Overview

Now that you know how to write and run pytest tests, the next step is learning how to **design good tests** and how to **interpret failures correctly**.

When testing, it is very important to understand this rule:

A failing test does not always mean the test is wrong.

Sometimes the test is correct, and the function being tested is wrong. Other times the function is correct, and the test is written incorrectly. As the programmer, **you must know the expected result** before writing a test.

Thinking Like a Tester

Before writing a test, ask yourself:

- What is this function supposed to do?
- What result should I get for specific inputs?
- Can I verify that result without running the code?

Testing is about certainty, not guessing.

Creating Simple Math Functions

Let's start by creating a file called `math_functions.py`.

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

def subtract(a, b):
    return a - b
```

These functions are simple enough that we clearly know the expected results.

Writing Correct Tests

Create a test file called `test_math_functions.py`.

```
from math_functions import add, subtract

def test_add_basic():
    assert add(1, 4) == 5

def test_subtract_basic():
    assert subtract(10, 3) == 7
```

These tests use known inputs and correct outputs. These tests should pass if the function is written correctly.

Example of an Incorrect Test

Now let's look at a test that is written **incorrectly**, even though the function is correct.

```
def test_add_incorrect_expectation():
    assert add(1, 4) == 6
```

This test will fail and provide the programmer with the following information:

Expected Output	Actual Output
`6`	`5`

When the programmer examines the result of the test, they should be able to notice that the expected output of `5` is what `1 + 4` equals. This should indicate to the programmer that the test is incorrect and needs to be fixed.

Correct the test to:

```
def test_add_incorrect_expectation():
    assert add(1, 4) == 5
```

Now the test will pass.

Example of an Incorrect Function

Now let's flip the situation.

Change the `subtract` function so it is wrong:

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

The test is correct (`10 - 3` DOES equal `7`):

```
def test_subtract_basic():
    assert subtract(10, 3) == 7
```

In this situation, the test fails and provides the programmer with the following information:

Expected Result	Actual Result
`7`	`13`

These results should indicate to the programmer that something is wrong with their function. They *know* that `10 - 3` equals `7`. The function should *not* produce an output of `13`. In this case, the programmer needs to go back and fix their function.

Correct the function back to:

```
def subtract(a, b):
    return a - b
```

Now the test will pass.

Responsible Testing

`pytest` does not know what your function *is supposed to do* or *what results you are expecting*. Only you can know that. `pytest` is only a tool for checking actual results against expected results.

As the programmer, you are responsible for:

- Making sure *you* know what your function is supposed to do
- Making sure *you* are covering all the necessary test cases
- Making sure *you* calculated the expected values of your tests correctly
- Identifying when *you* need to fix the test or fix the function.

Testing with `pytest` is only as effective as the functions and tests you make. If you make ineffective tests, you will get ineffective results.

Never change code blindly just to make tests pass.