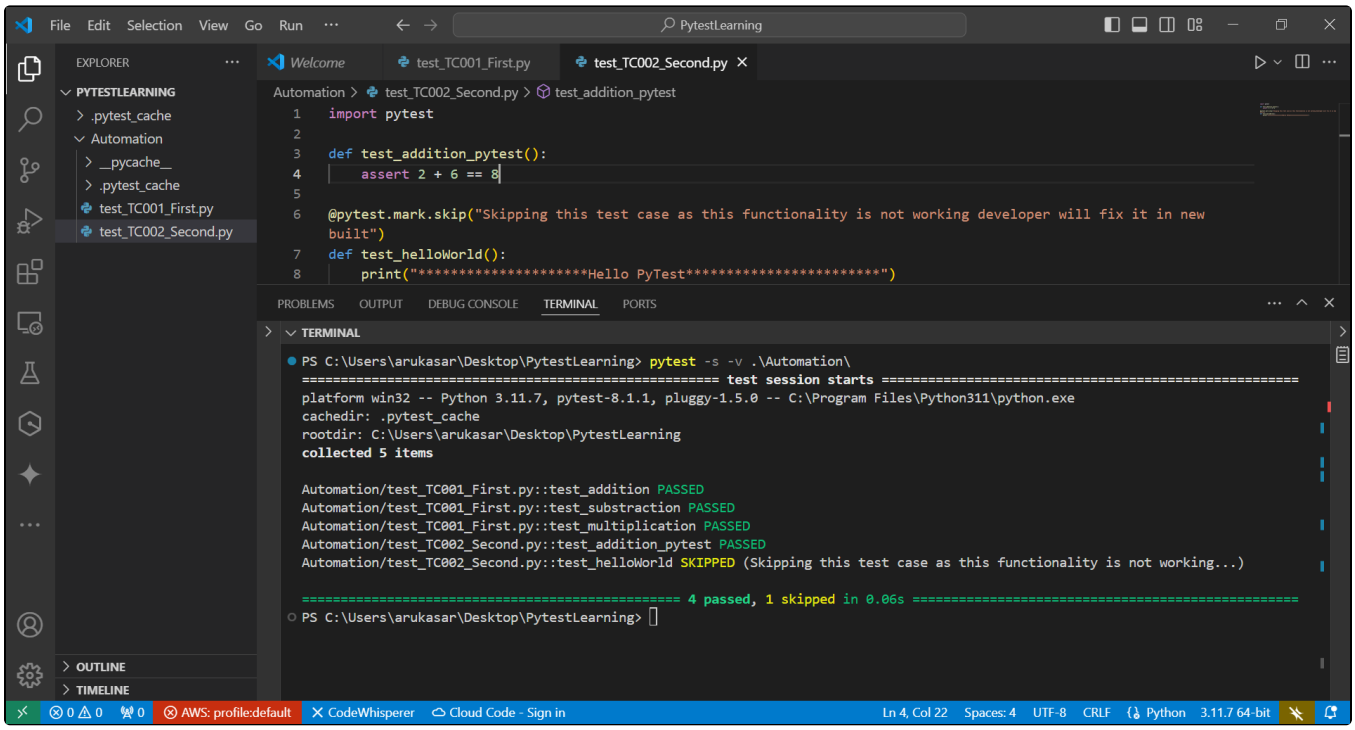


test_TC002_Second.py

```
import pytest

def test_addition_pytest():
    assert 2 + 6 == 8

//Decorator
@pytest.mark.skip("Skipping this test case as this functionality is not working developer will fix it in new
built")
def test_helloWorld():
    print("********************Hello PyTest************************")
```

print statement command

```
pytest -s -v <file_name>
```



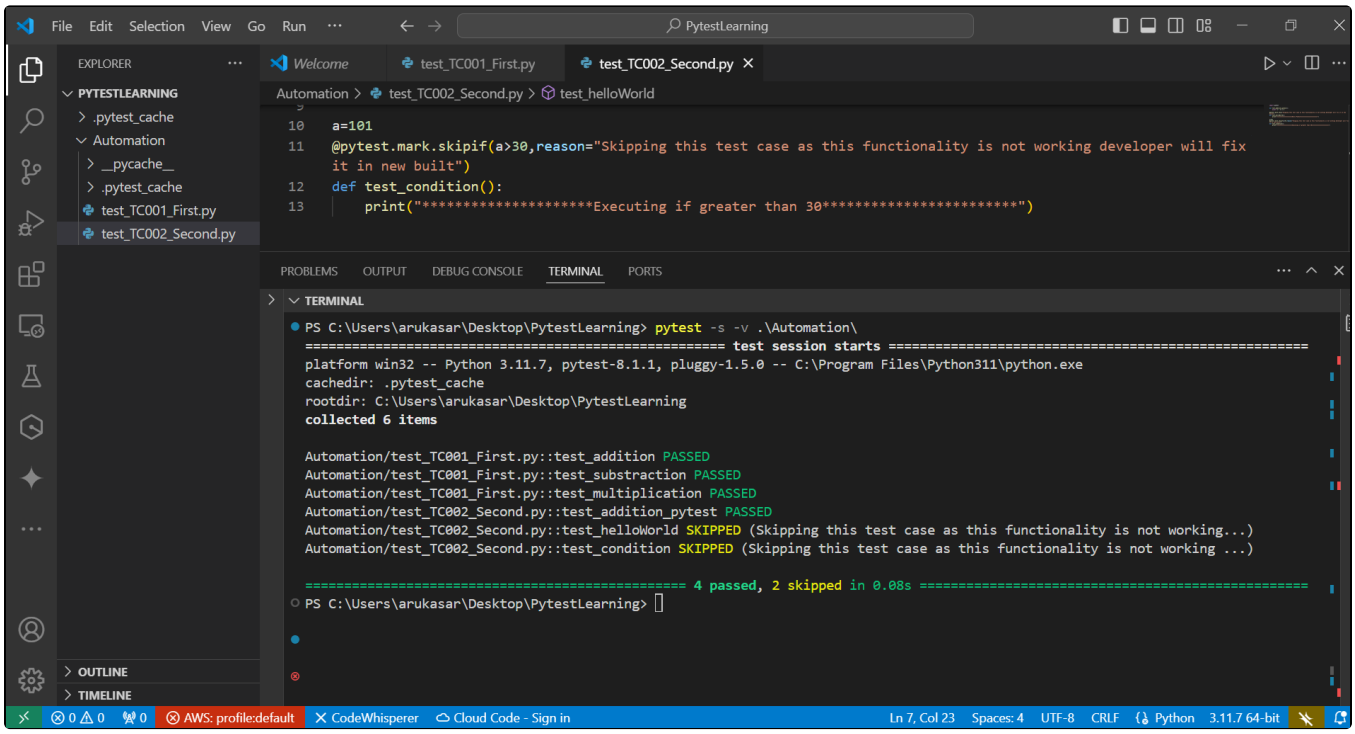
Conditionally skip execution of any specific test cases

test_TC002_Second.py

```
import pytest

def test_addition_pytest():
    assert 2 + 6 == 8

@pytest.mark.skip("Skipping this test case as this functionality is not working developer will fix it in new built")
def test_helloWorld():
    print("********************Hello PyTest************************")

a=101
@pytest.mark.skipif(a>30,reason="Skipping this test case as this functionality is not working developer will fix it in new built")
def test_condition():
    print("********************Executing if greater than 30************************")
```

print statement command

```
pytest -s -v <file_name>
```



Execute only specific test cases

Approach 1:

- You can execute specific test cases in pytest using the -k option, which allows you to specify a substring expression to match test names against. Here's how you can do it:
- Replace "test_specific_case" with the substring that matches the test case name(s) you want to execute. This will run only the test cases whose names contain the specified substring.

command to execute specific test case
`pytest -k "test_specific_case"`

Approach 2:

- Alternatively, if you have marked your test cases with custom markers, you can use the -m option to execute only the tests marked with a specific marker. For example:
- Replace "specific_marker" with the name of the marker you've defined for the test case(s) you want to execute.

command to execute specific test case using marker
`pytest -m specific_marker`

Approach 3:

- You can also combine multiple options. For example, if you want to execute only the test cases with a specific substring in their names and marked with a specific marker, you can do:
- This command will execute only the test cases that match both the substring expression and the specified marker.

command to execute specific test case using marker
`pytest -k "test_specific_case" -m specific_marker`

Tagging/Grouping

> *Define tags for the test cases*
> *1. Write tags to test cases*
> *2. Execute test cases with single tags*
> *3. Skip execution of test cases by giving tag name*
> *4. Write multiple tags on test cases*
> *5. Execute test cases using more than 1 tag(or and)*

Writing Tags for Test Cases: Assign tags to your test cases using decorators or markers in pytest. These tags can represent various attributes like functionality, priority, environment, etc.

test_TC003_Third.py

```
import pytest

@pytest.mark.smoke
def test_addition():
    print("This is Smoke Test")
    assert 2 + 2 == 4

@pytest.mark.sanity
def test_substraction():
    print("This is Sanity Test")
    assert 5 - 3 == 2

@pytest.mark.smoke
def test_multiplication():
    print("This is Smoke Test")
    assert 5 - 3 == 2
```

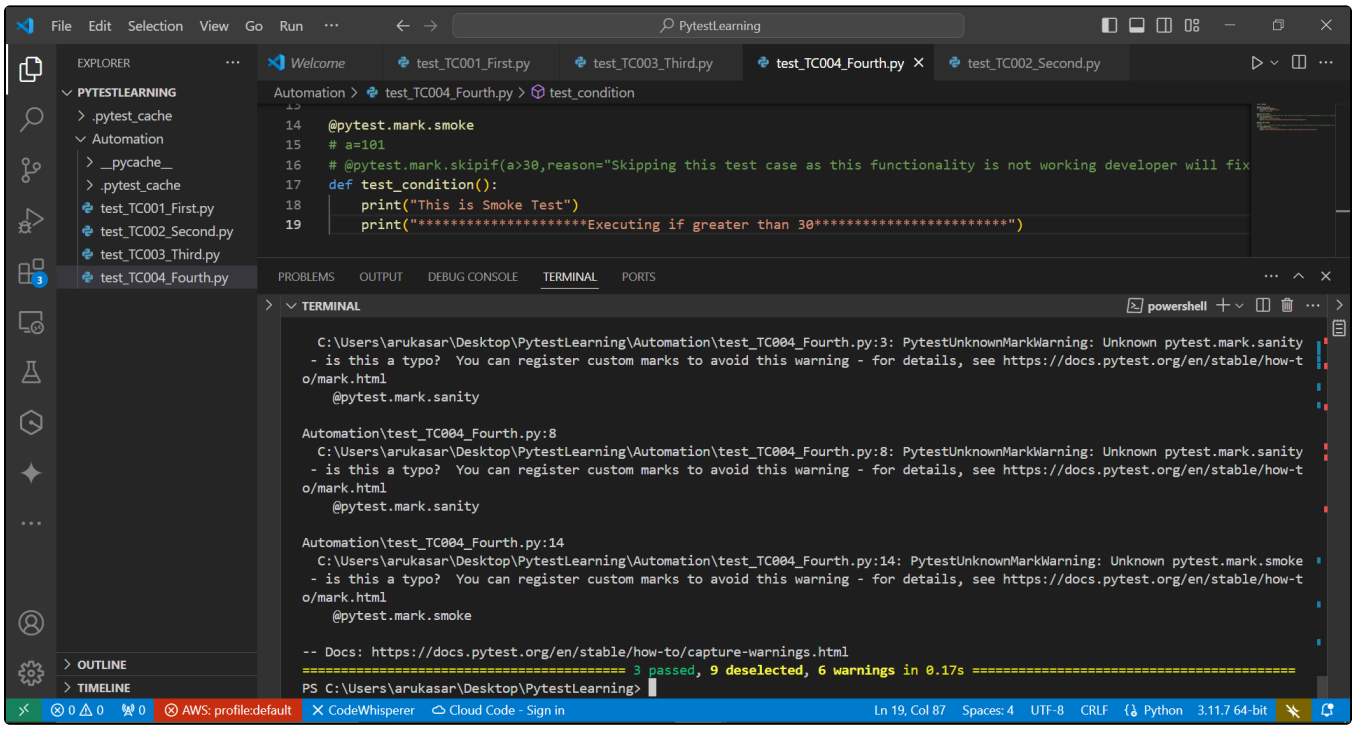
test_TC004_Fourth.py

```
import pytest

@pytest.mark.sanity
def test_addition_pytest():
    print("This is Sanity Test")
    assert 2 + 6 == 8

@pytest.mark.sanity
#@pytest.mark.skip("Skipping this test case as this functionality is not working developer will fix it in new
built")
def test_helloWorld():
    print("This is Sanity Test")
    print("*********************Hello PyTest************************")

@pytest.mark.smoke
# a=101
# @pytest.mark.skipif(a>30,reason="Skipping this test case as this functionality is not working developer will
fix it in new built")
def test_condition():
    print("This is Smoke Test")
    print("*********************Executing if greater than 30************************")
```

Executing Test Cases with Single Tags: Use the -m option followed by the tag name to execute test cases with a specific tag.

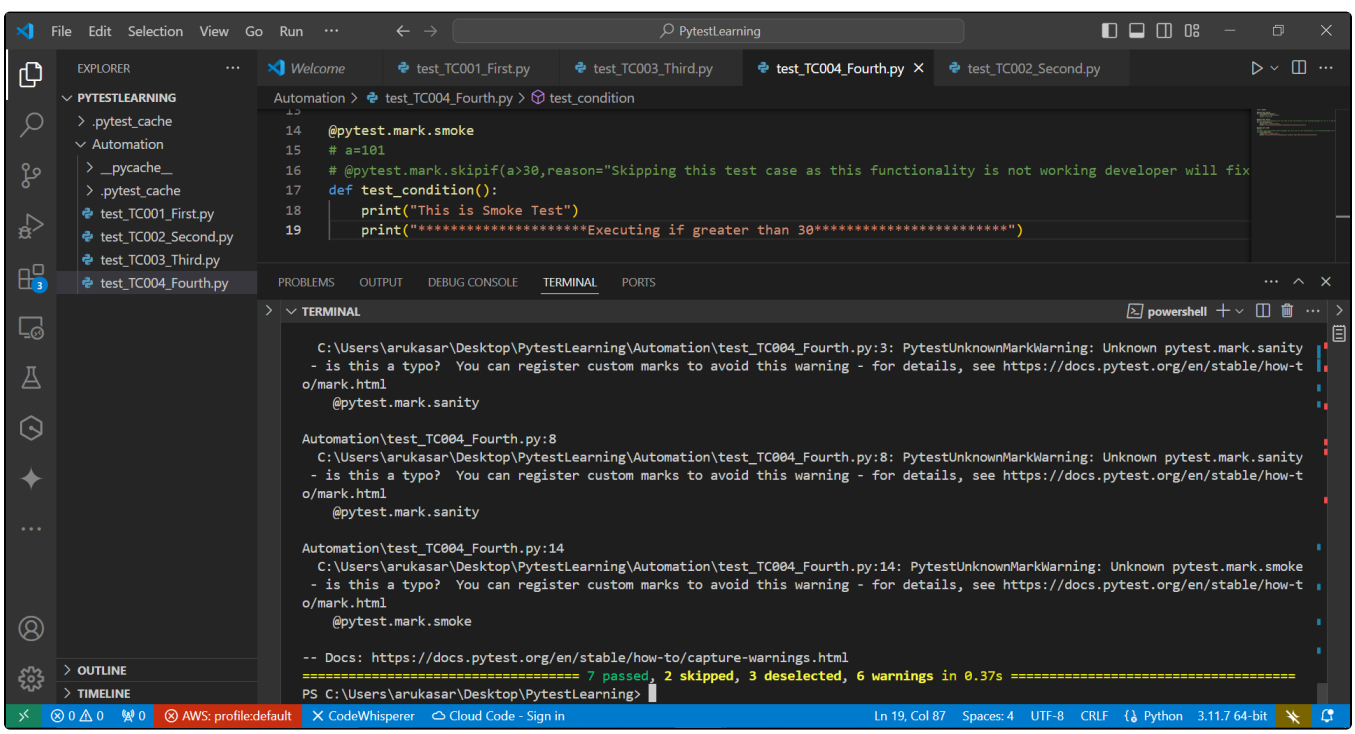
tag/grouping command

```
pytest -m smoke <folder_name>
```



Skipping Execution of Test Cases by Tag Name: Skip executing test cases with a specific tag by using the -m option with a tilde (~) followed by the tag name.

tag/grouping command
`pytest -m "not smoke" <folder_name>`



Assigning Multiple Tags to Test Cases: You can assign multiple tags to a single test case by chaining multiple markers.

test_TC003_Third.py

```
import pytest

@pytest.mark.smoke
@pytest.mark.regression
def test_addition():
    print("This is Smoke Test")
    print("This is Regression Test")
    assert 2 + 2 == 4

@pytest.mark.sanity
@pytest.mark.regression
def test_substraction():
    print("This is Sanity Test")
    print("This is Regression Test")
    assert 5 - 3 == 2

@pytest.mark.smoke
def test_multiplication():
    print("This is Smoke Test")
    assert 5 - 3 == 2
```

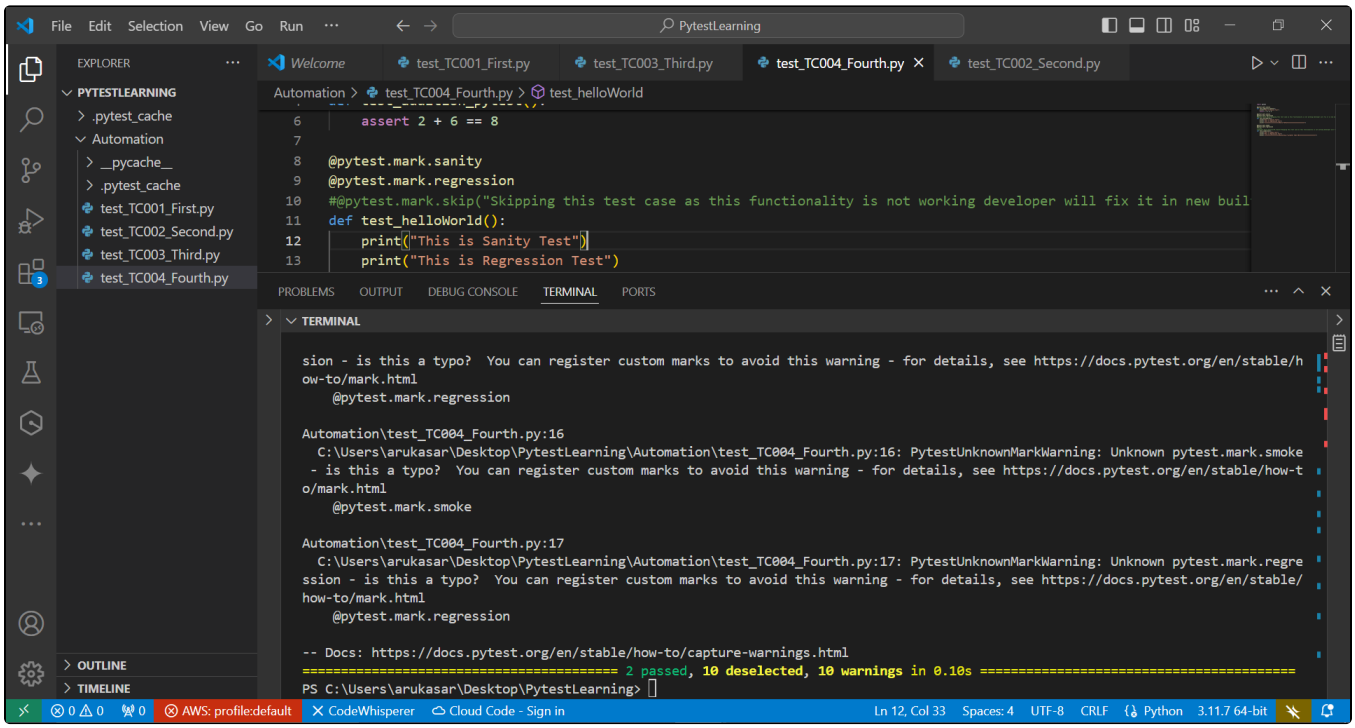
test_TC004_Fourth.py

```
import pytest

@pytest.mark.sanity
def test_addition_pytest():
    print("This is Sanity Test")
    assert 2 + 6 == 8

@pytest.mark.sanity
@pytest.mark.regression
#@pytest.mark.skip("Skipping this test case as this functionality is not working developer will fix it in new
built")
def test_helloWorld():
    print("This is Sanity Test")
    print("This is Regression Test")
    print("********************Hello PyTest************************")

@pytest.mark.smoke
@pytest.mark.regression
# a=101
# @pytest.mark.skipif(a>30,reason="Skipping this test case as this functionality is not working developer will
fix it in new built")
def test_condition():
    print("This is Smoke Test")
    print("This is Regression Test")
    print("********************Executing if greater than 30************************")
```

Executing Test Cases Using Multiple Tags (AND Logic): To execute test cases using multiple tags (AND logic), combine the tag names with `and` within quotes.

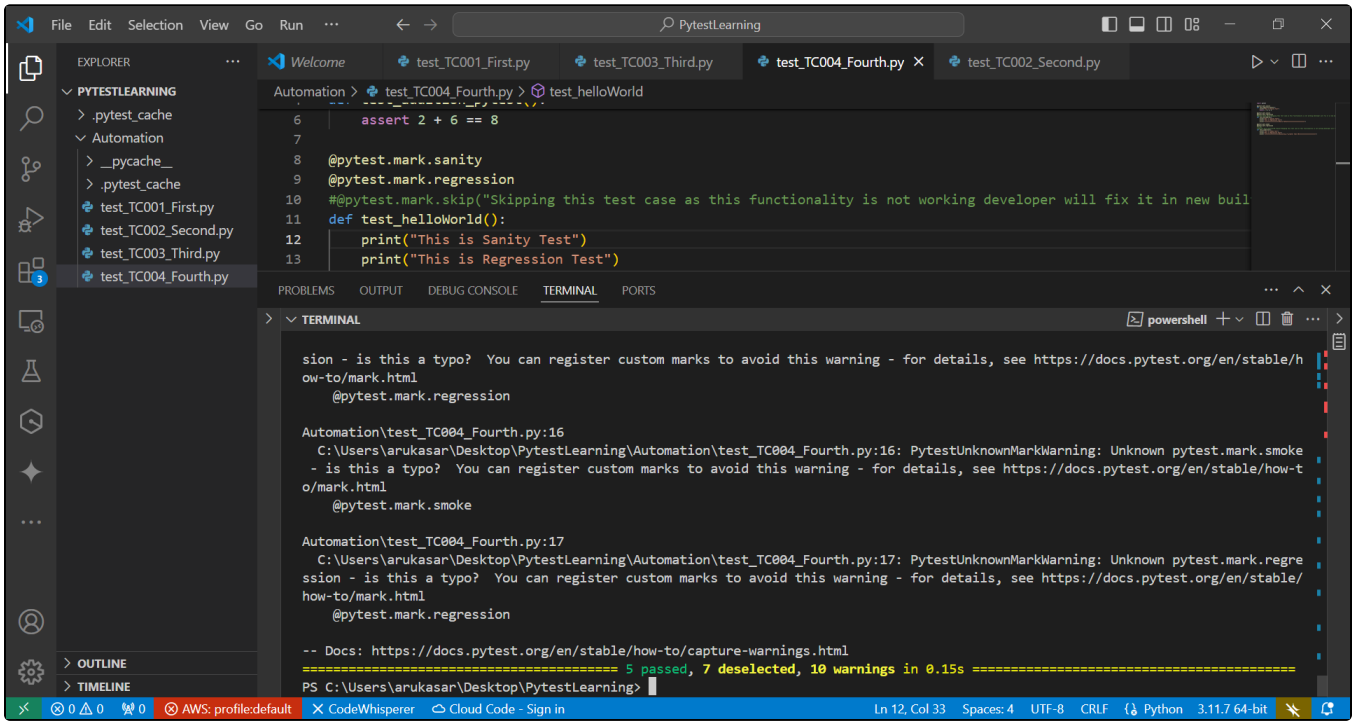
tag/grouping command

```
pytest -m "smoke and regression" <folder_name>
```



OR: You can also use `or` within quotes for executing test cases that have either of the specified tags.

tag/grouping command
`pytest -m "smoke and regression" <folder_name>`



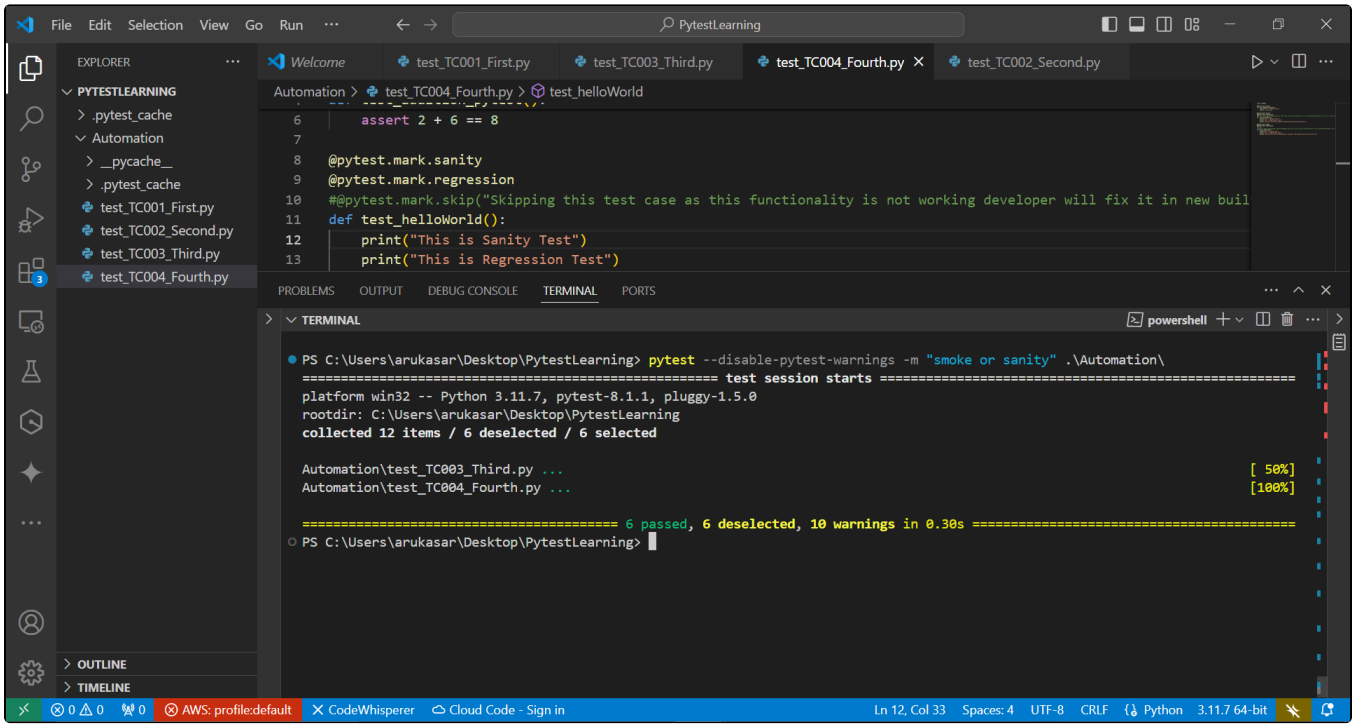
Handle Custom Tagging issues

> *Approach 1: --disable-pytest-warnings*
>
> *Approach 2: register the custom marker*

Approach 1:

tag/grouping command

```
pytest --disable-pytest-warnings -m "smoke or sanity" <folder_name>
```

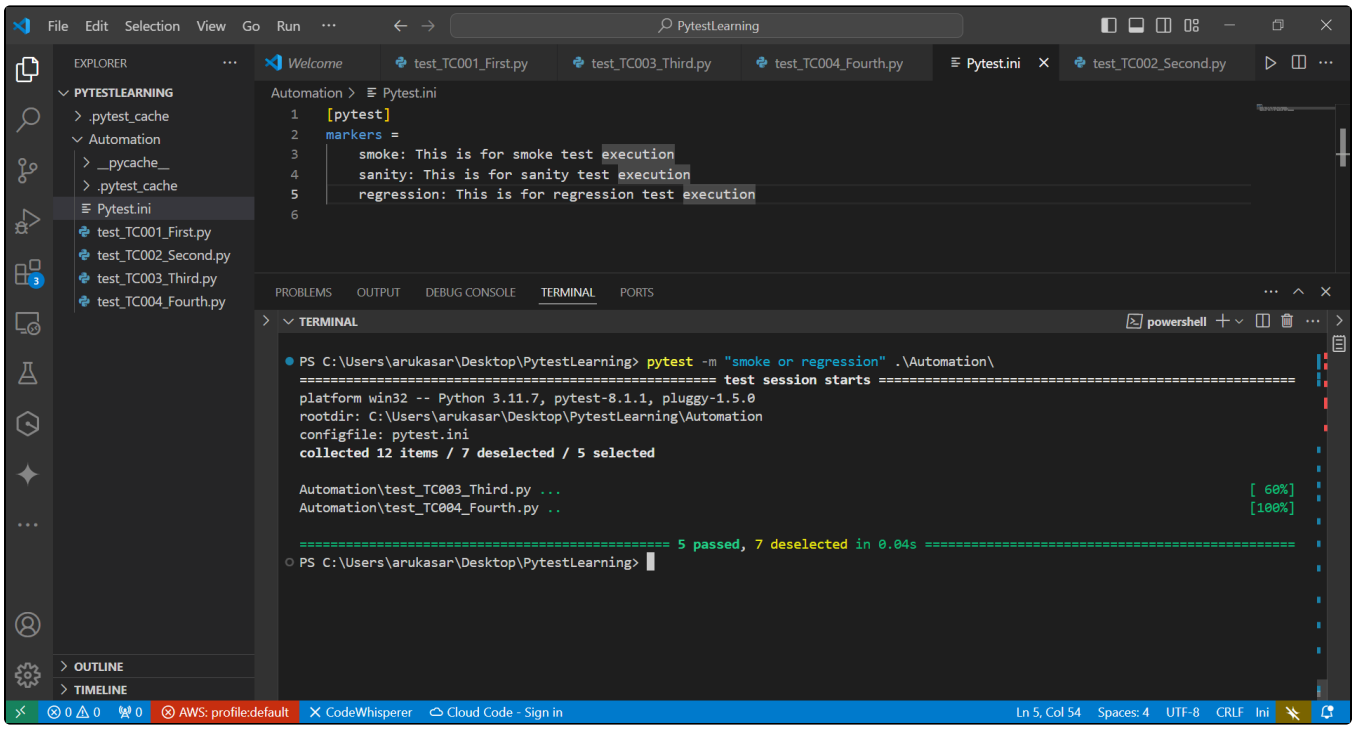


Approach 2:

1. Create a new file named pytest.ini in your project directory.
2. Open the pytest.ini file using a text editor.
3. Add the following content to the file:
4. Save the changes.

Pytest.ini

```
[pytest]
markers =
    smoke: This is for smoke test execution
    sanity: This is for sanity test execution
    regression: This is for regression test execution
```



Assertions

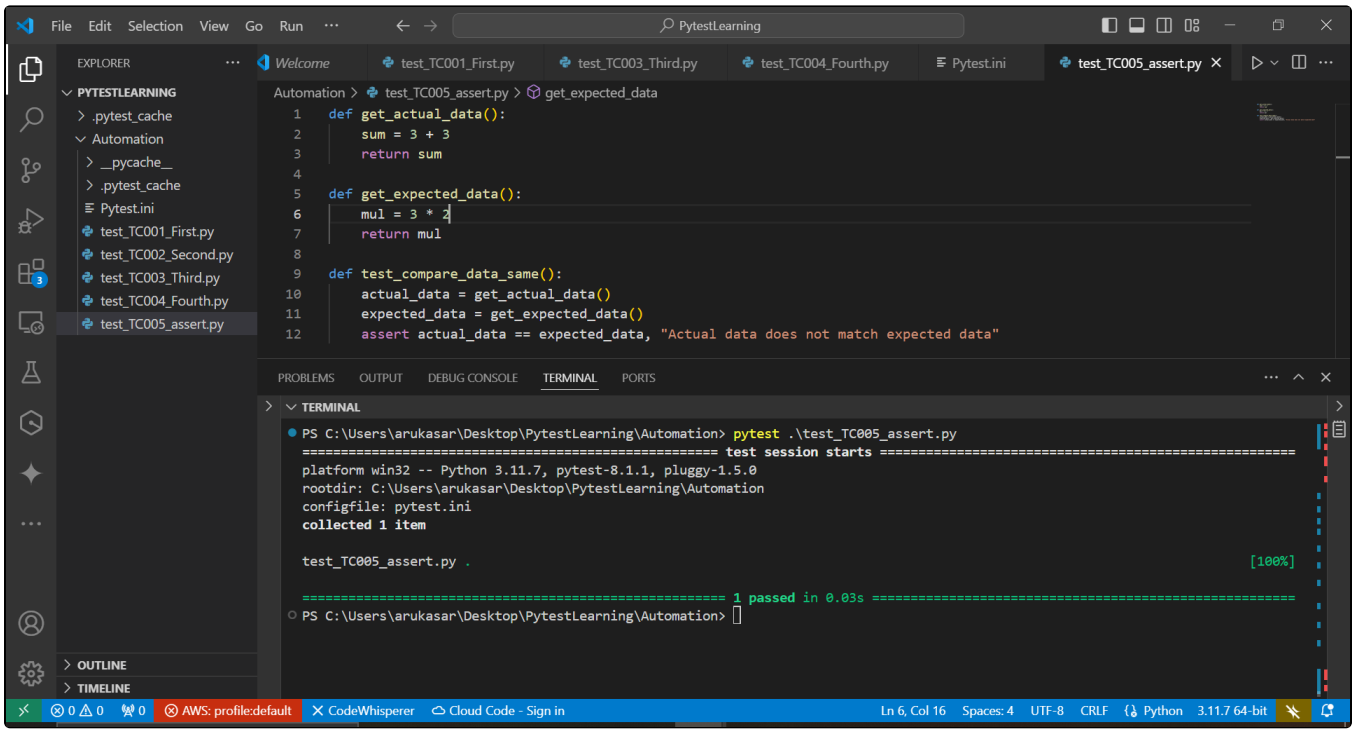
> *Compare actual results with expected results*

- Compare data to be the same
- Compare data NOT to be the same
- Compare data and display customized messages in case of failure

Compare data to be the same:

test_TC005_assert.py

```
def get_actual_data():
    sum = 3 + 3
    return sum

def get_expected_data():
    mul = 3 * 2
    return mul

def test_compare_data_same():
    actual_data = get_actual_data()
    expected_data = get_expected_data()
    assert actual_data == expected_data, "Actual data does not match expected data"
```



Compare data NOT to be the same:

test_TC005_assert.py

```
def get_actual_data():
    sum = 3 + 3
    return sum

def get_unexpected_data():
    mul = 3 * 8
    return mul

def test_compare_data_not_same():
    actual_data = get_actual_data()
    unexpected_data = get_unexpected_data()
    assert actual_data != unexpected_data, "Actual data matches unexpected data"
```

Failure Result:

```
def get_actual_data():
    sum = 3 + 3
    return sum

def get_unexpected_data():
    mul = 3 * 2
    return mul

def test_compare_data_not_same():
    actual_data = get_actual_data()
    unexpected_data = get_unexpected_data()
    assert actual_data != unexpected_data, "Actual data matches unexpected data"
```

```
PS C:\Users\arukasar\Desktop\PytestLearning\Automation>
========================================================== FAILURES ==========================================================
_____________________________________________ test_compare_data_not_same _____________________________________________

    def test_compare_data_not_same():
        actual_data = get_actual_data()
        unexpected_data = get_unexpected_data()
>       assert actual_data != unexpected_data, "Actual data matches unexpected data"
E       AssertionError: Actual data matches unexpected data
E       assert 6 != 6

test_TC005_assert.py:16: AssertionError
==================================================== short test summary info =====================================================
FAILED test_TC005_assert.py::test_compare_data_not_same - AssertionError: Actual data matches unexpected data
====================================================== 1 failed in 0.58s =========================================================
PS C:\Users\arukasar\Desktop\PytestLearning\Automation>
```

Positive Result:

```
def get_actual_data():
    sum = 3 + 3
    return sum

def get_unexpected_data():
    mul = 3 * 8
    return mul

def test_compare_data_not_same():
    actual_data = get_actual_data()
    unexpected_data = get_unexpected_data()
    assert actual_data != unexpected_data, "Actual data matches unexpected data"
```

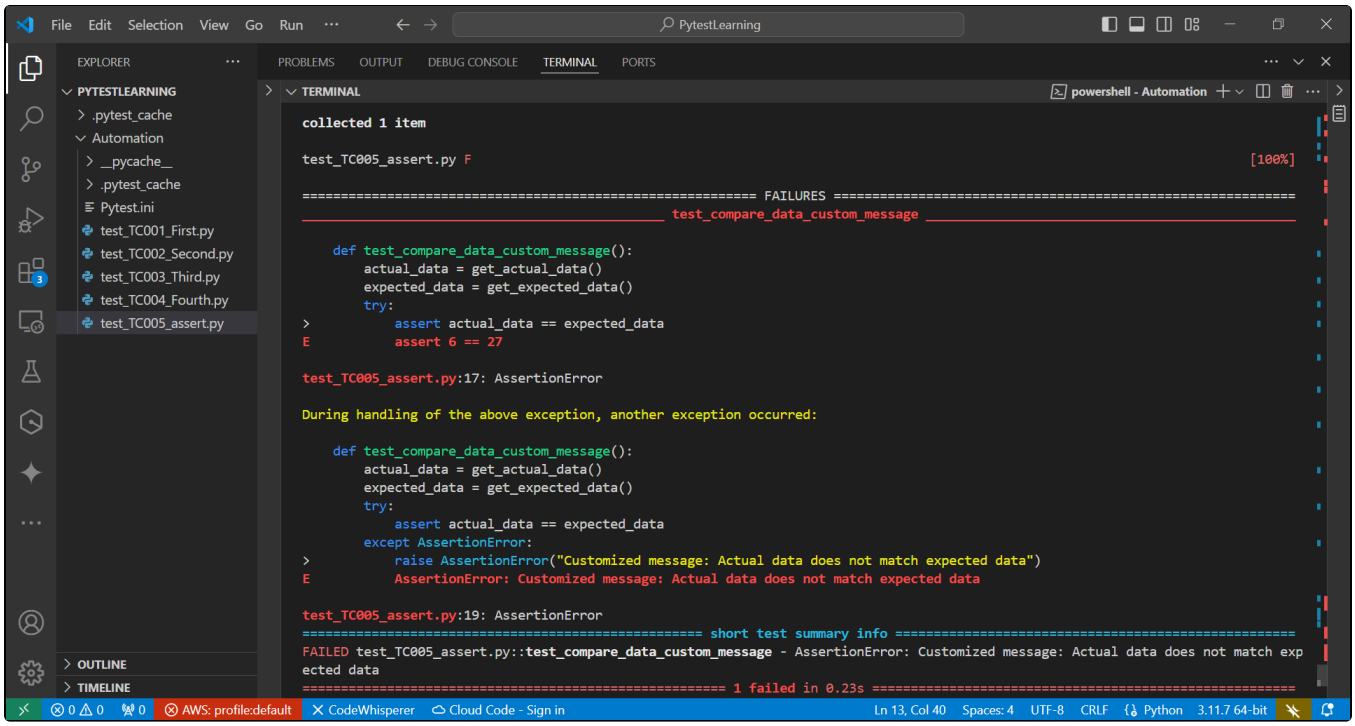
```
PS C:\Users\arukasar\Desktop\PytestLearning\Automation> pytest .\test_TC005_assert.py
========================================================= test session starts ========================================================
platform win32 -- Python 3.11.7, pytest-8.1.1, pluggy-1.5.0
rootdir: C:\Users\arukasar\Desktop\PytestLearning\Automation
configfile: pytest.ini
collected 1 item

test_TC005_assert.py .                                                                                                      [100%]

========================================================= 1 passed in 0.16s ==========================================================
PS C:\Users\arukasar\Desktop\PytestLearning\Automation>
```

Compare data and display a customized message in case of failure:

test_TC005_assert.py

```
def get_expected_data():
    mul = 3 * 9
    return mul

def get_actual_data():
    sum = 3 + 3
    return sum

def test_compare_data_custom_message():
    actual_data = get_actual_data()
    expected_data = get_expected_data()
    try:
        assert actual_data == expected_data
    except AssertionError:
        raise AssertionError("Customized message: Actual data does not match expected data")
```

Failure Result:



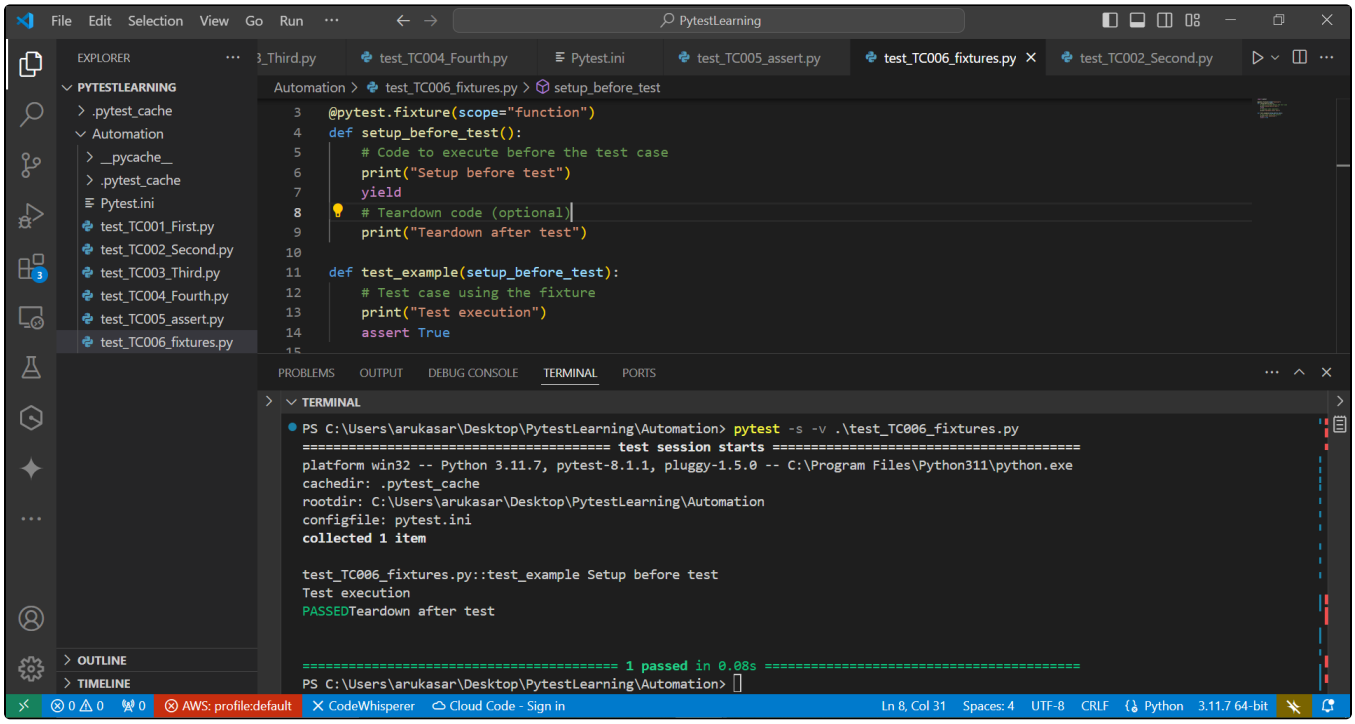
PyTest Fixtures

1. Execute something before the test case
2. Execute after test cases
3. Execute only one

Execute something before the test case:

test_TC006_fixtures.py

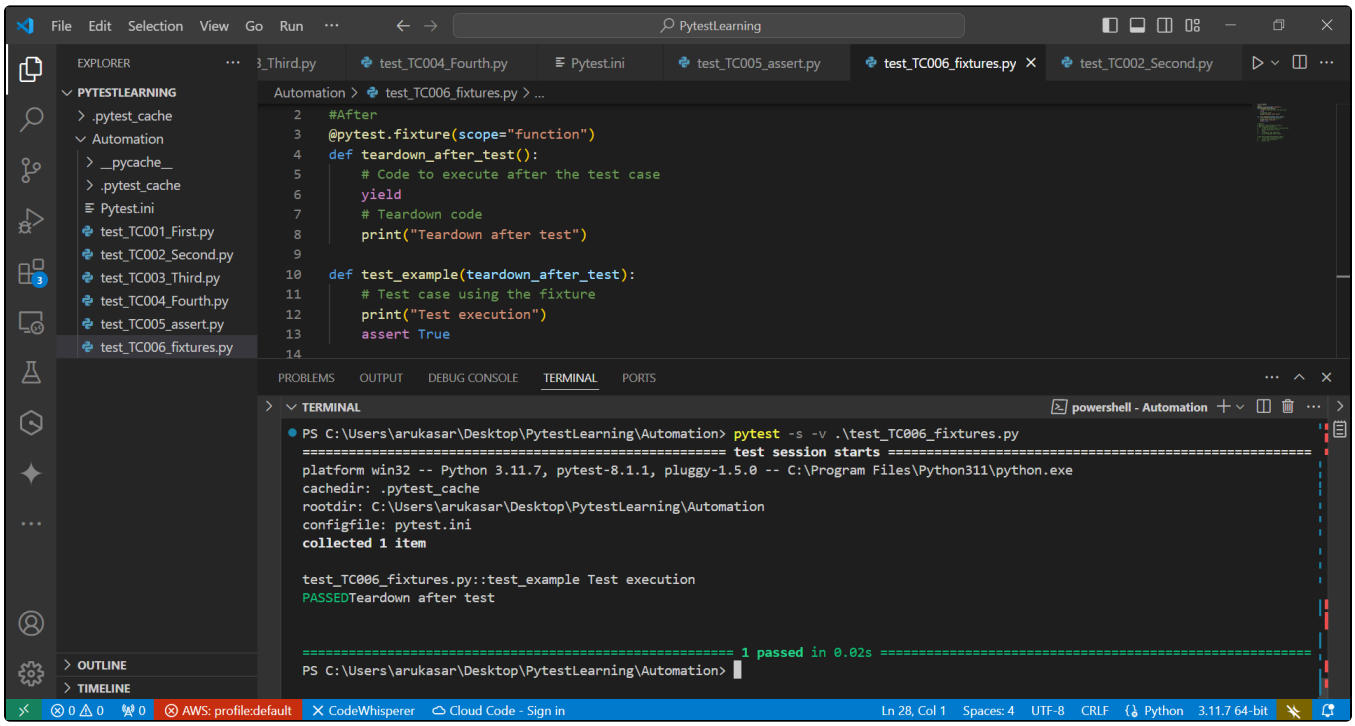
```
import pytest

@pytest.fixture(scope="function")
def setup_before_test():
    # Code to execute before the test case
    print("Setup before test")
    yield
    # Teardown code (optional)
    print("Teardown after test")

def test_example(setup_before_test):
    # Test case using the fixture
    print("Test execution")
    assert True
```



Execute after test cases:

test_TC006_fixtures.py

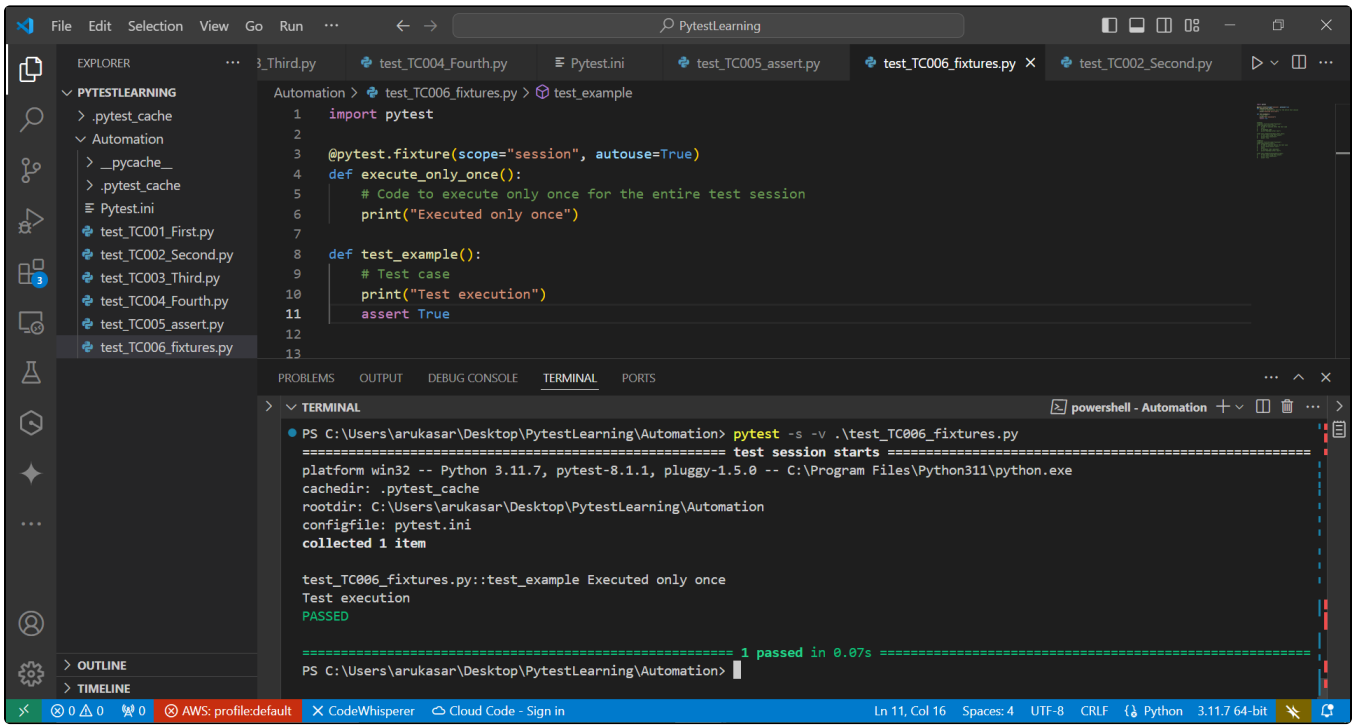
```
import pytest

@pytest.fixture(scope="function")
def teardown_after_test():
    # Code to execute after the test case
    yield
    # Teardown code
    print("Teardown after test")

def test_example(teardown_after_test):
    # Test case using the fixture
    print("Test execution")
    assert True
```



Execute only one:

test_TC006_fixtures.py

```
import pytest

@pytest.fixture(scope="session", autouse=True)
def execute_only_once():
    # Code to execute only once for the entire test session
    print("Executed only once")

def test_example():
    # Test case
    print("Test execution")
    assert True
```



Attachment for reference:

PytestLearning_Notes.zip