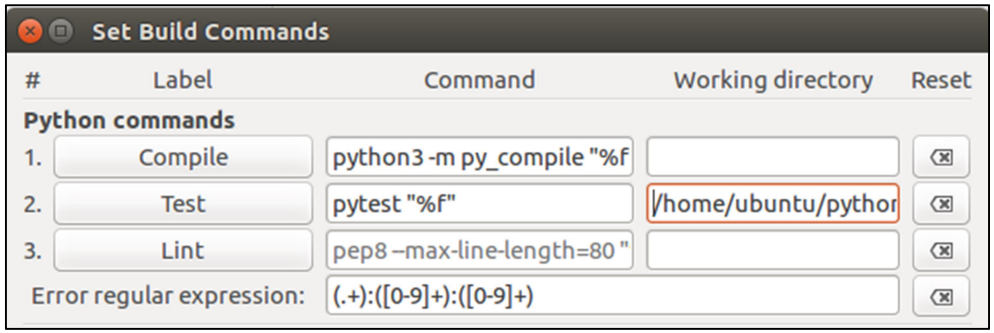
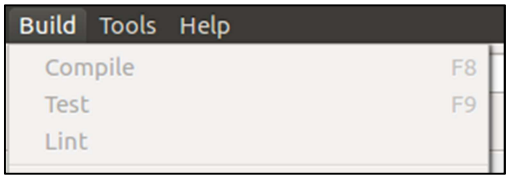


If you haven't already done so this would be a good point to change the **Compile** and **Execute** commands if necessary.

Open the **Build** menu and make sure the new **Test** entry appears. If you don't currently have a file open, it will appear greyed out.



62.5 Setup PyCharm

This section is only relevant if you are using PyCharm.
You must have Pytest installed before doing this part.

In PyCharm, open the settings. If you have a project open, you can pick **File>Settings** from the menu. If you don't have a project open, choose **Customize** then **All settings…**. Both methods will take you to the settings dialog box.
On the settings dialog, pick **Tools** then **Python integrated tools**.
In the **Testing** panel, in the **Default test runner** dropdown list select **pytest**. Note that if you don't have Pytest installed, an error will appear at this point.
OK everything. These settings will become the defaults for all new projects.

63 Pytest basics

63.1 About

This section introduces some basic pytest concepts

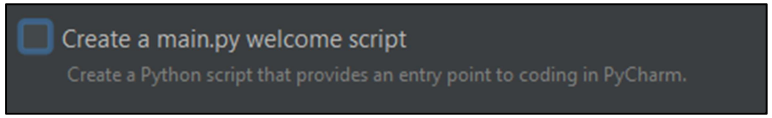
- Creating a test
- Using asserts
- Running a test
- Examining test output

63.2 Before you start

Make sure pytest is installed, as described in the previous section.

63.3 Create a project

Create a new project. Call it **pytestProject**.
In the PyCharm project settings, deselect theoption **Create a main.py welcome script**.



63.4 Create a test file

In the new project, create a new file. Call it **test_pass.py**.
In the new file, write the following script.

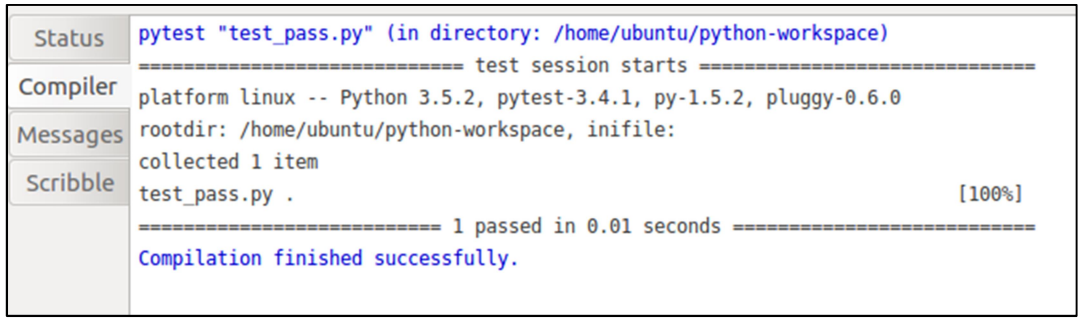
```
def test_pass1():
```

```
    assert 1 == 1
```

Note that the standard naming convention for pytest tests is to call files **test_<name>.py** or **<name>_test.py**. Using **test_** at the beginning means the files will appear together in a directory listing.

63.5 Run the test in Geany

This only applies if you are using Geany. Otherwise, skip this section.
If you have configured Geany as described, you can run the test directly in Geany.
Choose **Build>Test** (the menu option you configured).
The test runs and the output appears on the **Compiler** tab (open the tab if it does not automatically appear).

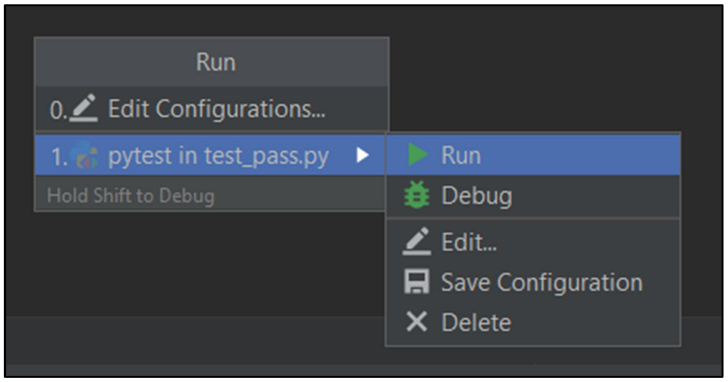


63.6 Run the test in PyCharm

This only applies if you are using PyCharm. Otherwise, skip this section.
There are several different ways to run the test in PyCharm. Try all of them and see which is most natural for you.

63.6.1 On the Run menu

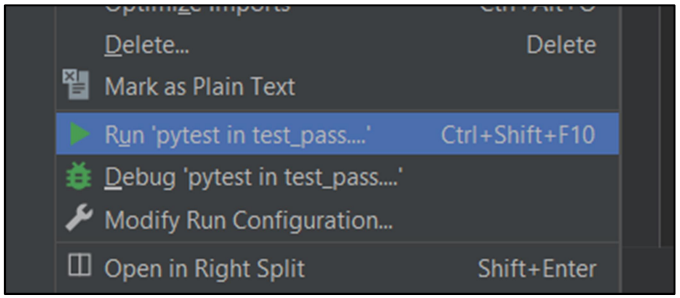
Pick **Run** on the menu then **Run…**
On the dialog box, pick **pytest in test_pass.py** then **Run**.



63.6.2 Right click the file

In the project explorer, right-click the **test_pass.py** file

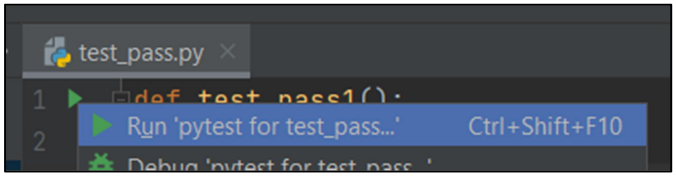
On the popup menu, pick **Run 'pytest in test_pass…'**



63.6.3 Use the run button

With the **test_pass.py** file open, find the run button next to the test name.

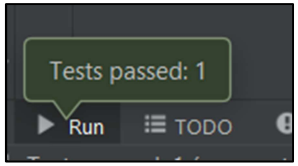
Click it and pick **Run 'pytest for test_pass…'**



63.6.4 Examine the test output

All of the above methods run the test.

As the test runs, the test results appear at the bottom of the screen. Note the quick view in the bottom left that indicates how many tests have run and passed.



Examine the test output in the main panel.

```
Testing started at 01:43 ...
Launching pytest with arguments test_pass.py::test_pass1 in C:\python-workspace\pytestProject

============================= 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\pytestProject
collecting ... collected 1 item

test_pass.py::test_pass1 PASSED                                          [100%]

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

Process finished with exit code 0
```

63.6.5 Examine the results history

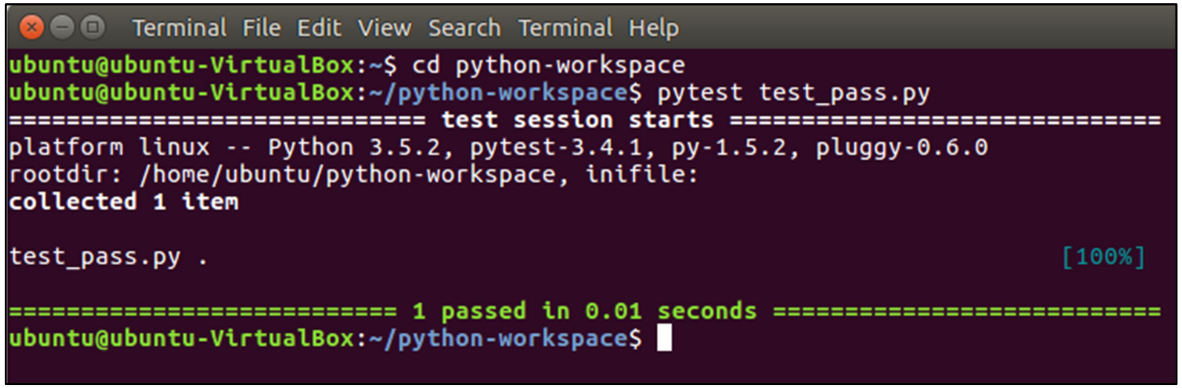
After you have run the test a few times, pick **Run>Test History** on the menu. This opens a sub-menu with a list of your test runs. Pick one and examine the results.

63.7 Run the test on the command line

The same test can be run on the command line. To do this, open a terminal and change to the folder where the test is stored.

Run the test using **pytest test_pass.py**.

Output from Linux and Windows:




```
C:\python-workspace\pytestProject>pytest test_pass.py
========================================= test session starts =========================================
platform win32 -- Python 3.9.1, pytest-6.2.1, py-1.10.0, pluggy-0.13.1
rootdir: C:\python-workspace\pytestProject
collected 1 item

test_pass.py .                                                                               [100%]

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

63.8 Examine the output

In all of these cases the output is the same (except the Ubuntu terminal window version is more colourful).

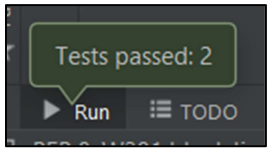
- pytest output always begins with the **test session starts** line.
- The next two lines (beginning **platform** and **rootdir**) give information about the environment where pytest is running.
- The line that says **collected 1 item** indicates that one test file was found and is being executed. It is possible to run many tests in a sequence- we will see that later.
- The next line (after the blank) is the output of the file, in this case **test_pass.py**. The full stop (**.**) shows that a test was passed (there would be a row of full stops if there were many tests in the file).
- The last line (in this case **1 passed in 0.01 seconds**) is a summary of the results.

63.9 Add another test

Add another method to the **test_pass.py** file:

```
def test_pass2():
    assert 2 == 2
```

Save and run the tests and observe the output again. This output is from PyCharm:



```
Launching pytest with arguments C:/python-workspace/pytestProject/test_pass.py in C:\python-workspace\pytestProject

============================= 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\pytestProject
collecting ... collected 2 items

test_pass.py::test_pass1 PASSED                                      [ 50%]
test_pass.py::test_pass2 PASSED                                      [100%]

============================== 2 passed in 0.01s ==============================
```

Note that this time the report says **collected 2 items**, and the summary line reports **2 passed**.

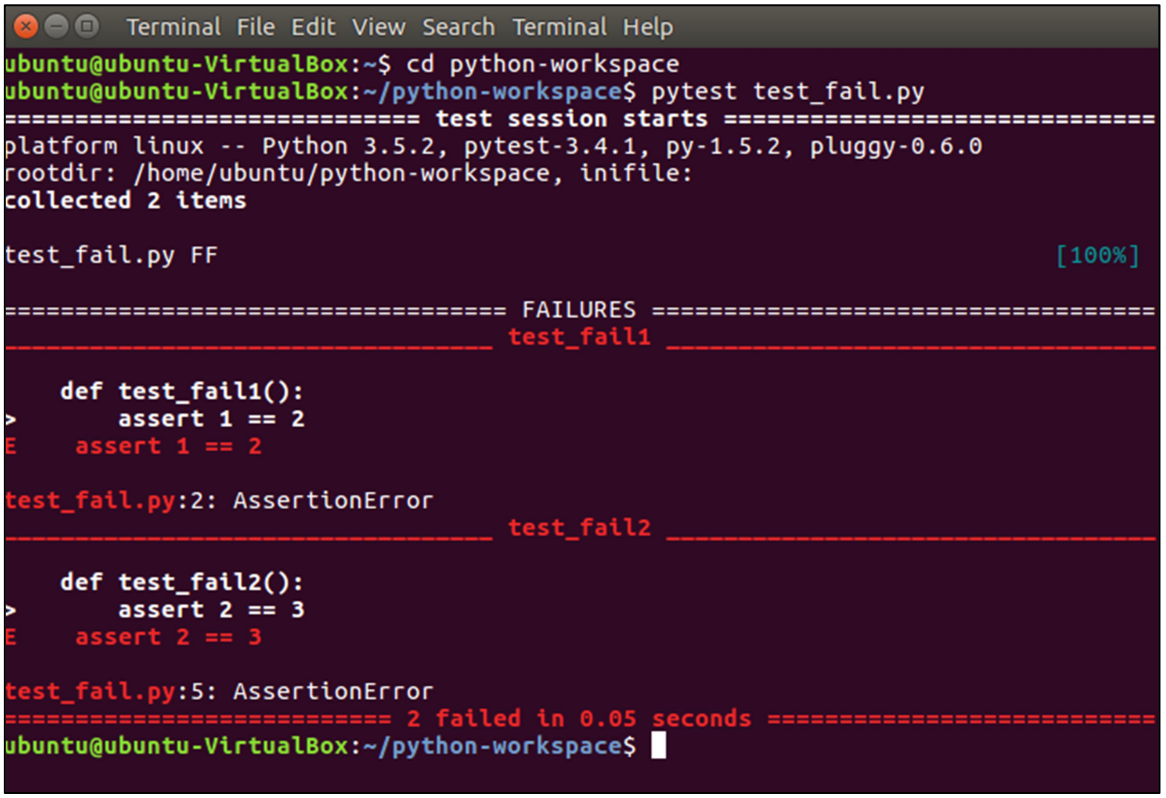
63.10 Create another test file

Create a new file. Call it **test_fail.py**.
Add two new tests to the new file:

```
def test_fail1():
    assert 1 == 2

def test_fail2():
    assert 2 == 3
```

Run the test, either within your editor or on the command line (**pytest test_fail.py**).
Examine the output, which will be effectively the same in both cases.

```
================================= FAILURES =================================
_______________________________ test_fail1 ________________________________

    def test_fail1():
>       assert 1 == 2
E       assert 1 == 2

test_fail.py:2: AssertionError
_______________________________ test_fail2 ________________________________

    def test_fail2():
>       assert 2 == 3
E       assert 2 == 3

test_fail.py:5: AssertionError
========================== short test summary info ==========================
FAILED test_fail.py::test_fail1 - assert 1 == 2
FAILED test_fail.py::test_fail2 - assert 2 == 3
============================== 2 failed in 0.06s ===============================
```

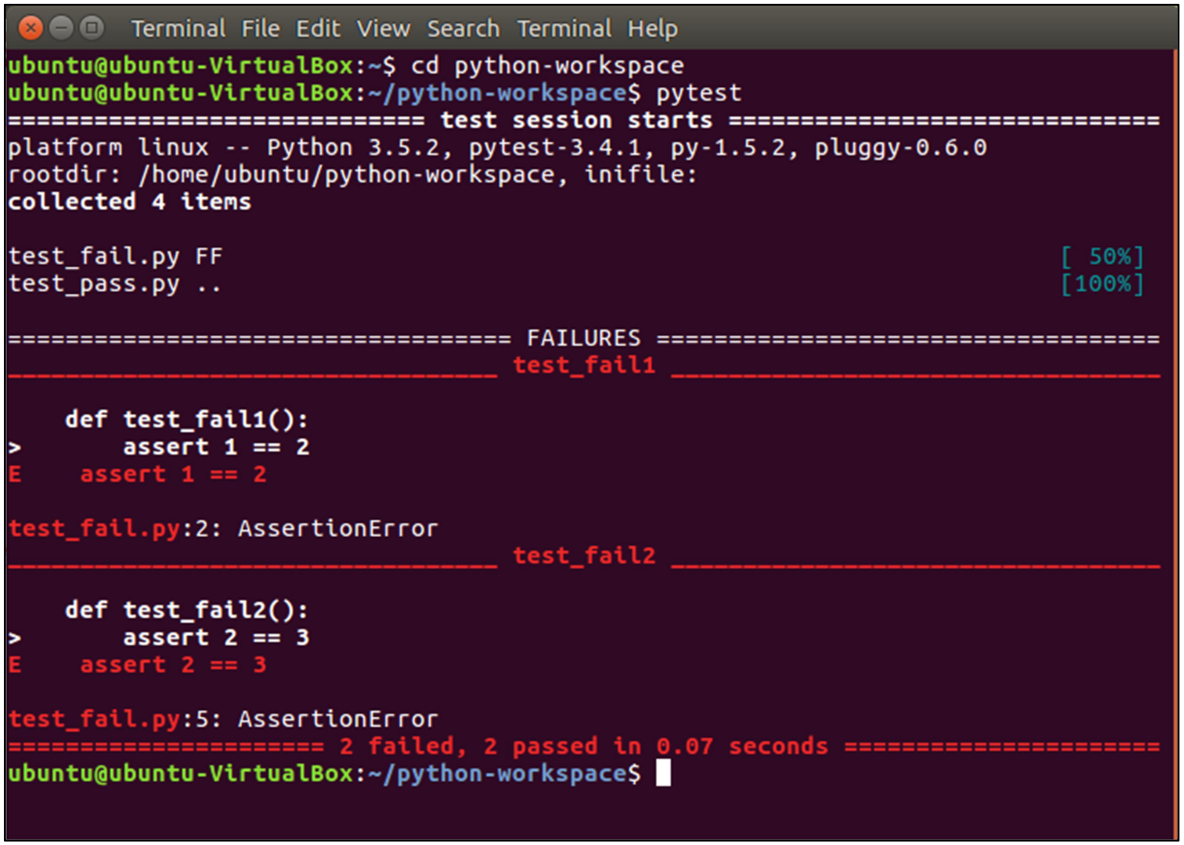


The **FAILURES** section contains information about each failure. In both cases it is indicating an **AssertionError**- that is, the assert failed, which is to be expected since 1 is not equal to 2, and 2 is not equal to 3.
The values next to the file names in the failures (**2** and **5** in the example above) are the line numbers where the failures occurred.

63.11 Run all the tests together

Do this on the command line in a terminal or command prompt window.
Make sure you are in the folder where the two test files are stored.

Type the command **pytest** without any file names. When used without a file name, pytest searches for all the files called **test_<name>.py** (or **<name>_test.py**) in the current folder.
In this case two files are found.



Each of the files contains two tests, so pytest reports that it **collected 4 items**.

Run the tests again, using the command **pytest -v**.

```
ubuntu@ubuntu-VirtualBox:~/python-workspace$ pytest -v
============================= test session starts ==============================
platform linux -- Python 3.5.2, pytest-3.4.1, py-1.5.2, pluggy-0.6.0 -- /usr/bin
/python3
cachedir: .pytest_cache
rootdir: /home/ubuntu/python-workspace, inifile:
collected 4 items

test_fail.py::test_fail1 FAILED                                          [ 25%]
test_fail.py::test_fail2 FAILED                                          [ 50%]
test_pass.py::test_pass1 PASSED                                          [ 75%]
test_pass.py::test_pass2 PASSED                                          [100%]

================================== FAILURES ===================================
_________________________________ test_fail1 _________________________________

    def test_fail1():
>       assert 1 == 2
E    assert 1 == 2

test_fail.py:2: AssertionError
_________________________________ test_fail2 _________________________________

    def test_fail2():
>       assert 2 == 3
E    assert 2 == 3

test_fail.py:5: AssertionError
====================== 2 failed, 2 passed in 0.05 seconds ======================
ubuntu@ubuntu-VirtualBox:~/python-workspace$
```

In this case the **-v** indicates **verbose**, so instead of just dots or Fs pytest gives a bit more information about the state of each test.

63.12 Explore more

63.12.1 Command line options

There are many more pytest command line options. Type **pytest --help** to see them.

We saw the effect of using **-v** to make the output more verbose. Try **-q** to make the output quieter.

Try **pytest --collect-only**. What does it do? In what circumstances might it be useful?

Try **pytest --pdb** to invoke a debugger for each failure or **pytest -x --pdb** to stop after the first failure and debug.

63.12.2 Asserts

assert is a standard Python command that is usually used by Pytest to compare an expected value (in the script) to an actual value (in an application under test). The examples in this section just used equivalence expressions (**==**). Unlike some unit testing frameworks, there is only a single **assert** command, but it can be used for many different types of expression.

Modify one of the tests to use some different asserts such as the following:

```
assert [“Bob”, “Ben”, “Betty”] == [“Bob”, “Ben”, “Barbara”]
assert “Ben” in [“Bob”, “Ben”, “Betty”]
assert (0 or 1) == True
assert (0 and 1) == True
assert "Jackdaws love my big sphinx of quartz".split()[3] == "big"
assert 2 ** 10 > 1000
```

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.

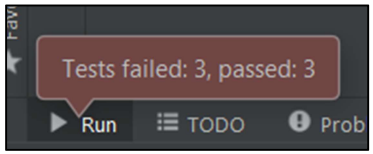
64.4 Create more unit tests

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.

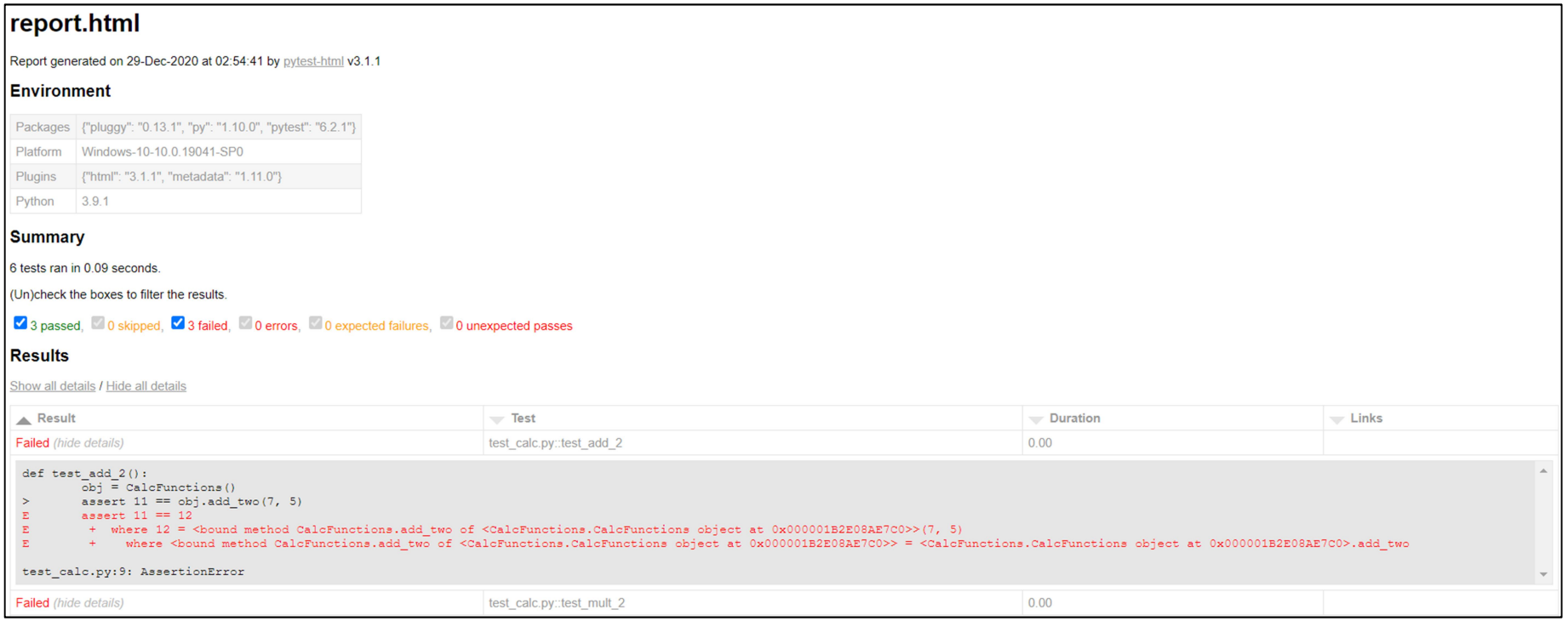
Pytest supports plugins, which can be installed using **pip** (the package manager). Use the command **pip install pytest-html** to install a reporting plugin that produces an HTML report of the Pytest results.

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
```

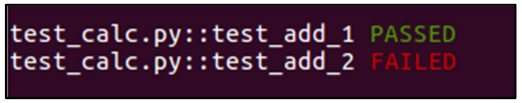
Before the first and last tests (**test_add_1** and **test_cube_2**) also add the following marker:

```
@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.