

Writing Simple Tests Using pytest

The Python tool pytest is a popular framework for unit tests. In this unit, we'll write tests for a single function of production code.

Running pytest

When pytest is run, there are three stages:

- Discovery. The tests scheduled to be run are 'discovered', and then 'collected' and included in a list. The default discovery behaviour will include:
 - All functions with a name that starts or ends with 'test', that are located in files with a name that starts with 'test'
 - All methods that meet the above conditions, provided that their containing class does not have an '__init__' method.
- Execution. The tests are executed, which includes setting up any test resources (if needed) and 'tearing down' these resources afterwards.
- Logging. The test results are logged, to the standard output (the terminal display) but also the results are saved in the specified files and format, so that other processes can use them.

pytest will return an 'exit code' from 0-6, the two most useful exit codes being:

- 0: All tests were collected and passed successfully
- 1: Tests were collected and run but some of the tests failed

How does a test 'pass successfully'?

If a test function completes its execution (i.e. gets to the end of the function code, or else executes a 'return' statement), then pytest treats this test as 'successfully passing'.

Hence, the way for a test to fail is to raise some kind of Exception.

The standard pattern that we use, when deciding whether to not to raise an exception, is to use the assert statement.

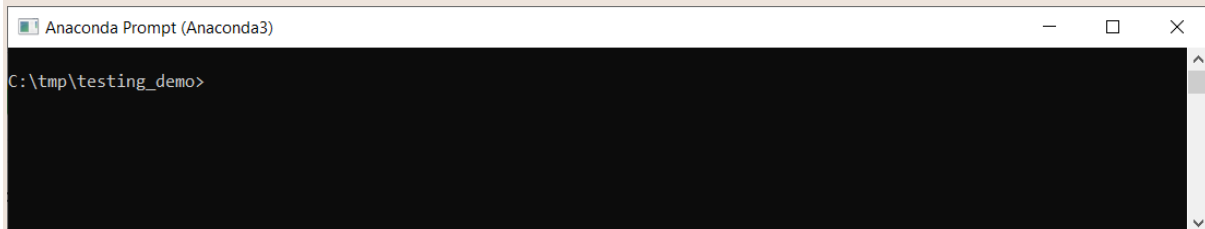
- If the expression following assert is True, then execution continues on the next line with no Exception.
- But if the expression following the 'assert' statement is False, an AssertionError is raised.

Pytest will treat the AssertionError (or any other raised Exception) as a failed test.

Configuring the Environment to Run pytest

We start with an Anaconda Prompt in the (currently empty) root folder of the project we will be working in. Here, the folder 'tmp/testing_demo' is shown, but you should select a folder in your normal documents location to be your 'chosen root folder'.

Use the 'cd' command to change directory to your chosen root folder.



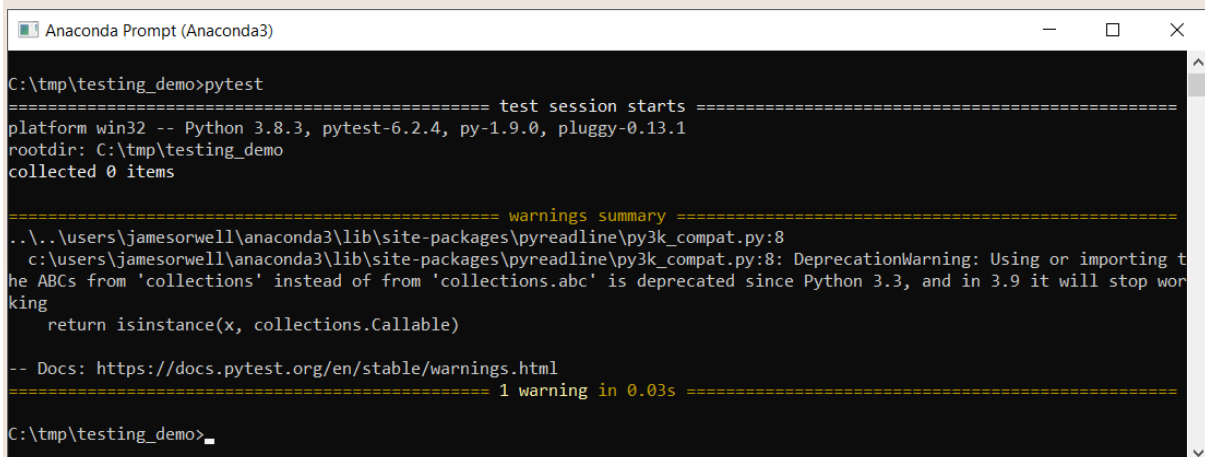
```
Anaconda Prompt (Anaconda3)
C:\tmp\testing_demo>
```

First, check pytest is installed: run 'pip install pytest' + ENTER.

We can run it by typing 'pytest' + ENTER on the command line.

Adding a pytest.ini file

At the time of writing, most versions of python and pytest produce the warning shown in the screenshot below. (If you don't get this warning, you can skip to the next section.)



```
Anaconda Prompt (Anaconda3)
C:\tmp\testing_demo>pytest
===== test session starts =====
platform win32 -- Python 3.8.3, pytest-6.2.4, py-1.9.0, pluggy-0.13.1
rootdir: C:\tmp\testing_demo
collected 0 items

===== warnings summary =====
..\..\users\jamesorwell\anaconda3\lib\site-packages\pyreadline\py3k_compat.py:8
  c:\users\jamesorwell\anaconda3\lib\site-packages\pyreadline\py3k_compat.py:8: DeprecationWarning: Using or importing the ABCs from 'collections' instead of from 'collections.abc' is deprecated since Python 3.3, and in 3.9 it will stop working
    return isinstance(x, collections.Callable)

-- Docs: https://docs.pytest.org/en/stable/warnings.html
===== 1 warning in 0.03s =====
C:\tmp\testing_demo>
```

To fix this warning, we can add the following two lines to the 'pytest.ini' initialization file for pytest:

```
[pytest]
filterwarnings = ignore:.*Using or importing the ABCs.*is deprecated:DeprecationWarning
```

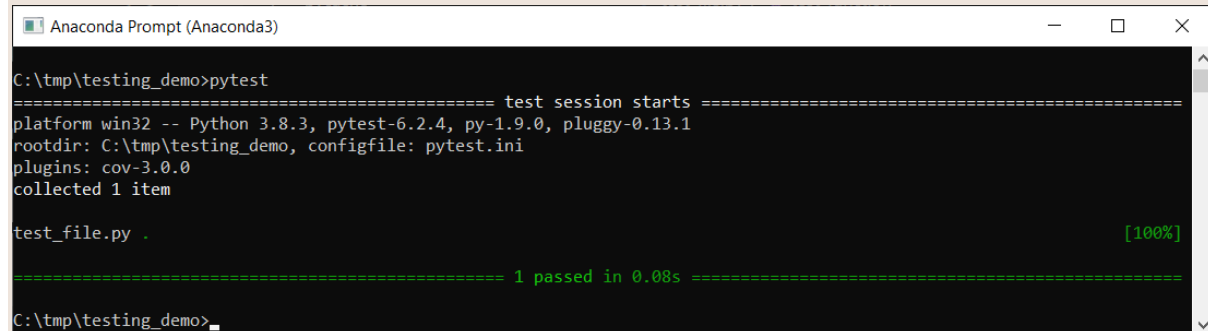
We then should be provided with the more helpful warning, i.e. that no tests were 'discovered' in our root folder (which we should be expecting).

Making a Demo Test Function

In our root folder, let's make a folder called 'tests', and inside that, create a file called 'test_file.py', containing one test function that demonstrates the 'assert' statement (because we'll be using this statement in our real tests):

```
def test_function():  
    print('hello test')  
    assert 6==6
```

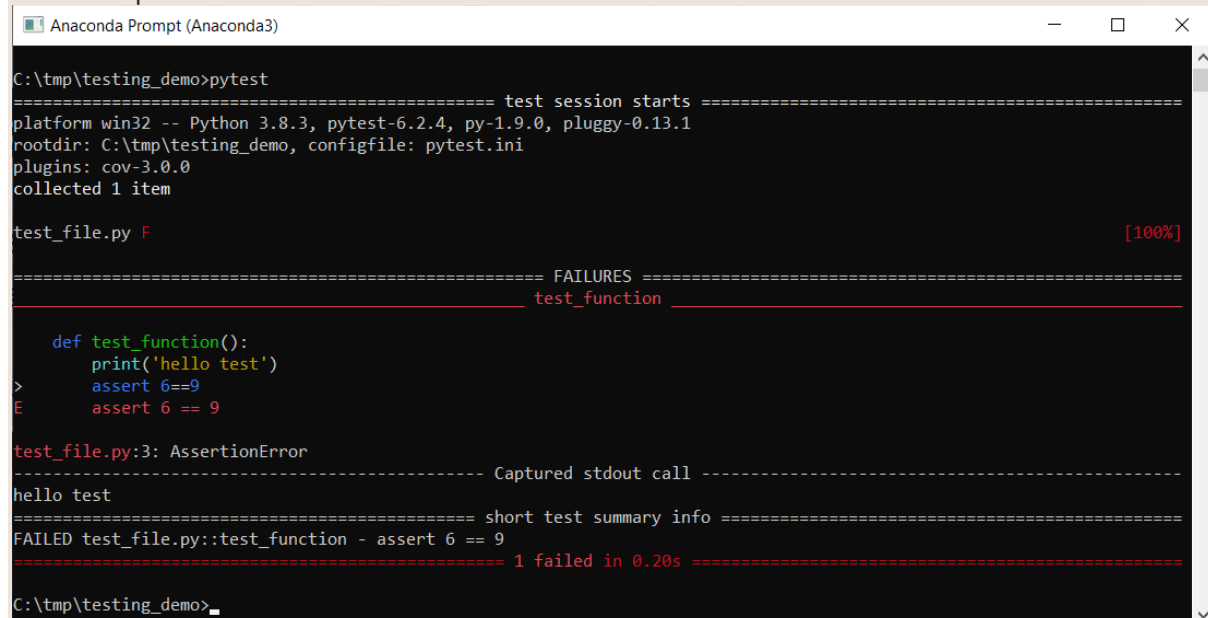
Now, running pytest shows one test 'collected', and this one test passed -- our first 'green test bar':



```
Anaconda Prompt (Anaconda3)  
C:\tmp\testing_demo>pytest  
===== test session starts =====  
platform win32 -- Python 3.8.3, pytest-6.2.4, py-1.9.0, pluggy-0.13.1  
rootdir: C:\tmp\testing_demo, configfile: pytest.ini  
plugins: cov-3.0.0  
collected 1 item  
  
test_file.py . [100%]  
  
===== 1 passed in 0.08s =====  
C:\tmp\testing_demo>
```

This 'green bar' is an important part of the test cycle: it means we can now do some refactoring, or else go on to write another test.

If, instead, we assert something False (like `6==9`), then the test will fail and the print statement will be 'captured' and shown to us:



```
Anaconda Prompt (Anaconda3)  
C:\tmp\testing_demo>pytest  
===== test session starts =====  
platform win32 -- Python 3.8.3, pytest-6.2.4, py-1.9.0, pluggy-0.13.1  
rootdir: C:\tmp\testing_demo, configfile: pytest.ini  
plugins: cov-3.0.0  
collected 1 item  
  
test_file.py F [100%]  
  
===== FAILURES =====  
test_function  
  
def test_function():  
    print('hello test')  
>    assert 6==9  
E      assert 6 == 9  
  
test_file.py:3: AssertionError  
----- Captured stdout call -----  
hello test  
===== short test summary info =====  
FAILED test_file.py::test_function - assert 6 == 9  
===== 1 failed in 0.20s =====  
C:\tmp\testing_demo>
```

Obviously it's a relief to get back the 'green bar' by fixing the above AssertionError. Once that's fixed, we won't be editing test_file.py any more in this unit. It can be deleted from the repository, or it can be kept for reference and for further testing purposes.

We'll now jointly write some unit tests and some production code.

Folder Organization

Inside your chosen root folder, create a folder for the production code called 'my_utils'. This will be our production code package.

Inside the folder 'my_utils', create a zero-length file called '__init__.py'. This file designates the 'my_utils' folder as a package. Python files within this folder are modules.

We store the test code outside the package, so that the package can be deployed without the test code, and also so that the tests are run externally to the package, which is possibly a better testing environment.

Hence, inside your chosen root folder, create a folder for the test code called 'tests'. This will be our test code package.

Inside the folder 'tests', create a zero-length file called '__init__.py'. This file designates the 'tests' folder as a package. Python files within this folder are modules. It's useful to designate the tests folder as a package, because it allows pytest to locate the production code package.

(Another valid configuration of folders is to put the 'tests' folder inside the 'my_utils' package. You can choose this configuration if you plan to distribute your tests along with the production code.)

Our Specified Production Code (to Test)

The function to be written and tested will be named `how_many_shared_lists(list_a, list_b)`.

- This function will be located in a module called 'memory', inside the 'my_utils' package.
- The two input arguments, `list_a` and `list_b`, are both lists containing elements that are either integers or lists of integers.
- The function should return the number of lists that `list_a` and `list_b` have in common.

Writing a Single Unit test

Although some teams will prefer to write some production code first, we're following the 'Test Driven Development' (TDD) methodology here, so we write the following code in a file called 'test_memory.py'.

```
from my_utils.memory import how_many_shared_lists

def test_with_one_shared_list():
    x = [3]
    list_a = [1,2,x]
    list_b = [1,2,x]
    actual_result = how_many_shared_lists(list_a,list_b)
    expected_result = 1
    assert actual_result==expected_result
```

When we run this test, it will fail, but we haven't even started to write the production code yet!

Making the test pass

Now's finally our chance to write some production code! In the file 'memory.py', start off like this:

```
def how_many_shared_lists(list_a, list_b):
    pass # write more code here
```

Can you complete a solution that simply passes `test_with_no_shared_lists` and `'test_with_one_shared_list'`? Have a go at this. If you get stuck, then there's a solution at the bottom of this notebook that you can copy.

After we get the 'green bar', but now with two extra passing tests (yay!) we now get a chance to refactor. There's an opportunity to reduce duplication by making one test function that runs both tests. While we could do this, we actually prefer to handle these types of cases using so-called 'parameterization', which we'll cover in a later reading unit. So we'll skip the refactoring step for now, and move straight on to the next tests.

Writing the next tests

What are the next tests we should write? A development teams makes that decision based on whatever production code they'd like to write next. For an agile environment (and naturally TDD is all about agile), the dev team would like to write the code that implements the stories for this sprint -- whatever they may be.

So, if the priority is to implement stories around data validation (in our case, to ensure the input is only integers and lists), then we first write the failing test for that. (In fact we will prioritise data validation in the very reading unit 6)

On the other hand, if the priority is to transform the output of this function into an html page that presents the number of shared lists in multicoloured lights, then a test is written that can only be passed by writing this transformation component. (Sadly we don't have time to include this test and production code in our backlog.)

For the remainder of this reading unit, let's prioritise the different combinations of lists and integers.

Concept Check: Continue the Test-Driven Development of this Production Code

We propose following the Test-Driven Development cycle of:

- Write the next failing test (or related tests), using a new combination of lists and integers. If you need some hints for some new combinations of lists and integers to provide as input, some ideas are provided at the bottom of this document.
- Edit the production code so that all the tests pass.
- Make the production and test code cleaner by refactoring, if possible.

Next Steps

We've demonstrated how unit tests can be written alongside production code, to check the implementation.

In the next reading unit, we'll show how these unit tests can be used by a continuous integration server to automatically test new versions of the source code.

Solutions and Solution Hints:

Production code that passes the first two tests

```
# a solution that passes our first unit test!
# what tests would show defects in this solution?
def how_many_shared_lists(list_a, list_b):

    num_shared_lists = 0
    for a in list_a:
        if type(a) == list:
            for b in list_b:
                if a is b:
                    num_shared_lists += 1

    return num_shared_lists
```

