64 Create a script and test it

64.1 About

Unit testing is about testing the smallest testable pieces of software. This could be a class or individual methods. In this activity, you will create a class which will be part of a larger application, and create some unit tests to test the methods of the class.

In this activity you will:

- Create a Python class
- Create a unit test class
- Run the unit test
- Create more unit tests

64.2 Create a Python class

Create a new project called **calcFunctionsProject**. Do not create a **main.py** file.
Create a new file in the project called **CalcFunctions.py**.
In the new script, create a new class called **CalcFunctions**. Add three methods called **add_two**, **multiply_two** and **cube**, as follows.

```
class CalcFunctions:

    def add_two(self, nFirst, nSecond):
        return nFirst + nSecond

    def multiply_two(self, nFirst, nSecond):
        return nFirst * nSecond

    def cube(self, nFirst):
        return nFirst ** 3
```

64.3 Create a unit test

Create a new file in the project. Call the file **test_calc.py**.
At the top of the new file import the class you have just created.
Add a method to the file called **test_add_1**.
Inside the new method, create an instance of the **CalcFunctions class** and add an assert that checks that the **add_two** method is working successfully.

```
from CalcFunctions import *

def test_add_1():
    obj = CalcFunctions()
    assert 10 == obj.add_two(4, 6)
```

Run the test and check the output.

```
============================= test session starts =============================
platform win32 -- Python 3.9.1, pytest-6.2.1, py-1.10.0, pluggy-0.13.1 -- C:\Users\sky_a\AppData\Local\Programs\Python\Python39\python.exe
cachedir: .pytest_cache
rootdir: C:\python-workspace\calcFunctionsProject
collecting ... collected 1 item

test_calc.py::test_add_1 PASSED                                      [100%]

============================== 1 passed in 0.01s ==============================
```

The test should be successful, since 6 + 4 does equal 10. If you get a failure or error, or any other output, go back and check the two files (the test script and the CalcFunctions class) before continuing.

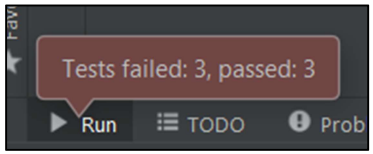
64.4 Create more unit tests

Add five more unit tests to the test script file.
Call them **test_add_2**, **test_mult_1**, **test_mult_2**, **test_cube_1**, **test_cube_2**.

Add code to the methods as follows. Note that the **_2** methods are deliberately incorrect.
The entire script should look like this:

```
from CalcFunctions import *

def test_add_1():
    obj = CalcFunctions()
    assert 10 == obj.add_two(4, 6)

def test_add_2():
    obj = CalcFunctions()
    assert 11 == obj.add_two(7, 5)

def test_mult_1():
    obj = CalcFunctions()
    assert 24 == obj.multiply_two(3, 8)

def test_mult_2():
    obj = CalcFunctions()
    assert 25 == obj.multiply_two(4, 6)

def test_cube_1():
    obj = CalcFunctions()
    assert 27 == obj.cube(3)

def test_cube_2():
    obj = CalcFunctions()
    assert 65 == obj.cube(4)
```

Save and run and examine the output.



```
========================== short test summary info ==========================
FAILED test_calc.py::test_add_2 - assert 11 == 12
FAILED test_calc.py::test_mult_2 - assert 25 == 24
FAILED test_calc.py::test_cube_2 - assert 65 == 64
======================== 3 failed, 3 passed in 0.07s ========================
```

Run the tests on the command line with the **-v** options for more output.

64.5 Explore more

64.5.1 Report

Pytest's output is informative but not very friendly, especially if we are running a large number of tests and need to filter the results to show all failures or all passes and so on, or pass the report to someone else.

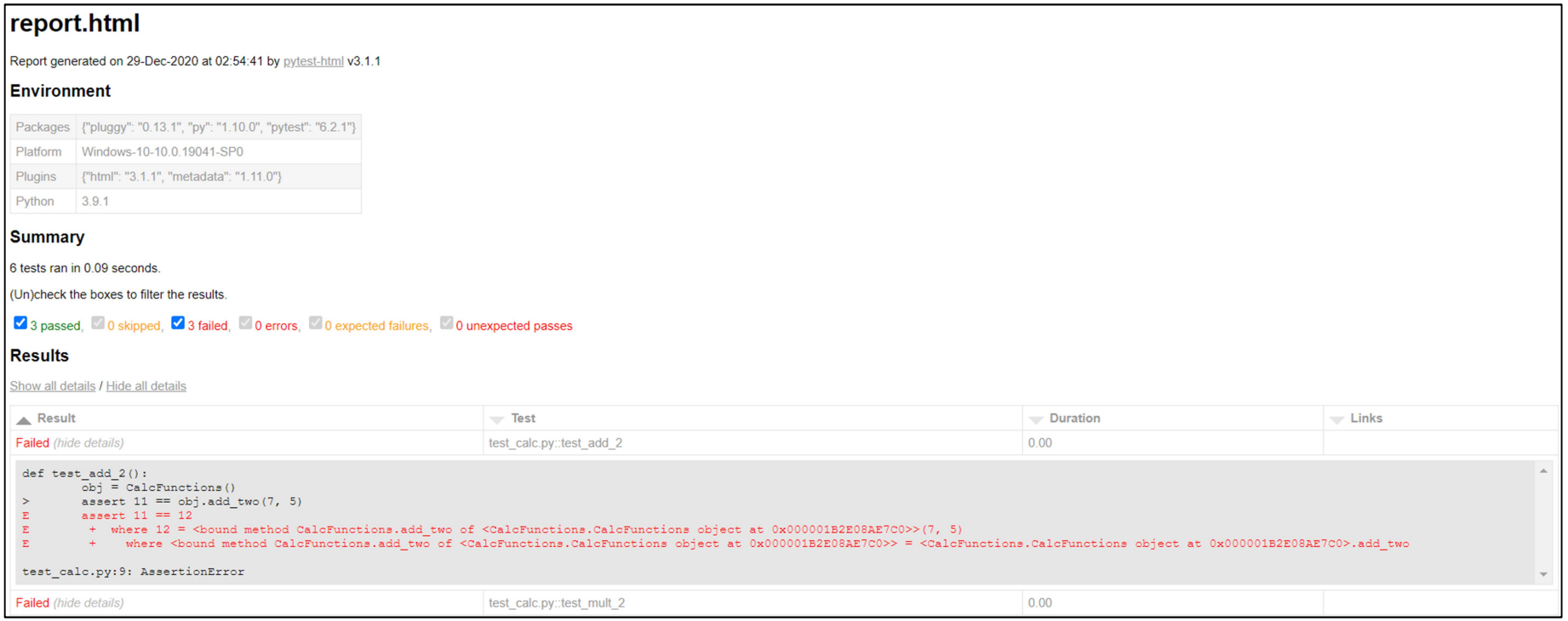
Pytest supports plugins, which can be installed using **pip** (the package manager). Use the command **pip install pytest-html** to install a reporting plugin that produces an HTML report of the Pytest results.

Run the tests on the command line using **pytest test_calc.py --html=report.html**.

Note that the output indicates the file has been created.

```
--- generated html file: file://C:\python-workspace\calcFunctionsProject\report.html ---
```

Open the file explorer and navigate to the project folder and open the file. It will look something like this:



report.html

Report generated on 29-Dec-2020 at 02:54:41 by pytest-html v3.1.1

Environment

Packages	{"pluggy": "0.13.1", "py": "1.10.0", "pytest": "6.2.1"}
Platform	Windows-10-10.0.19041-SP0
Plugins	{"html": "3.1.1", "metadata": "1.11.0"}
Python	3.9.1

Summary

6 tests ran in 0.09 seconds.

(Un)check the boxes to filter the results.

3 passed, 0 skipped, 3 failed, 0 errors, 0 expected failures, 0 unexpected passes

Results

Show all details / Hide all details

Result	Test	Duration	Links
Failed *(hide details)*	test_calc.py::test_add_2	0.00	

```
def test_add_2():
        obj = CalcFunctions()
>       assert 11 == obj.add_two(7, 5)
E       assert 11 == 12
E        +  where 12 = <bound method CalcFunctions.add_two of <CalcFunctions.CalcFunctions object at 0x000001B2E08AE7C0>>(7, 5)
E        +    where <bound method CalcFunctions.add_two of <CalcFunctions.CalcFunctions object at 0x000001B2E08AE7C0>> = <CalcFunctions.CalcFunctions object at 0x000001B2E08AE7C0>.add_two

test_calc.py:9: AssertionError
```

Failed *(hide details)*	test_calc.py::test_mult_2	0.00

Examine the sorting and filtering options available in the report.

Note that there is also a folder called **assets** which contains the CSS file used by the report. To generate a self-contained file (which would be easier to share), add the **--self-contained-html** option to the command line. The report is the same but it requires no external files.

Refer to https://pypi.python.org/pypi/pytest-html for more information about the plugin.

64.5.2 Not equals

In the example above, some of the tests were deliberately made to fail so we could see the output. In reality of course 7 + 5 is not equal to 11, and so on.

Go back and change the **assert** statements for the **_2** tests to use the not equals operator (**!=**), run the tests again and examine the output.

Change them back to the failng version before the next topic.

65 Markers

65.1 About

In the previous section, we wrote some tests, including some which failed. Of course, we knew that the ones which failed would fail.
Pytest allows us to add **markers** to a test script so that we can say that we expect certain tests to fail. We can also mark tests to tell Pytest to skip the test and apply our own user-defined markers and use command-line options to run tests with certain markers.

In this activity you will

- Mark unit tests as “expected to fail”
- Mark unit tests as “skip”
- Mark unit tests with user defined markers

65.2 Before you start

You will need the **calcFunctionsProject** including **CalcFunctions.py** and **test_calc.py** files from the previous section.

65.3 Import the Pytest library

To use markers in a test script, we need to import the pytest library into the script.
At the top of the **test_calc.py** file add the following line:

```
import pytest
```

65.4 Mark unit tests as “expected to fail”

In the **test_calc.py** file, the tests named **_2** all fail because they have incorrect calculations.

Directly above the declarations of **add_test_2**, **mult_test_2** and **cube_test_2**, add the following line:

```
@pytest.mark.xfail
```

Save and run the script and examine the output.
Running it on the command line with **pytest test_calc.py -v** produces the following output.

```
collected 6 items

test_calc.py::test_add_1 PASSED                                          [ 16%]
test_calc.py::test_add_2 XFAIL                                           [ 33%]
test_calc.py::test_mult_1 PASSED                                         [ 50%]
test_calc.py::test_mult_2 XFAIL                                          [ 66%]
test_calc.py::test_cube_1 PASSED                                         [ 83%]
test_calc.py::test_cube_2 XFAIL                                          [100%]

============================== 3 passed, 3 xfailed in 0.10s ===============================
```

65.5 Mark unit tests as “skip”

There is another built in marker that allows us to skip tests, for example if the test code is not complete yet.
Mark the two cube tests as skip by adding this line above the two **test_cube** test methods:

```
@pytest.mark.skip
```

Note that it is OK for a test to have more than one marker- **test_cube_2** now has both **xfail** and **skip** markers.
Save and run and examine the output.

```
collected 6 items

test_calc.py::test_add_1 PASSED                                          [ 16%]
test_calc.py::test_add_2 XFAIL                                           [ 33%]
test_calc.py::test_mult_1 PASSED                                         [ 50%]
test_calc.py::test_mult_2 XFAIL                                          [ 66%]
test_calc.py::test_cube_1 SKIPPED (unconditional skip)                   [ 83%]
test_calc.py::test_cube_2 SKIPPED (unconditional skip)                   [100%]

======================== 2 passed, 2 skipped, 2 xfailed in 0.08s ========================
```

Note that when skipping tests it is possible to add a reason to the marker:

```
@pytest.mark.skip(reason = "Cube function not ready")
```

Run the tests again and note that the reason appears in the output.

```
collected 6 items

test_calc.py::test_add_1 PASSED                                          [ 16%]
test_calc.py::test_add_2 XFAIL                                           [ 33%]
test_calc.py::test_mult_1 PASSED                                         [ 50%]
test_calc.py::test_mult_2 XFAIL                                          [ 66%]
test_calc.py::test_cube_1 SKIPPED (Cube function not ready)              [ 83%]
test_calc.py::test_cube_2 SKIPPED (Cube function not ready)              [100%]
```

The reason text also appears in the HTML report, if it is created.

There is also a **skipif** option that skips a test based on a condition. For example, let's imagine we want to skip the cube tests but only if it is Monday.
Add the **datetime** library to the script by adding this line at the top:

```
import datetime
```

Modify the two **skip** markers to the following. Note that in Python, the **weekday()** function returns **0** for Monday (up to **6** for Sunday):

```
@pytest.mark.skipif(datetime.datetime.today().weekday() == 0, reason = "It's Monday")
```

Save and run and examine the output. If today is Monday, the output will look like this:

```
collected 6 items

test_calc.py::test_add_1 PASSED                                          [ 16%]
test_calc.py::test_add_2 XFAIL                                           [ 33%]
test_calc.py::test_mult_1 PASSED                                         [ 50%]
test_calc.py::test_mult_2 XFAIL                                          [ 66%]
test_calc.py::test_cube_1 SKIPPED (It's Monday)                          [ 83%]
test_calc.py::test_cube_2 SKIPPED (It's Monday)                          [100%]
```

On any other day, the output will look like this:

```
collected 6 items

test_calc.py::test_add_1 PASSED                                          [ 16%]
test_calc.py::test_add_2 XFAIL                                           [ 33%]
test_calc.py::test_mult_1 PASSED                                         [ 50%]
test_calc.py::test_mult_2 XFAIL                                          [ 66%]
test_calc.py::test_cube_1 PASSED                                         [ 83%]
test_calc.py::test_cube_2 XFAIL                                          [100%]
```

Of course, you can test the functionality by changing the script to match whatever the day of the week currently is.

65.6 Mark unit tests with user defined markers

Pytest lets us make up our own user defined markers, which can be used to select which tests to run.

Before continuing, remove (or comment out) all the markers currently in the script.

Before the add, multiply and cube tests in the script, add the following markers respectively:

```
@pytest.mark.add
```

```
@pytest.mark.multiply
```

```
@pytest.mark.cube
```

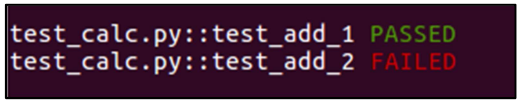
Before the first and last tests (**test_add_1** and **test_cube_2**) also add the following marker:

```
@pytest.mark.mygroup
```

To choose which tests to run, use the **-m** option on the command line. **-m** can choose a specific marker, or use a logical expression.
Use this command:

```
pytest test_calc.py -v -m “add”
```

Note which tests are executed.



Replace the **-m** option with the following and check the test list each time you run the command.

- **-m “add or mygroup”**
- **-m “add and mygroup”**
- **-m “not multiply”**
- **-m “cube and not mygroup”**

65.7 Explore more

65.7.1 Registering markers

You might notice that when you use user-defined markers, the tests run OK but generate a warning message:

```
test_calc.py:32
  C:\python-workspace\calcFunctionsProject\test_calc.py:32: PytestUnknownMarkWarning: Unk
nown pytest.mark.mygroup - is this a typo?  You can register custom marks to avoid this w
arning - for details, see https://docs.pytest.org/en/stable/mark.html
    @pytest.mark.mygroup
```

This appears in case the name of the marker has been entered incorrectly. You can prevent this warning appearing by registering the markers. Refer to the Pytest documentation page mentioned in the warning: https://docs.pytest.org/en/stable/mark.html and register the markers inside the script file.

66 Parameterising a test

66.1 About

Rather than use hard-coded values we may wish to pass several sets of values into a test. We could use techniques such as equivalence partitioning or boundary analysis to choose sets of values to use.

There are several ways that we could parameterise a test- for example we could read values from a file- but pytest has a method for supplying parameters to a test which is convenient.

In this activity you will

- Create a test
- Parameterise the test

66.2 Before you start

This activity uses the **calcFunctionsProject** from the previous activity. Make sure you have that project available.

66.3 Create a test

Create a new file in the project called test_params.py and write a simple test method that verifies the **add_two** method.

```
from CalcFunctions import *

def test_add():
    obj = CalcFunctions()
    assert 10 == obj.add_two(4, 6)
```

Run the test, either on the command line or in the editor, and make sure it passes.

```
collecting ... collected 1 item

test_params.py::test_add PASSED                                          [100%]

============================== 1 passed in 0.01s ===============================
```

66.4 Parameterise the test

To use the parameterisation feature, we have to import the pytest library into the test. Start by adding this line at the top of the test:

```
import pytest
```

Create a set of values to use in the test. The **@pytest.mark.parametrize** (note, the spelling "parametrize" is correct) marker is used to do this. The first argument is the names of the parameters, the following arguments are tuples containing sets of values to use in the test.

```
@pytest.mark.parametrize("first, second, result", [
(4, 6, 10),
(1, 9, 10),
(2, 8, 10),
(3, 7, 10)])
```

Modify the method definition to use the parameters. The parameters work just like normal Python method parameters, so they can be used like variables inside the method.

```
def test_add(first, second, result):
    obj = CalcFunctions()
```

```
    assert result == obj.add_two(first, second)
```

Checkpoint: at this point the whole script should look like this.

```
import pytest
from CalcFunctions import *

@pytest.mark.parametrize("first, second, result", [
(4, 6, 10),
(1, 9, 10),
(2, 8, 10),
(3, 7, 10)])
def test_add(first, second, result):
    obj = CalcFunctions()
    assert result == obj.add_two(first, second)
```

Save and run the script and observe the output.

```
collecting ... collected 4 items

test_params.py::test_add[4-6-10] PASSED                                  [ 25%]
test_params.py::test_add[1-9-10] PASSED                                  [ 50%]
test_params.py::test_add[2-8-10] PASSED                                  [ 75%]
test_params.py::test_add[3-7-10] PASSED                                  [100%]

============================== 4 passed in 0.01s ===============================
```

Note that four tests were executed andeach result is reported together with the values of the three parameters.

Change the last set of values from **(3, 7, 10)** to **(3, 7, 11)** and run the test again. As expected, the last test fails.

```
========================== short test summary info ==========================
FAILED test_params.py::test_add[3-7-11] - assert 11 == 10
========================= 1 failed, 3 passed in 0.07s =======================
```

If we want to run the last test as a negative test we can add the "expected to fail" marker to the set of parameters. Change the **(3, 7, 11)** line to look like this:

```
pytest.param(3, 7, 11, marks = pytest.mark.xfail)])
```

Save and run and examine the output.

```
test_params.py::test_add[4-6-10]
test_params.py::test_add[1-9-10]
test_params.py::test_add[2-8-10]
test_params.py::test_add[3-7-11]

======================== 3 passed, 1 xfailed in 0.07s =======================
```

66.5 Parameterise several tests

Add more tests to the test_params.py file, together with sets of parameters.

```
@pytest.mark.parametrize("multfirst, multsecond, multresult", [
(2, 6, 12),
(2, 9, 18),
(2, 8, 16),
(3, 7, 21)])
def test_mult(multfirst, multsecond, multresult):
```

```
    obj = CalcFunctions()
    assert multresult == obj.multiply_two(multfirst, multsecond)

@pytest.mark.parametrize("cubevalue, cuberesult", [
(2, 8),
(3, 27),
(4, 64),
(5, 125)])
def test_cube(cubevalue, cuberesult):
    obj = CalcFunctions()
    assert cuberesult == obj.cube(cubevalue)
```

Save and run and examine the output.

```
test_params.py::test_mult[2-6-12] PASSED                                 [ 41%]
test_params.py::test_mult[2-9-18] PASSED                                 [ 50%]
test_params.py::test_mult[2-8-16] PASSED                                 [ 58%]
test_params.py::test_mult[3-7-21] PASSED                                 [ 66%]
test_params.py::test_cube[2-8] PASSED                                    [ 75%]
test_params.py::test_cube[3-27] PASSED                                   [ 83%]
test_params.py::test_cube[4-64] PASSED                                   [ 91%]
test_params.py::test_cube[5-125] PASSED                                  [100%]

======================== 11 passed, 1 xfailed in 0.10s ========================
```

These functions are similar to the ones from the earlier section. Add negative tests to the multiply and cube tests:

```
pytest.param(4, 6, 25, marks = pytest.mark.xfail),
```

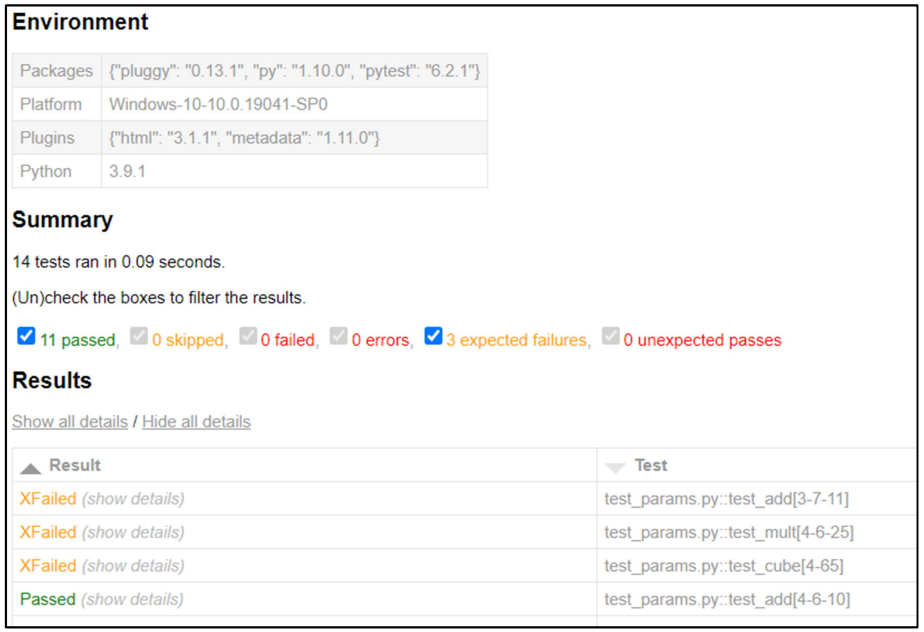
```
pytest.param(4, 65, marks = pytest.mark.xfail),
```

Save and run again, from within the editor and on the command line. Make sure everything runs as expected.

```
test_params.py::test_cube[3-27]
test_params.py::test_cube[4-64]
test_params.py::test_cube[4-65]
test_params.py::test_cube[5-125]

======================== 11 passed, 3 xfailed in 0.10s ========================
```

```
collected 14 items

test_params.py::test_add[4-6-10] PASSED                                  [  7%]
test_params.py::test_add[1-9-10] PASSED                                  [ 14%]
test_params.py::test_add[2-8-10] PASSED                                  [ 21%]
test_params.py::test_add[3-7-11] XFAIL                                   [ 28%]
test_params.py::test_mult[2-6-12] PASSED                                 [ 35%]
test_params.py::test_mult[2-9-18] PASSED                                 [ 42%]
test_params.py::test_mult[2-8-16] PASSED                                 [ 50%]
test_params.py::test_mult[4-6-25] XFAIL                                  [ 57%]
test_params.py::test_mult[3-7-21] PASSED                                 [ 64%]
test_params.py::test_cube[2-8] PASSED                                    [ 71%]
test_params.py::test_cube[3-27] PASSED                                   [ 78%]
test_params.py::test_cube[4-64] PASSED                                   [ 85%]
test_params.py::test_cube[4-65] XFAIL                                    [ 92%]
test_params.py::test_cube[5-125] PASSED                                  [100%]

============================ 11 passed, 3 xfailed in 0.09s =============================
```

Also generate the HTML report as described earlier (add **--html=report.html** to the command line):



Environment

Packages	{"pluggy": "0.13.1", "py": "1.10.0", "pytest": "6.2.1"}
Platform	Windows-10-10.0.19041-SP0
Plugins	{"html": "3.1.1", "metadata": "1.11.0"}
Python	3.9.1

Summary

14 tests ran in 0.09 seconds.

(Un)check the boxes to filter the results.

☑ 11 passed, ☐ 0 skipped, ☐ 0 failed, ☐ 0 errors, ☑ 3 expected failures, ☐ 0 unexpected passes

Results

Show all details / Hide all details

Result	Test
XFailed *(show details)*	test_params.py::test_add[3-7-11]
XFailed *(show details)*	test_params.py::test_mult[4-6-25]
XFailed *(show details)*	test_params.py::test_cube[4-65]
Passed *(show details)*	test_params.py::test_add[4-6-10]