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

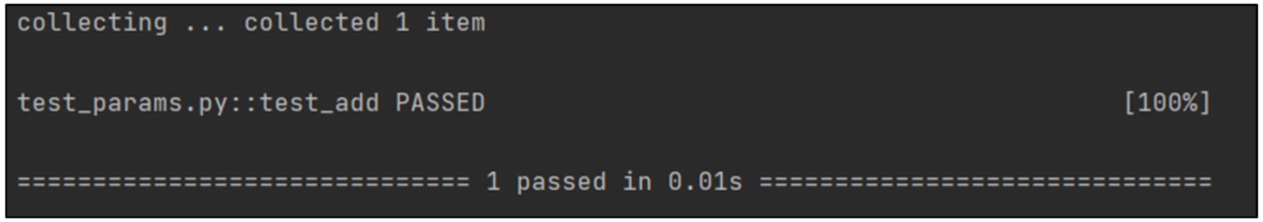
This activity uses the **calcFunctionsProject** from the previous activity. Make sure you have that project available.

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)])
```

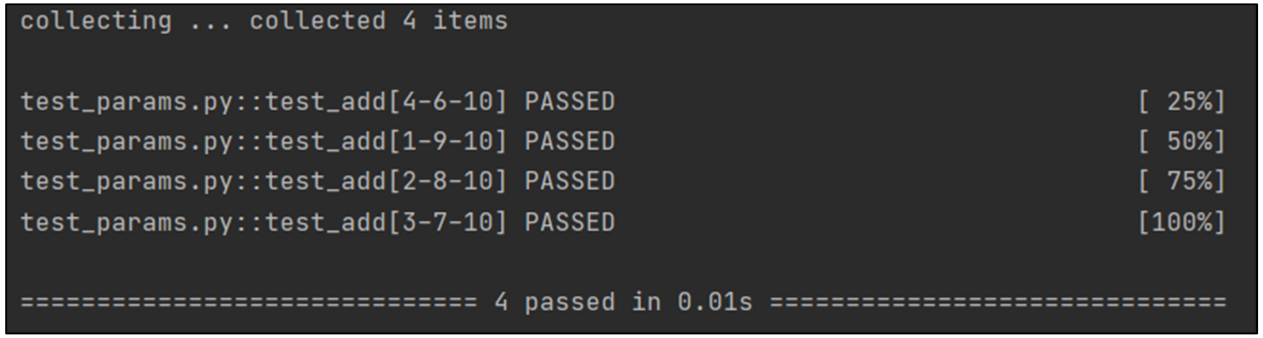
Modify the method definition to use the parameters. The parameters work just like normal Python method parameters, so they can be used like variables inside the method.

```
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)
```

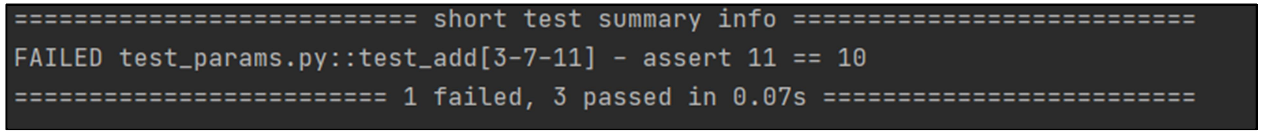
Save and run the script and observe the output.



```
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.

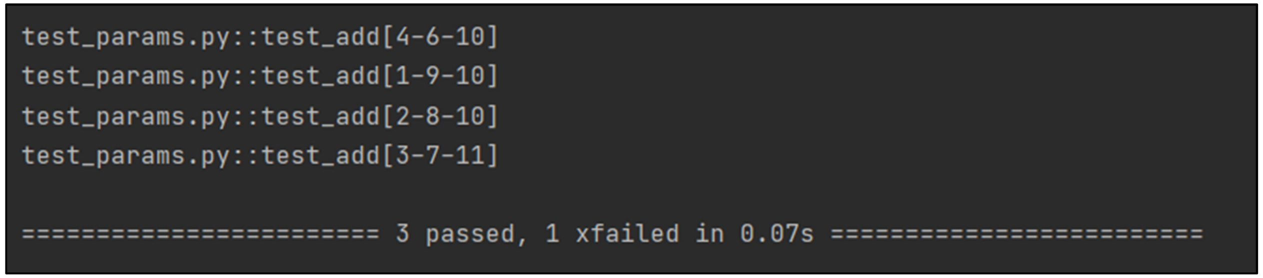


```
========================== short test summary info ============================
FAILED test_params.py::test_add[3-7-11] - assert 11 == 10
======================== 1 failed, 3 passed in 0.07s ==========================
```

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.



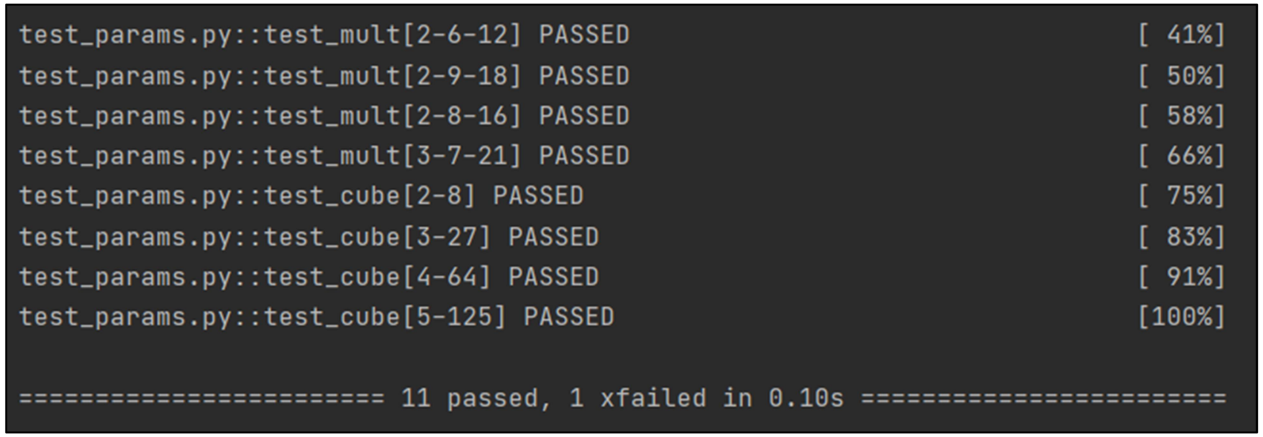
```
test_params.py::test_add[4-6-10]
test_params.py::test_add[1-9-10]
test_params.py::test_add[2-8-10]
test_params.py::test_add[3-7-11]

======================= 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.



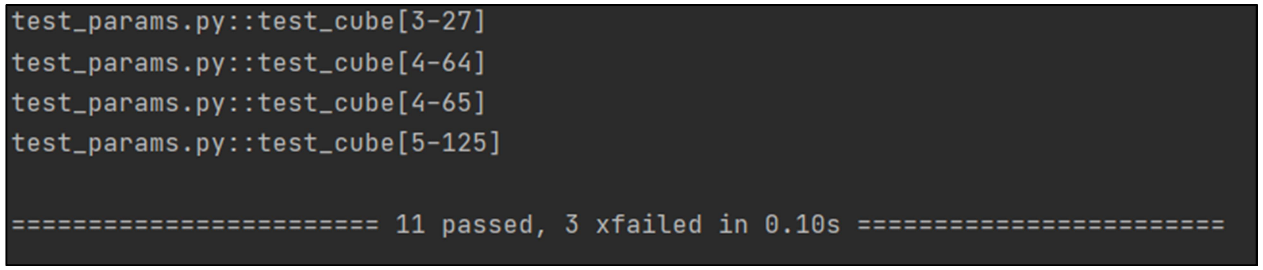
```
test_params.py::test_mult[2-6-12] PASSED                                    [ 41%]
test_params.py::test_mult[2-9-18] PASSED                                    [ 50%]
test_params.py::test_mult[2-8-16] PASSED                                    [ 58%]
test_params.py::test_mult[3-7-21] PASSED                                    [ 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:

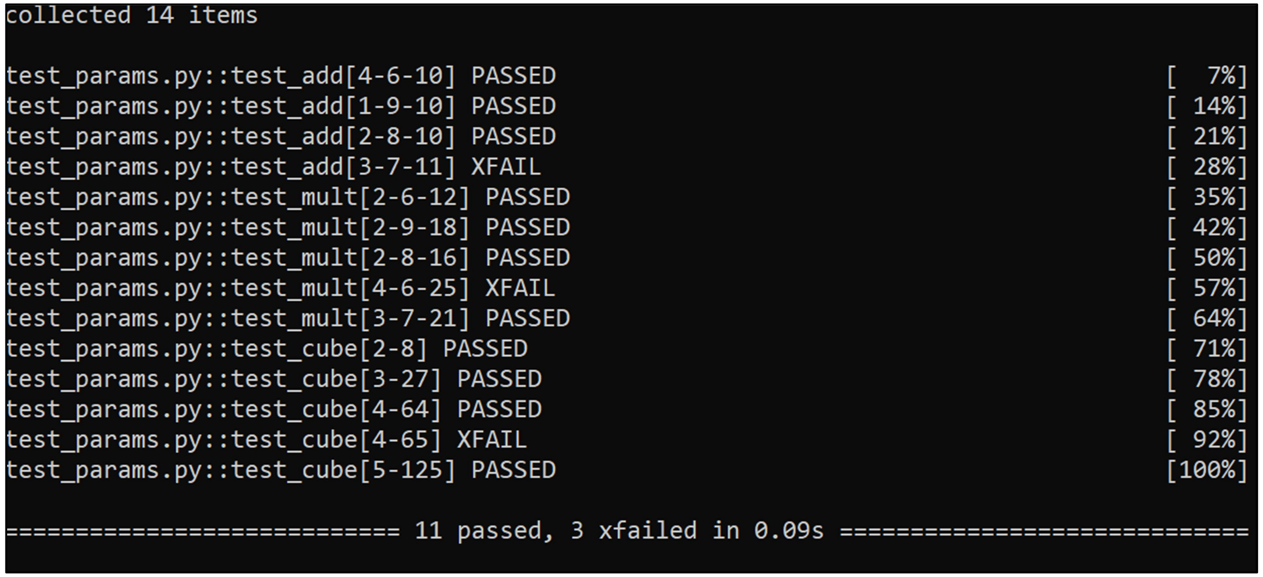
```
pytest.param(4, 6, 25, marks = pytest.mark.xfail),
```

```
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.

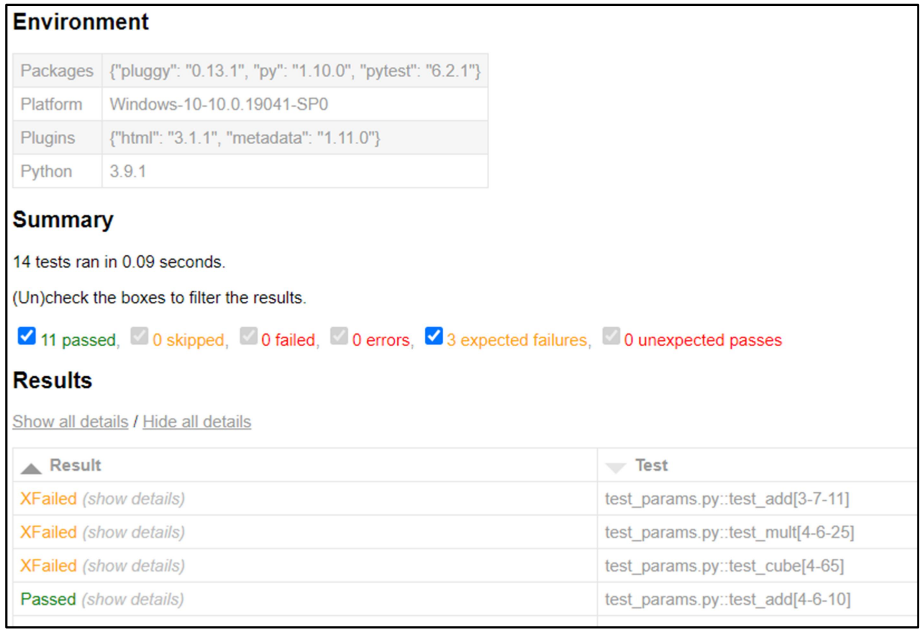


```
test_params.py::test_cube[3-27]
test_params.py::test_cube[4-64]
test_params.py::test_cube[4-65]
test_params.py::test_cube[5-125]

======================= 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]