Testing in Python

Testing in Python

Testing is a very important and essential part in software development process. It ensures our software, applications, and websites work as expected.

Testing is a best practice that can save time and money by finding and fixing issues before they cause major problems.

Unit tests are segments of code written to test other pieces of code, typically a single function or method, that we refer to as a unit.

Python has two main frameworks to make unit testing easier: **unittest** and **PyTest**.

The Assert Statement/1

The assert statement is a built-in statement in Python used to assert if a given condition is true or not.

If the condition is true, nothing happens, but if it's not true, an error is raised.

Although, at first, it may look like the try and except clauses, they are completely different, and assert should not be used for error handling but for debugging and testing reasons.

- As an example, the condition in the line below is true and, therefore, it does not output or return anything:

```
assert 1 > 0
```

The Assert Statement/2

- However, if we change this condition so it becomes false, we get an AssertionError:

assert 1 < 0

>>>
AssertionError
Traceback (most recent call last)
in <module>
assert 1 < 0
AssertionError:

The Assert Statement/3

- Notice that in the last row of the error message there isn't an actual message after AssertionError.
- That's because the user should pass this message. Here's how:

```
n = 0
assert 1 < n, 'The Condition is False’
```

```
>>>
 AssertionError
Traceback (most recent call last)
in <module>
 assert 1 < n, 'The Condition is False'
 AssertionError: The Condition is False
```

The Assert Statement Syntax

So, the basic syntax for **assert** statement is the following:

assert <condition being tested>, <error message to be displayed>

Assert is very simple to use.

Understanding it is critical for testing purposes, as we'll see in the following sections.

Pytest

The Pytest framework makes it easy to write small, readable tests, and can scale to support complex functional testing for applications and libraries.

To **install Pytest**,

1. Run the following command in your command line:

```
pip install -U pytest
```

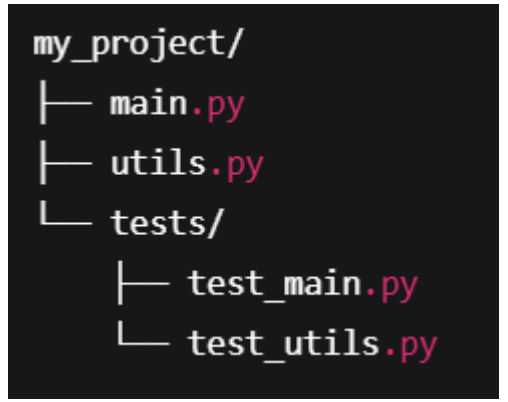
2. Check that you installed the correct version:

```
$ pytest --version
pytest 8.3.5
```

Test Conventions

Before diving into writing tests, it's important to review some test conventions that Pytest is based on.
Usually, for our projects, it is preferable to use a similar folder structure.

For tests in our project, it is recommended to:



- Use a **tests** directory to store test files and any nested test directories.

- Add a **test** prefix to all test files. This prefix indicates that the file contains test code.

Note.

Avoid using **test** as the name of the directory.

The name **test** is a Python module, so creating a directory with the same name may cause conflicts or replacement.
Always **use** the plural form **tests** instead.

Pytest – Your First Test/1

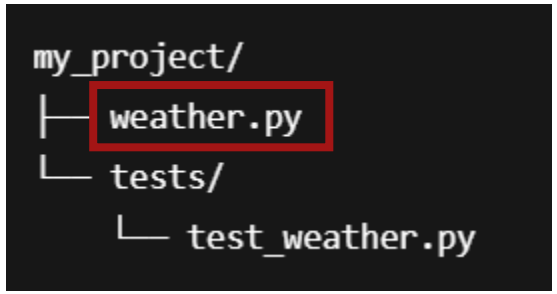
Let’s consider the following project structure.

```
my_project/
├── weather.py
└── tests/
    └── test_weather.py
```

Pytest – Your First Test/2

weather.py file:

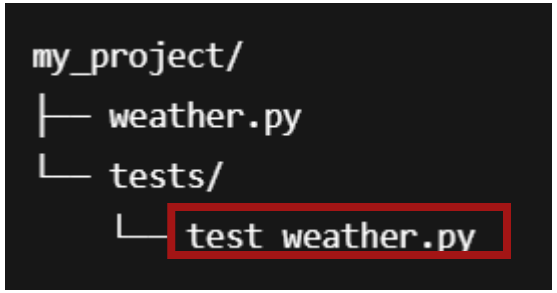
```
def check_weather(temperature: float) -> str:
    if temperature > 20:
        return "hot"
    elif 10 < temperature <= 20:
        return "average"
    else:
        return "cold"
```



Pytest – Your First Test/3

test_weather.py file:

```
from my_project.weather import check_weather

# passed
def test_check_weather():
    assert check_weather(21.00) == "hot", 'temperatures greater than 20 degree \
        must be considered as hot'
```



Pytest – Your First Test/4

To run file test_weather.py :

```
pytest my_projects/tests/test_weather.py
```

Output:

```
collected 1 item

my_project\tests\test_weather.py .                                                                    [100%]

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

Pytest – Your First Test/5

Pytest found 1 test to run in the file test_weather.py

```
collected 1 item

my_project\tests\test_weather.py .

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

The green dot (.) following my_project\tests\test_weather.py means that the test passed successfully.
Each dot represents a passed test.

[100%] refers to the overall progress of running all test cases.

```
                                                                                [100%]
1 passed in 0.03s ============================================================================
```

All tests (1 out of 1) were executed successfully in 0.03 seconds.

Pytest – Your First Test/6

test_weather.py file:

```
from my_project.weather import check_weather

# failed
def test_check_weather():
    assert check_weather(5.00) == “average", 'temperatures between 10 and 20 degree \
        must be considered as average temperature'
```

Pytest – Your First Test/7

To run file test_weather.py :

```
pytest my_projects/tests/test_weather.py
```

Output:

```
collected 1 item

my_project\tests\test_weather.py F                                                                    [100%]

============================================ FAILURES ============================================
_______________________________________ test_check_weather _______________________________________

    def test_check_weather():
>       assert check_weather(5.00) == "average", 'temepratures between 10 and 20 degree must be considered as average temperature'
E       AssertionError: temepratures between 10 and 20 degree must be considered as average temperature
E       assert 'cold' == 'average'
E
E         - average
E         + cold

my_project\tests\test_weather.py:10: AssertionError
===================================== short test summary info =====================================
FAILED my_project/tests/test_weather.py::test_check_weather - AssertionError: temepratures between 10 and 20 degree must be considered as average temperature
========================================= 1 failed in 0.30s =========================================
```

INFORMATION AND COMMUNICATIONS TECHNOLOGY

Pytest – Your First Test/8

```
collected 1 item

my_project\tests\test_weather.py F

=====================================
```

The red F (**F**) following my_project\tests\test_weather.py means that the test failed.

The error section (failures) shows the name of the test that failed: test_check_weather

```
collected 1 item

my_project\tests\test_weather.py F                                                                   [100%]

================================================ FAILURES ================================================
__________________________________________ test_check_weather ____________________________________________

    def test_check_weather():
>       assert check_weather(5.00) == "average", 'temepratures between 10 and 20 degree must be considered as average temperature'
E       AssertionError: temepratures between 10 and 20 degree must be considered as average temperature
E       assert 'cold' == 'average'
```

ICTAcademy
INFORMATION AND COMMUNICATIONS TECHNOLOGY

Pytest – Your First Test/9

```
collected 1 item

my_project\tests\test_weather.py F                                                                                    [100%]

================================================================================ FAILURES ================================================================================
_____________________________________________________________________ test_check_weather _____________________________________________________________________

    def test_check_weather():
>       assert check_weather(5.00) == "average", 'temepratures between 10 and 20 degree must be considered as average temperature'
E       AssertionError: temepratures between 10 and 20 degree must be considered as average temperature
E       assert 'cold' == 'average'
E
E         - average
E         + cold

my_project\tests\test_weather.py:10: AssertionError
============================================================================ short test summary info =============================================================================
FAILED my_project/tests/test_weather.py::test_check_weather - AssertionError: temepratures between 10 and 20 degree must be considered as average temperature
=============================================================================== 1 failed in 0.30s ================================================================================
```

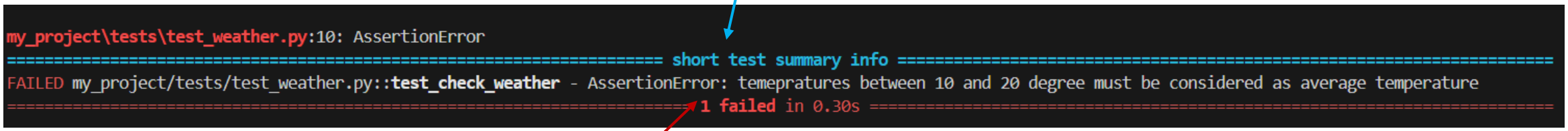
(-): Expected Value

(+): Actual Value

AssertionError

Pytest – Your First Test/10

Short test Summary. This is a compact version of the failure report:
it tells you exactly which test failed and what kind of error occurred (AssertionError)



Test Conclusion:
Pytest ran 1 test, the test failed and the entire execution took 0.26 seconds.

Pytest – Your First Test/11

test_weather.py file:

```python
from my_project.weather import check_weather

# passed
def test_check_weather1():
    assert check_weather(21.00) == "hot", 'temperatures greater than 20 degree must be considered as hot'

# failed
def test_check_weather2():
    assert check_weather(5.00) == "average", 'temepratures between 10 and 20 degree must be considered as average temperature'

# passed
def test_check_weather3():
    assert check_weather(5.00) == "cold", 'temperatures lower than 10 degree must be considered as cold'

# passed
def test_check_weather4():
    assert check_weather(13.00) == 'average', 'temepratures between 10 and 20 degree must be considered as average temperature'
```

Pytest – Your First Test/12

Output:

```
collected 4 items

my_project\tests\test_weather.py .F..                                    [100%]

================================== FAILURES ===================================
_____________________________ test_check_weather2 _____________________________

    def test_check_weather2():
>       assert check_weather(5.00) == "average", 'temepratures between 10 and 20 degree must be considered as average temperature'
E       AssertionError: temepratures between 10 and 20 degree must be considered as average temperature
E       assert 'cold' == 'average'
E
E         - average
E         + cold

my_project\tests\test_weather.py:10: AssertionError
=========================== short test summary info ===========================
FAILED my_project/tests/test_weather.py::test_check_weather2 - AssertionError: temepratures between 10 and 20 degree must be considered as average temperature
========================= 1 failed, 3 passed in 0.32s =========================
```

Each test_check_weather function defined is considered by Pytest as a single test to execute.
In fact, the output shows 4 tests executed: 1 failed and 3 passed.

ICTAcademy
INFORMATION AND COMMUNICATIONS TECHNOLOGY

Pytest – Your First Test/13

All the methods' names must start with the word **test**.

This is a convention we use so that **pytest** can identify the tests it's supposed to run.

For instance, the following code runs only three tests:

```python
from my_project.weather import check_weather

# passed
def not_a_test_check_weather1():
    assert check_weather(21.00) == "hot", 'temperatures greater than 20 degree must be considered as hot'

# failed
def test_check_weather2():
    assert check_weather(5.00) == "average", 'temepratures between 10 and 20 degree must be considered as average temperature'

# passed
def test_check_weather3():
    assert check_weather(5.00) == "cold", 'temperatures lower than 10 degree must be considered as cold'

# passed
def test_check_weather4():
    assert check_weather(13.00) == 'average', 'temepratures between 10 and 20 degree must be considered as average temperature'
```

Pytest – Your First Test/14

Output:

```
collected 3 items

my_project\tests\test_weather.py F..                                                                                [100%]

================================================================================ FAILURES ================================================================================
_____________________________________________________________________ test_check_weather2 _____________________________________________________________________

    def test_check_weather2():
>       assert check_weather(5.00) == "average", 'temepratures between 10 and 20 degree must be considered as average temperature'
E       AssertionError: temepratures between 10 and 20 degree must be considered as average temperature
E       assert 'cold' == 'average'
E
E         - average
E         + cold

my_project\tests\test_weather.py:9: AssertionError
============================================================================ short test summary info =============================================================================
FAILED my_project/tests/test_weather.py::test_check_weather2 - AssertionError: temepratures between 10 and 20 degree must be considered as average temperature
========================================================================== 1 failed, 2 passed in 0.24s ===========================================================================
```

Since one of the four functions does not have a name starting with the **test prefix**, **only 3 tests are executed** instead of 4, of which **1 failed and 2 passed**.

Pytest – Your First Test/15

test_weather.py

```python
from my_project.weather import check_weather

# passed
def test_check_weather1():
    assert check_weather(21.00) == "hot", 'temperatures greater than 20 degree must be considered as hot'

# failed
def test_check_weather2():
    assert check_weather(5.00) == "average", 'temepratures between 10 and 20 degree must be considered as average temperature'

# passed
def test_check_weather3():
    assert check_weather(5.00) == "cold", 'temperatures lower than 10 degree must be considered as cold'

# passed
def test_check_weather4():
    assert check_weather(13.00) == 'average', 'temepratures between 10 and 20 degree must be considered as average temperature'

# failed because every def test_function() is considered as a single test
def test_check_weather5():
    assert check_weather(30.00) == 'hot', 'temperatures greater than 20 degree must be considered as hot'
    assert check_weather(11.00) == 'cold', 'temperatures lower than 10 degree must be considered as cold'
```

Looking at test_check_weather 5, we expect in output 6 tests executed: 4 passed and 2 failed.

Pytest – Your First Test/16

Output shows 5 tests executed, with **2 failed and 3 passed**.
This happens because even though the function test_check_weather5 contains two **assert** statements, it is actually executed as a single test. Therefore, **each function** defined in the test file **is executed as a single test**.

```
collected 5 items

my_project\tests\test_weather.py .F..F                                   [100%]

================================== FAILURES ===================================
_____________________________ test_check_weather2 _____________________________

    def test_check_weather2():
>       assert check_weather(5.00) == "average", 'temepratures between 10 and 20 degree must be considered as average temperature'
E       AssertionError: temepratures between 10 and 20 degree must be considered as average temperature
E       assert 'cold' == 'average'
E
E         - average
E         + cold

my_project\tests\test_weather.py:9: AssertionError
_____________________________ test_check_weather5 _____________________________

    def test_check_weather5():
        assert check_weather(30.00) == 'hot', 'temperatures greater than 20 degree must be considered as hot'
>       assert check_weather(11.00) == 'cold', 'temperatures lower than 10 degree must be considered as cold'
E       AssertionError: temperatures lower than 10 degree must be considered as cold
E       assert 'average' == 'cold'
E
E         - cold
E         + average

my_project\tests\test_weather.py:22: AssertionError
=========================== short test summary info ===========================
FAILED my_project/tests/test_weather.py::test_check_weather2 - AssertionError: temepratures between 10 and 20 degree must be considered as average temperature
FAILED my_project/tests/test_weather.py::test_check_weather5 - Ass...than 10 degree must be considered as cold
========================= 2 failed, 3 passed in 0.26s =========================
```

Pytest – multiple tests/1

How can we handle tests with multiple specific input and output combinations?

When you want to test a function with many input and output combinations without writing multiple individual test functions, **@pytest.mark.parametrize** becomes especially useful.

@pytest.mark.parametrize is a decorator provided by Pytest that allows you to run the same test multiple times with different data.

Why use it?

- Reduces code repetition
- Makes tests more readable and maintainable
- Makes it easy to identify which specific cases pass or fail

Note.

To use **@pytest.mark.parametrize**, pytest module must be imported.

Pytest – multiple tests/2

What does **@pytest.mark.parametrize** do?

It runs the test_check_weather function **multiple times**, each **time passing in a pair of values** from the given list.

These values are '**temperature**' (that is the input temperature value) and '**expected**' (that is he expected output value).

test_weather.py file:

```
from my_project.weather import check_weather
import pytest


@pytest.mark.parametrize("temperature, expected", [
    (21.00, "hot"),
    (13.00, "average"),
    (0.00, "cold"),
    (15.00, "cold")
])
def test_check_weather(temperature, expected):
    assert check_weather(temperature) == expected
```

Pytest – multiple tests/3

@pytest.mark.parametrize runs the test_check_weather function **4 times**:

- The **first time** the statement **assert check_weather(21.00) == "hot“** is executed.
- The **second time** the statement **assert check_weather(13.00) == "average“** is executed.
- The **third time** the statement **assert check_weather(00.00) == "cold“** is executed.
- The **fourth time** the statement **assert check_weather(15.00) == "cold“** is executed.

test_weather.py file:

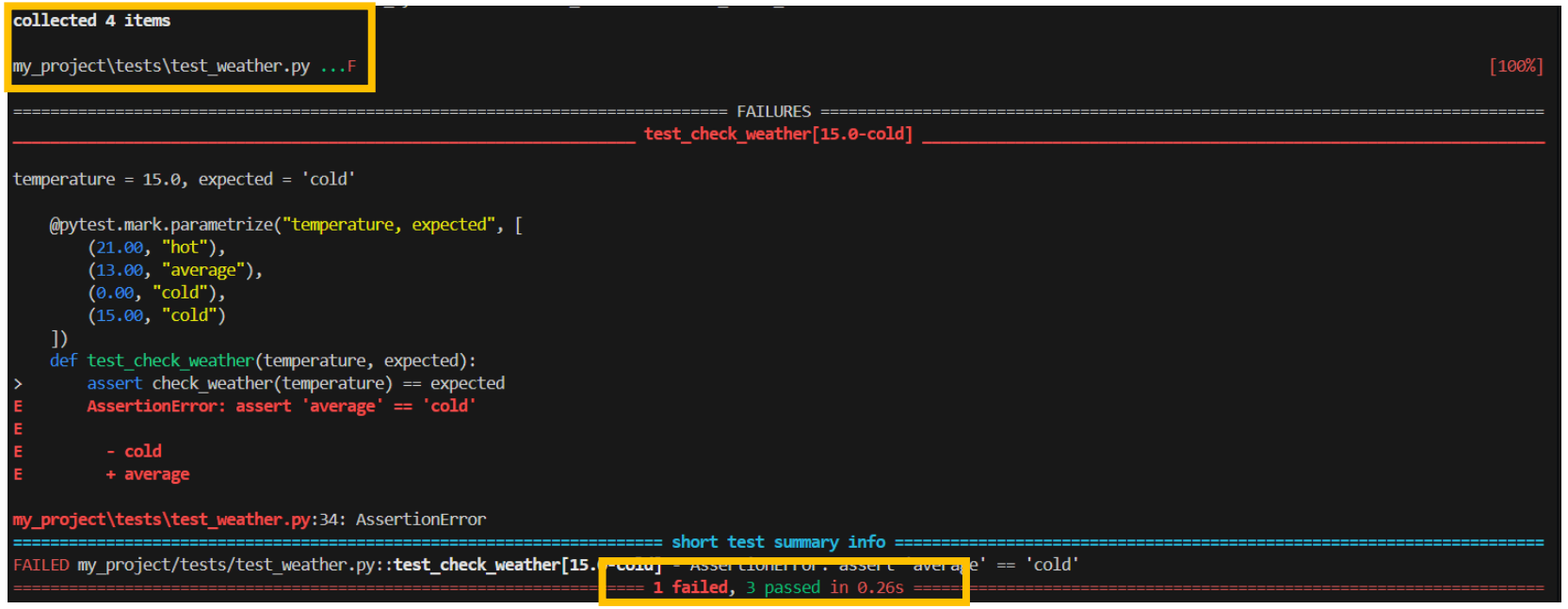
```
from my_project.weather import check_weather
import pytest


@pytest.mark.parametrize("temperature, expected", [
    (21.00, "hot"),
    (13.00, "average"),
    (0.00, "cold"),
    (15.00, "cold")
])
def test_check_weather(temperature, expected):
    assert check_weather(temperature) == expected
```

Pytest – multiple tests/4

4 tests are executed by Pytest: **3 passed and 1 failed**, because the fourth asserted condition is false.

Thanks to @pytest.mark.parametrize decorator, testing the check_weather() function becomes more dynamic and less repetitive.

Output:



ICTAcademy
INFORMATION AND COMMUNICATIONS TECHNOLOGY

Pytest – multiple tests/5

We can also assign a custom AssertionError if a test executed with **@pytest.mark.parametrize** fails.

This helps better understand the context of the failure during debugging.

```python
@pytest.mark.parametrize("temperature, expected", [
    (21.00, "hot"),
    (13.00, "average"),
    (0.00, "cold"),
    (15.00, "cold")
])
def test_check_weather(temperature, expected):
    ae: str = ""
    if temperature > 20:
        ae = 'temperatures greater than 20 degree must be considered as hot'
    elif 10 < temperature <= 20:
        ae = 'temperatures within 10 and 20 degree must be considered as average temperature'
    else:
        ae = 'temperatures lower than 10 degree must be considered as cold'
    assert check_weather(temperature) == expected, ae
```

Pytest – fixture/1

Let’s write a calculator class in python.

```python
class Calculator:

    def __init__(self, a: int, b: int):
        self.a = a
        self.b = b

    def addition(self) -> int:
        return int(self.a + self.b)

    def subtraction(self) -> int:
        return int(self.a - self.b)

    def multiplication(self) -> int:
        return int(self.a * self.b)

    def division(self) -> float | None:
        if self.b == 0:
            raise ValueError("Cannot divide by zero")
        return round(self.a / self.b, 2)
```

```
my_project/
├── weather.py
├── calculator.py
└── tests/
    ├── test_weather.py
    └── test_calculator.py
```

Pytest – fixture/2

Now let's write a series of test for the class!

```python
from my_project.calculator import Calculator

def test_addition():
    calculation: Calculator = Calculator(10,5)
    assert calculation.addition() == 13, 'The sum is wrong'

def test_subtraction():
    calculation: Calculator = Calculator(10,5)
    assert calculation.subtraction() == 5, 'The subtraction is wrong'

def test_multiplication():
    calculation: Calculator = Calculator(10,5)
    assert calculation.multiplication() == 50, 'The multiplication is wrong'

def test_division():
    calculation: Calculator = Calculator(10,5)
    assert calculation.division() == 2.00, 'The quotient is wrong'
```

```
my_project/
├── weather.py
├── calculator.py
└── tests/
    ├── test_weather.py
    └── test_calculator.py
```

Pytest – fixture/3

To run file test_calculator.py :

```
pytest my_project/tests/test_calculator.py
```

Output:

```
collected 4 items

my_project\tests\test_calculator.py F...                                 [100%]

================================== FAILURES ===================================
_____________________________ test_addition ______________________________

    def test_addition():
        calculation: Calculator = Calculator(10,5)
>       assert calculation.addition() == 13, 'The sum is wrong'
E       AssertionError: The sum is wrong
E       assert 15 == 13
E        +  where 15 = addition()
E        +    where addition = <my_project.calculator.Calculator object at 0x000001B785D51250>.addition

my_project\tests\test_calculator.py:5: AssertionError
=========================== short test summary info ===========================
FAILED my_project/tests/test_calculator.py::test_addition - AssertionError: The sum is wrong
========================= 1 failed, 3 passed in 0.36s =========================
```

4 tests executed: 1 failed and 3 passed.
The test_addiction failed.

Pytest – fixture/4

Let's optimize our code a bit. You probably have noticed that inside each test we initialized an object of the Calculations class, which will be tested. We can avoid that by using **@pytest.fixture** decorator.

What is @pytest.fixture ?

@pytest.fixture is a decorator that allows you **to create reusable objects** or resources **to be used in your tests**.
Instead of creating an object of the Calculator class in every test, this object is created once beforehand and then reused across multiple tests.

What is it for?

It is used to automatically prepare data, objects, connections, files, environments, etc., before a test runs — helping you avoid code repetition.

Use @pytest.fixture when:

- You need an object (e.g., a class instance, a dictionary, a file) in multiple tests
- You want to separate configuration/setup from the test logic
- You want to write cleaner, more maintainable tests

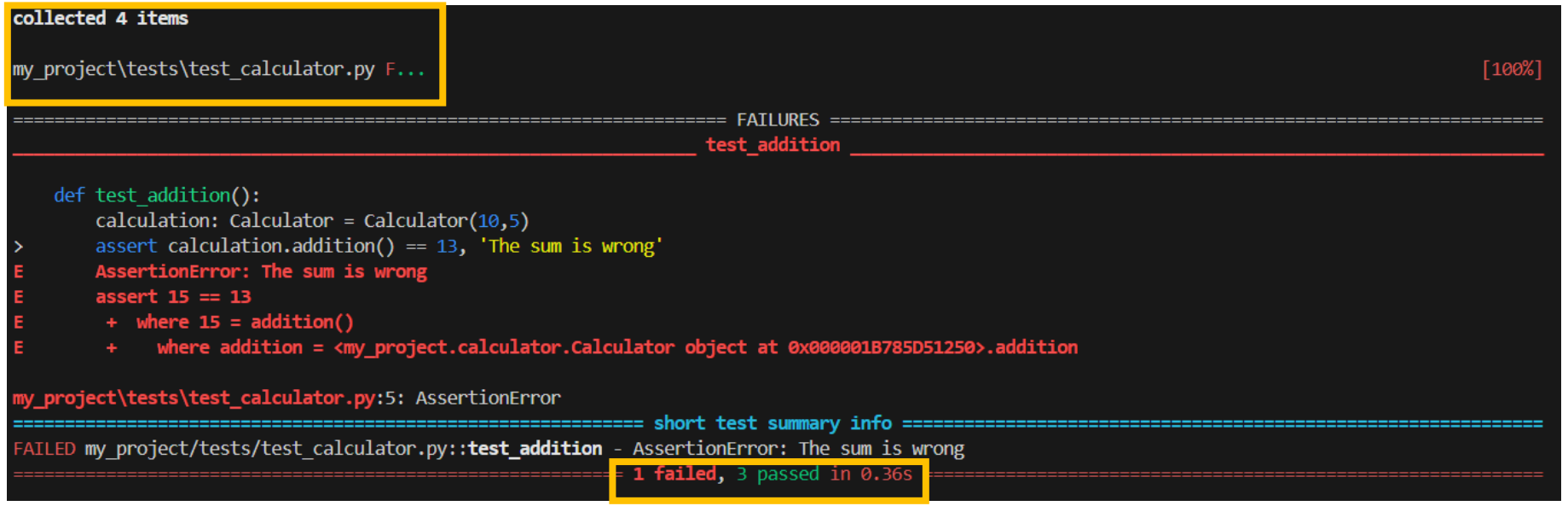
Pytest – fixture/5

Let's optimize our code using **@pytest.fixture**!

```python
from my_project.calculator import Calculator
import pytest
@pytest.fixture
def calculation():
    # creates a fresh istance of Calculator before each test
    return Calculator(10,5)

def test_addition(calculation):
    assert calculation.addition() == 13, 'The sum is wrong'

def test_subtraction(calculation):
    assert calculation.subtraction() == 5, 'The subtraction is wrong'

def test_multiplication(calculation):
    assert calculation.multiplication() == 50, 'The multiplication is wrong'

def test_division(calculation):
    assert calculation.division() == 2.00, 'The quotient is wrong'
```

Pytest – fixture/6

Output:



4 tests executed: 1 failed and 3 passed.
The test_addiction failed.

ICTAcademy
INFORMATION AND COMMUNICATIONS TECHNOLOGY

Pytest

To run all the tests in a directory we must use the command **pytest**:

This command:

- Automatically searches for all files that start with test or end with _test.py
- Executes all functions that start with test_
- Supports tests written with pytest

Pytest

The **pytest –v** command is used to run tests in "**verbose**" mode with the Pytest framework.

That is:

- It displays the full list of executed tests
- It shows the parameters as well, if you're using @pytest.mark.parametrize
- For each test, it shows:
  - the name of the file
  - the name of the test function
  - The result: FAILED, PASSED, etc.

```
my_project/tests/test_calculator.py::test_addition FAILED
my_project/tests/test_calculator.py::test_subtraction PASSED
my_project/tests/test_calculator.py::test_multiplication PASSED
my_project/tests/test_calculator.py::test_division PASSED
my_project/tests/test_weather.py::test_check_weather[21.0-hot] PASSED
my_project/tests/test_weather.py::test_check_weather[13.0-average] PASSED
my_project/tests/test_weather.py::test_check_weather[0.0-cold] PASSED
my_project/tests/test_weather.py::test_check_weather[15.0-cold] FAILED
```

Pytest

We can also specify a single test file inside a folder and run just that using the command

```
pytest -v my_project/tests/test_calculator.py
```

Output:

```
collected 4 items

my_project/tests/test_calculator.py::test_addition FAILED
my_project/tests/test_calculator.py::test_subtraction PASSED
my_project/tests/test_calculator.py::test_multiplication PASSED
my_project/tests/test_calculator.py::test_division PASSED
```

Pytest

Or we can do the same with a single test function inside a test file

```
pytest -v my_project/tests/test_calculator.py::test_addition
```

Output:

```
collected 1 item

my_project/tests/test_calculator.py::test_addition FAILED
```

Note.

In the **pytest** command, command, the double colons **::** are used to navigate inside a Python file and **specify** a particular **function to run**.