Testing

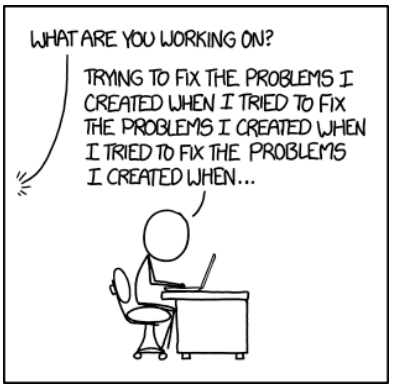

Figure 1: How to make code working - and moreover - **how to make code changeable?** Source: xkcd

Why Testing

Software developers produce bugs. This has nothing to do with experience or knowledge, we all produce code that can fail in one way or another.

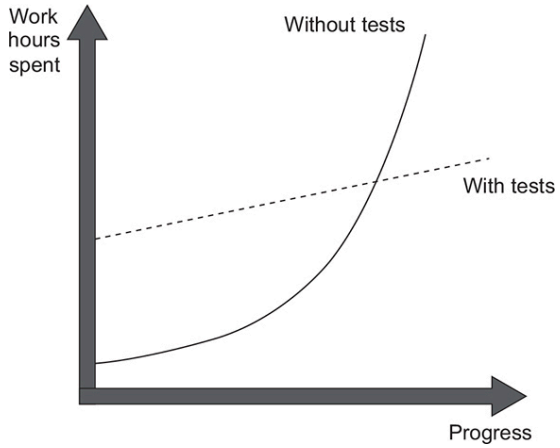

Figure 2: The difference in growth dynamics between projects with and without tests.
Unit Testing Principles, Practices, and Patterns

We need to test our code. Until now, we have simply "*tested*" our code by running it several times, trying different inputs, adding `print()` statements, or running it with a debugger to check what the code does.

This is not only very time consuming, it is usually very limited. You add some functionality, you try it - all is well. However, somewhere else you've "broken" some other functionality, but you don't notice it because you're not looking at it right now.

If you don't test your code and find such **breakages**, someone else will find them, for example your "customer". It would be better to know about possible **bugs** before someone else finds them.

Software development is a **continuous process**, so your project and the amount of source code will grow! It will also evolve over time. Requirements change, bugs appear. It's not "*I write my function/class once and that's it*". Simple output will eventually not be enough.

Besides the debugger and `print`/logging, we will need some more "powerful tools" to **test our code *continuously*** and (semi-)***automatically***.

Different meanings of 'Failures'

What is the difference between **error**, **defect** and **failure**? Source: Software Testing Fundamentals

Error

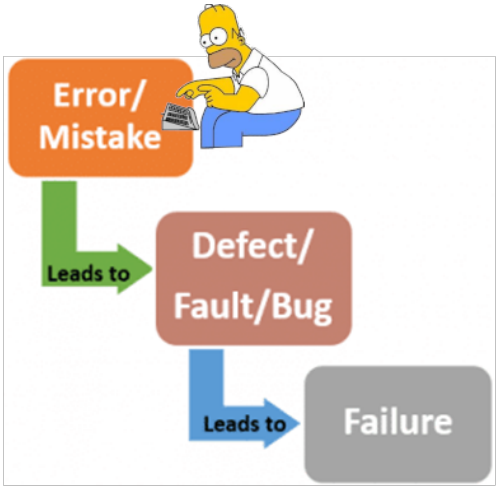

Figure 3: Error or Mistake

An **error** is *"a human action that produces an incorrect result". (ISTQB)*
They can lead to defects, but they don't have to.

For example:

```
def divide(a: int, b: int) -> float:
    return a / b

>>> a = 5
>>> b = input("please enter a number: ")
please enter a number: 0
>>> divide(a, b)
```

Another Example:

```
def multiply(a: int, b: int) -> int:
    """will return the remainder of a int-division"""
    return a + b
```

Reasons for errors could be:

- **Negligence** because of time constraints (*submission gets closer*)
- **Inexperience** of coder
- **Miscommunication** or **Misunderstanding** of requirements or UI/UX
- **Complexity** of code / design /architecture / technology / …

An error in **one place** can cause multiple defects in completely other places.

Defect

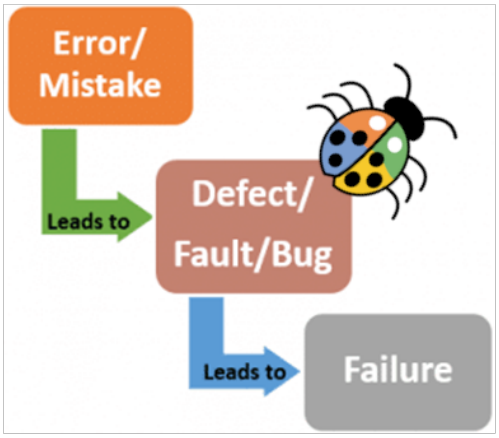

Figure 4: Defect, also called Fault or **Bug**

A **defect** is *"an imperfection or deficiency in a work product where it does not meet its requirements or specifications." (ISTQB)*
Defects do not necessarily lead to failures. They may require very specific conditions to trigger a failure.

For example:

```
def divide(a: int, b: int) -> float:
    return a / b

>>> a = 5
>>> b = input("please enter a number: ")
please enter a number: 2
>>> divide(a, b)
```

Failure

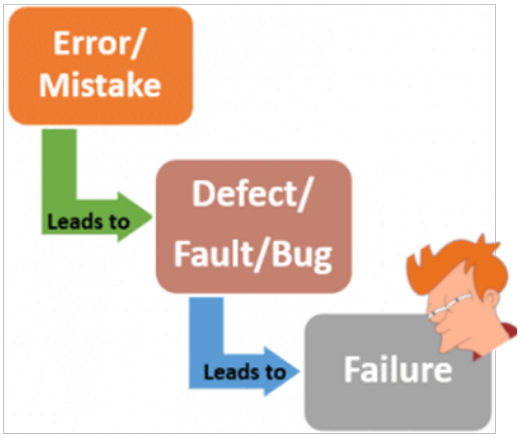

Figure 5: Failure

A **failure** is *"an event in which a component or system does not perform a required function within specified limits." (ISTQB)*
They can lead to unhappy users or lost revenue.

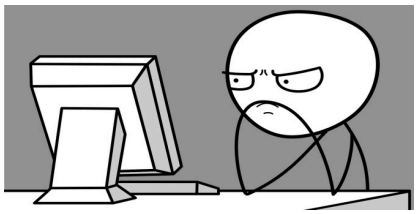

Figure 6: User that run into a failure or your software

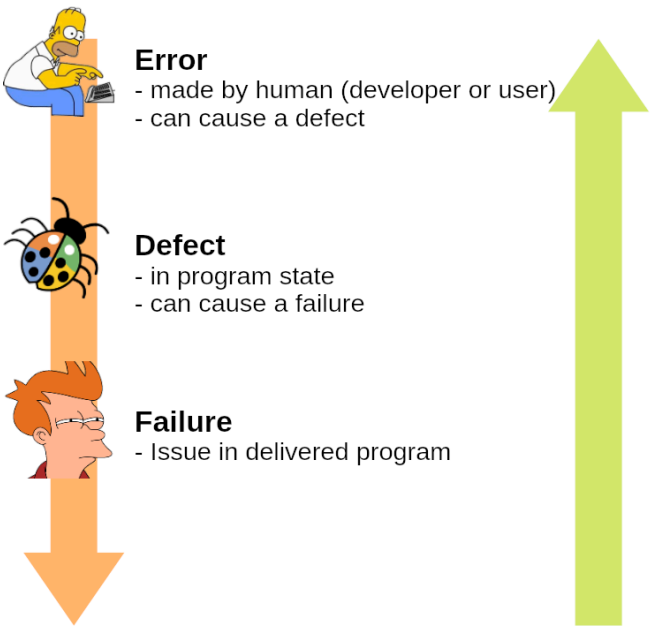

Figure 7: To fix failures, you have to go through the **infection chain**

How Testing

So far, our “testing” has looked like this:

```
print("it work's")

print(f"value: {some_var}")

if foo == bar:
    print("foo should be like bar")
```

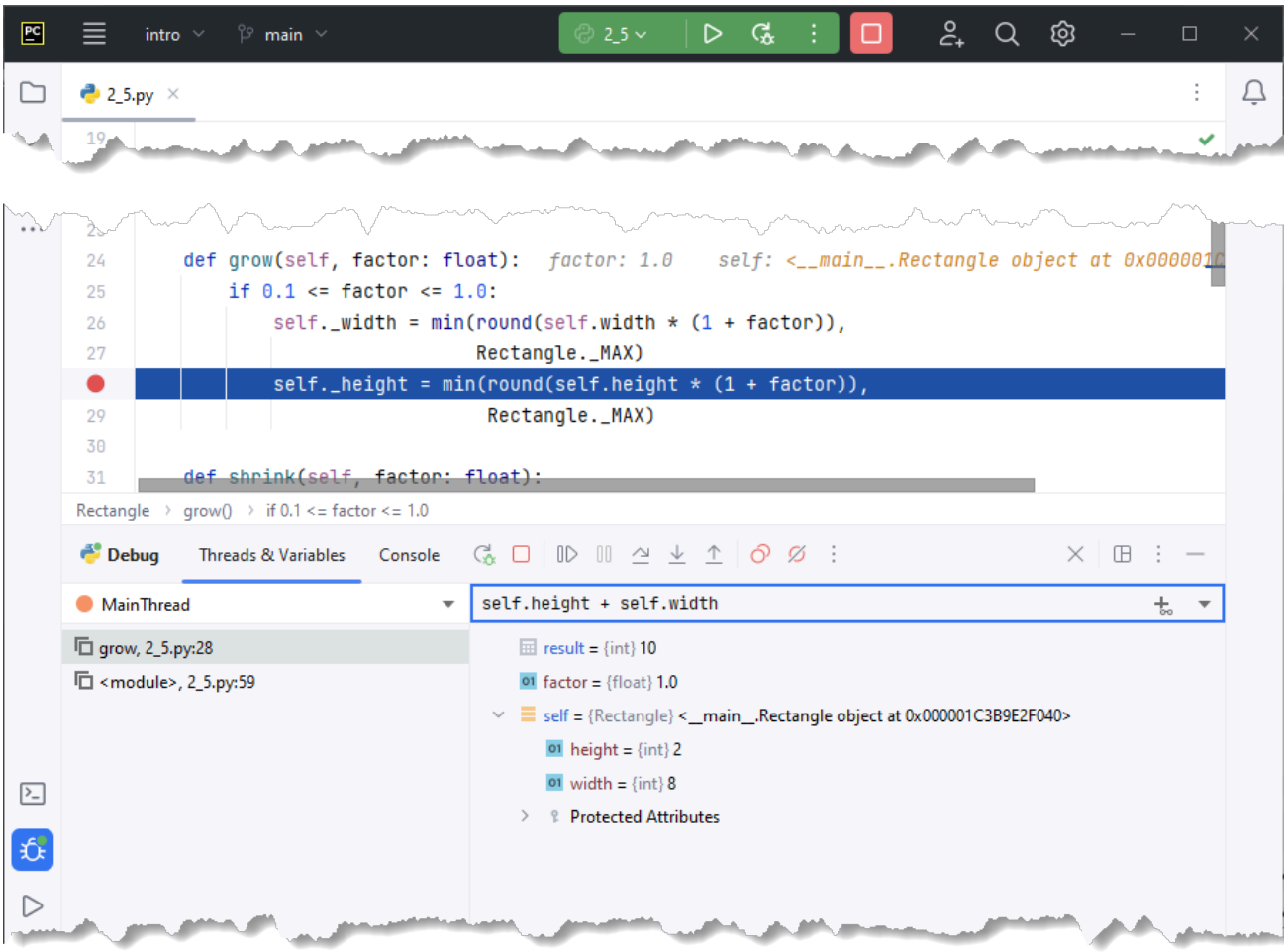

Figure 8: testing using the debugger

> *“Program testing can be a very effective way to show the **presence of bugs**, but is hopelessly **inadequate for showing their absence**.” - Dijkstra*

T.R.A.F.F.I.C - 7 steps of debugging

- **T**rack the problem
- **R**eprodruce it - otherwise it’s not possible to identify a successful fix
- **A**utomate + simplify your code and testing process
- **F**ind possible infection origins - otherwise the defect could lead to other problems

- **F**ocus on most likely origins
- **I**solate the infection chain
- **C**orrect error behind defect **&** start testing **ALL** other tests again

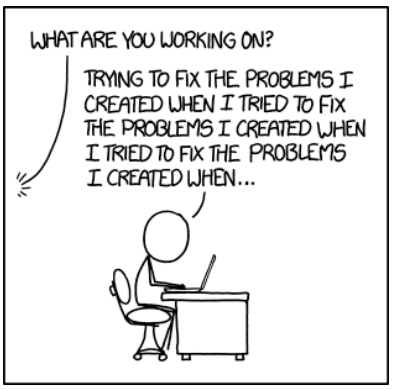

Figure 9: Sounds familiar?

Different types of testing

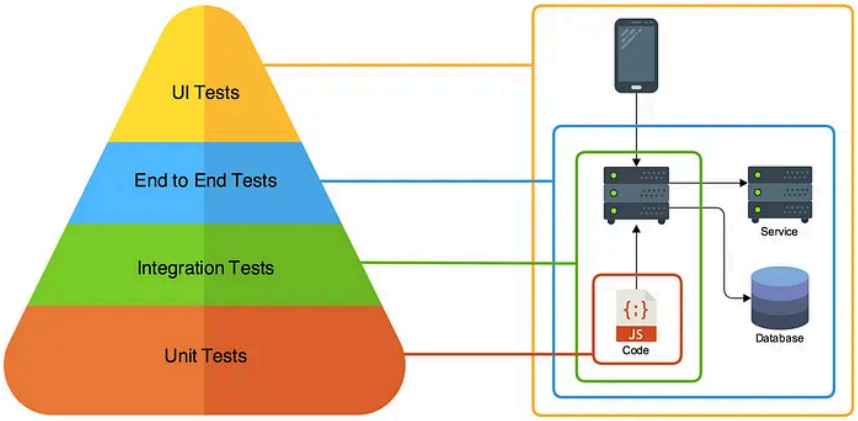

Figure 10: Testing pyramid Source: Medium

With **Unit Tests**, you test the smallest parts of your code. For example, individual functions or classes. **Integration Tests** test the behaviour between different parts (e.g. functions/classes) of your code that are related. **End to End Tests** test a part of your system, including all the necessary parts (e.g. from frontend click to database update). **UI Tests** verify the correct behaviour of your user interface, e.g. are the elements visible and can you interact with them as expected.

In addition to these test categories, there are many others, such as **Performance Tests** or **Stress Tests**, **Security Tests**, etc. In this course we will start with **Unit Tests** and stick to them.

Unit Testing

We will be testing **small pieces of code** - called **units**. Such units are typically single *functions* or *methods* of classes, up to a *class* itself. Today's unit testing - and especially the frameworks used for testing - are based on the **xUnit Testing** frameworks such as JUnit, PHPUnit, etc.

With unit testing, we will test *every* production code - as much as possible. Each test increases your **test coverage** - a measure of software quality. In unit testing, each test should be as small as possible, testing only a small part or a single behaviour.

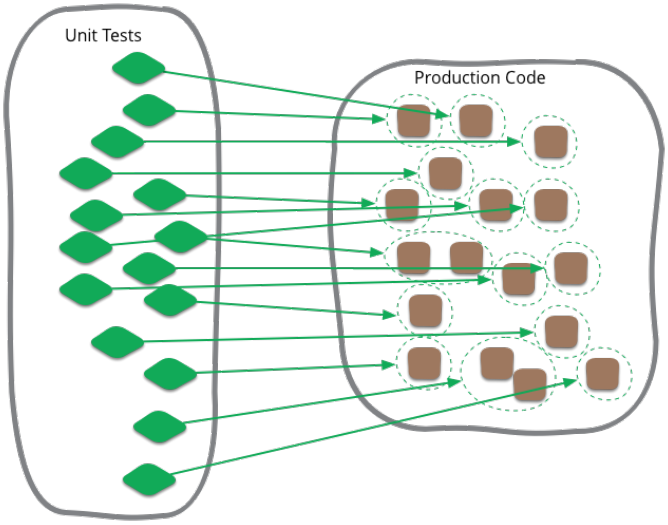

Figure 11: Source: Martin Fowler

Unit tests themselves are code-based, and they **test** the "production source code". The behaviour of the tested methods should be **verified**.

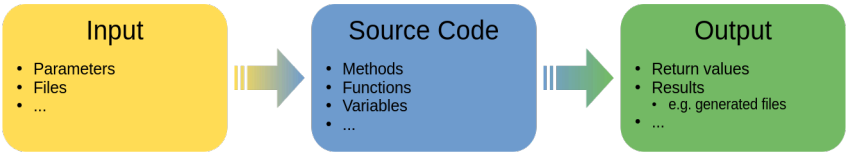

Figure 12: unit test in a nutshell

- use different **Inputs**
- check behaviour of **Source Code**
- verify the **Outputs**

```
def divide(a: int, b: int) -> float:
    return a / b
```

How could tests for this simple function look like?

Parts of Testing

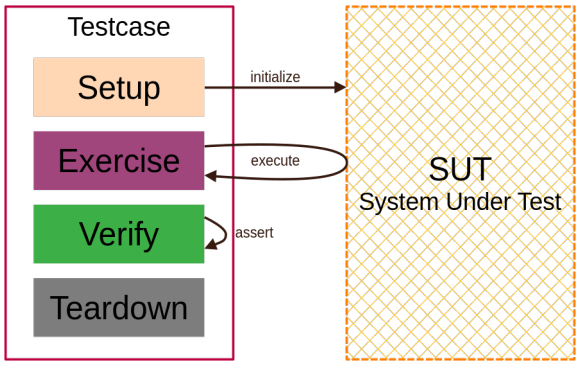

Figure 13: test structure

First of all, what is the **SUT**? The SUT (system under test) is the **piece of code** that we want to write tests for. For example, a function or a class.

```
class MyClass:
    def do_something(self):
        return 'something'
```

Each single test contains up to 4 phases.

- **Setup**: initialise the SUT
- **Exercise**: execute the necessary code
- **Verify**: assert the result of the execution
- ***Teardown***: SUT needs to reset to its pre-setup state

In the **setup** phase we initialise the SUT. For example, we create an object of the class we want to test. Or we prepare data to be passed to a function we want to test.

```
## setup
>>> sut = MyClass()
```

During the **exercise** phase, we execute the method/function we want to test. The *result* is then used for verification.

```
## exercise
>>> result = sut.do_something()
```

With the **verify** phase, we **assert** the *result* of the previous execution.

```
## verify
>>> assert result == 'something', 'should return "something"'
```

> `assert` is a built-in **statement** in Python that could be used as in the example above. It expects a bool expression and an optional error message if the expression would be `False`. In this case an `AssertionError` will be thrown.
>
> ```
> >>> assert 1 == 2, '1 should be 2'
> Traceback (most recent call last):
>   File "<stdin>", line 1, in <module>
> AssertionError: 1 should be 2
> ```
>
> `assert` could be used for testing, but there are better ways, as we see in *a minute*.

At least in the **teardown** phase, we'll bring the sut back to the state before setup/execution, so that each test can run independently! In the case of Python, we usually have nothing special to do, because in the next test we simply create a new object, for example.

> *For example, in a test that interacts with a database, in the **teardown** phase we would clear/reset the changes made during the test. But such tests are far beyond the scope of unit testing!*

unittest module

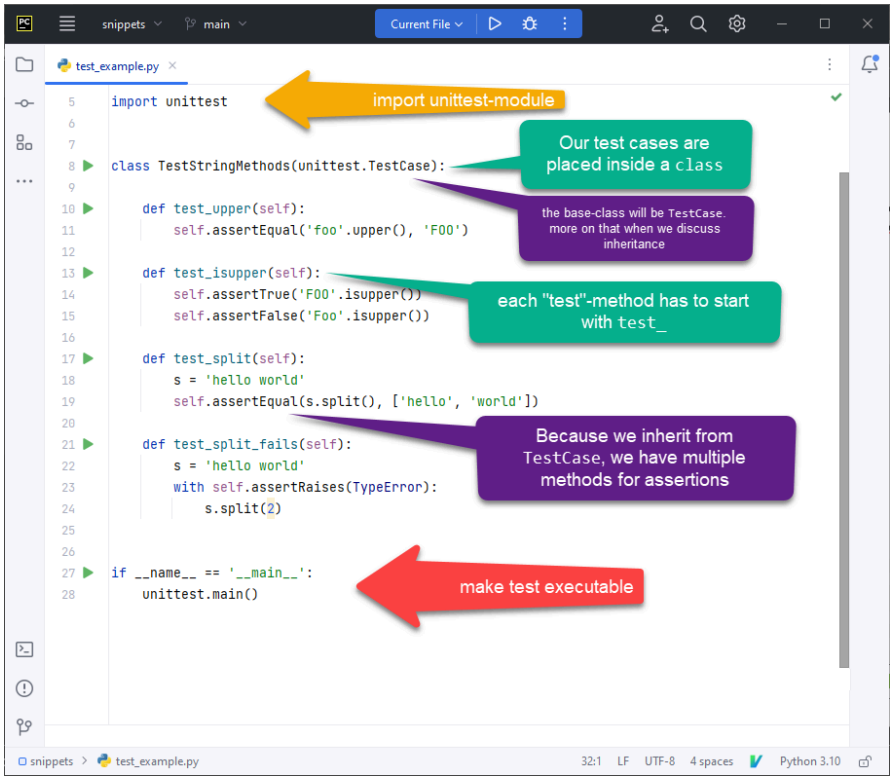

Figure 14: `unittest` structure in Python

In the Python standard library we can find the unittest module, a module based on the xUnit frameworks. It allows us to write software tests in a familiar way (*as long as you are already familiar with other xUnit frameworks*).

A typical test structure will look like:

```
## test_example.py
import unittest

class TestStringMethods(unittest.TestCase):
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])

    def test_split_fails(self):
        s = 'hello world'
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

First we need to import the `unittest` module. The module provides many classes and functions to work with, all we need are at least two parts.

```
import unittest
```

Next, we create our **test-class**. This class groups the tests that relate to a *"unit"*. For example, if we have several classes in our project, we will create a separate test class for each class. Even if we just want to test a single function, all the tests related to that function will be placed inside such a test-class.

The special thing here is that our test class is a child class of the `TestCase` class. Because of the **inheritance***, we don't have to worry about the details, but can concentrate on the test cases. The `TestCase` class also provides a lot of methods to **verify** our test cases.

> *: *more about inheritance in another session.*

```
class TestStringMethods(unittest.TestCase):
```

> *By convention, we prefix our test classes with "Test". For example, if we have a `Rectangle` class, the test class would be `TestRectangle`.*

Within our test class, we create several methods. These methods are the individual test cases. While naming the class as "***Test****Something*" is just a convention, the `test_` prefix to the methods is necessary. This informs the test-runner which methods represent tests and which are for other operations (*such as additional methods for utility*).

```
class TestStringMethods(unittest.TestCase):
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')
```

As mentioned above, the `TestCase` class provides a set of methods to **verify** our test cases. These assertion-methods are all prefixed with `assert` and called on the `self` object.

Method	Checks that
`.assertEqual(a, b)`	`a == b`
`.assertTrue(x)`	`bool(x) is True`
`.assertFalse(x)`	`bool(x) is False`
`.assertIs(a, b)`	`a is b`
`.assertIsNone(x)`	`a is None`
`.assertIn(a, b)`	`a in b`
`.assertIsInstance(a, b)`	`isinstance(a, b)`

Most of the above methods also have **not** equivalents - e.g. `.assertNotEqual()` or `.assertIsNotNone()`.

With `.assertRaises(Exception)` you can also check for exceptions to be raised - without the need of a `try`/`catch` construct.

```
    def test_split_fails(self):
        s = 'hello world'
        with self.assertRaises(TypeError):
            s.split(2)
```

> *note the slightly different syntax using `with`!*

At the end, you provide a `if __name__ == '__main__'` block, to allow the tests called in different ways.

```
if __name__ == '__main__':
    unittest.main()
```

This allows us for example to run the tests directly with Python.

```
$ python test_example.py
python test_example.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

The single dots (`.`) represent the tests that were successfully executed. We also get a little summery about how many tests where executed in what time.

If a test **failed**, we get the details where and why it failed. Also a `F` is displayed instead of a `.` in the summary.

```
$ python test_example.py
..F.
======================================================================
FAIL: test_split_fails (test_example.TestStringMethods)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_example.py", line 23, in test_split_fails
    with self.assertRaises(TypeError):
AssertionError: TypeError not raised

----------------------------------------------------------------------
Ran 4 tests in 0.000s

FAILED (failures=1)
```

Here is another example of a unit test of a function. This function has different “paths” depending on the input. You can recognise it by the two `return’ statements. To test such a function/method, we need at least **two** separate tests. One for each path.

```
## long_string.py
def is_long_string(string: str) -> bool:
    if len(string) > 5:
        return True

    return False
```

```
## test_long_string.py
import unittest
from long_string import is_long_string

class TestLongString(unittest.TestCase):
    def test_short_string(self):
        self.assertFalse(is_long_string('abc'))

    def test_long_string(self):
        self.assertTrue(is_long_string('abcdef'))

if __name__ == '__main__':
    unittest.main()
```

Also checkout [this example for testing a class](#).