

Infrastructure as Code- Walkthrough

Python Tests



Contents

Introduction	2
Deliverables	2
Prerequisites	2
Pylint.....	3
Exercise	3
Unit Test.....	4
Building a test.....	5
Finally.....	7

A **walkthrough** is intended to bring you through a technical exercise. A walkthrough shows you how I completed a task in a particular context, on specific systems, at a point in time. The document is a good basic guide, but you should always confirm that the information is current and if a more recent best practice exists, you should make yourself aware of it.

Introduction

This walkthrough is provided to demonstrate how to get started with writing tests in Python.

When I'm writing simple automation scripts, I write them as functions and then I called the functions from `__main__` as demonstrated earlier in the course. For simple scripts with no dependencies this is fine. However, I also demonstrated how to make modular code, how to build functions which will be permanently reusable. For example, utilities for TCP, udp, cryptography and whatever other code you spend time building. By its very nature this code will be modified and improved during its unlimited operational lifetime. If I make it change in my code for one project, does that break the code for use in other projects.

Do a little bit of reading to understand DevOPS. In a modern environment, we might write code, test the individual components (unit testing), test the integrated project, and deploy it to production, all in an automated workflow. In this walkthrough, I want to look at mandatory requirements for reusable code, the integration of tests.

There is a quote attributed to Martin Fowler...

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

Deliverables

You will be required to keep a list of commands with notes as to why you are using them as well as results of any testing you have done. You may be asked to present these notes as evidence of work completed or as part of a lab book. Check with your lecturer!

Prerequisites

1. I will carry out my exercises using a Windows 11 desktop. You can carry out this work on almost any modern platform.
2. You should have already installed Python as per my notes.
3. You should have already installed VSC as per my notes.
4. You have created a directory to keep example files in and you are ready to code.
 - a. This could be on your OneDrive, in these examples, I am using **OneDrive\Python\Exercises_09**

Pylint

Pylint is a library which examines your code and reports back possible issues, it is a static code analyser. In the terminal window I type **pip install pylint**

I create the following with Notepad++ and save as **pylint1.py**

```
a = 1
b = 2
c = "JOR"

print(a+b)
print(a+B)
print(a+c)
```

pylint not in my path variable, I must give the full path to run it!

My full path was **C:\Users\John.ORaw\AppData\Roaming\Python\Python310\Scripts\pylint**

```
C:\Users\John.ORaw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_08>C:\Users\John.ORaw\AppData\Roaming\Python\Python310\Scripts\pylint test1.py
***** Module test1
test1.py:7:0: C0304: Final newline missing (missing-final-newline)
test1.py:1:0: C0114: Missing module docstring (missing-module-docstring)
test1.py:1:0: C0103: Constant name "a" doesn't conform to UPPER_CASE naming style (invalid-name)
test1.py:2:0: C0103: Constant name "b" doesn't conform to UPPER_CASE naming style (invalid-name)
test1.py:3:0: C0103: Constant name "c" doesn't conform to UPPER_CASE naming style (invalid-name)
test1.py:6:8: E0602: Undefined variable 'B' (undefined-variable)

-----
Your code has been rated at 0.00/10 (previous run: 0.00/10, +0.00)
```

Wow, it gave me 0/10 marks!

The C prefixes are style issues flagged by pylint.

The E prefix is an actual error, I have mistakenly used capital **B** in my code.

Look at <https://pylint.pycqa.org/en/latest/tutorial.html>

Exercise

Rewrite my simple code and get as close as you can to 10/10. Make any assumptions you need. I wrote and saved as **pylint2.py**

```
C:\Users\John.ORaw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_08>C:\Users\John.ORaw\AppData\Roaming\Python\Python310\Scripts\pylint test2.py
-----
Your code has been rated at 10.00/10 (previous run: 8.33/10, +1.67)
```

It took me a few goes looking at the test before I got full marks.

A static code checked is very good for quality assurance in teams.

Unit Test

There is a built-in unit testing library. This allows you to provide input to programmes and check the responses.

Up up to now, we have tested our code by

1. Casually including some lines at the end of the program.
2. Using the `__main__` design pattern
3. Calling functions from `main.py`

I write a file called **formater.py** as below.

```
def convert_upper(my_text):  
    return my_text.upper()  
  
def convert_lower(my_text):  
    return my_text.lower()  
  
def convert_capital(my_text):  
    return my_text.capitalize()
```

I add the following lines to test.

```
print(convert_lower("John ORaw"))  
print(convert_upper("John ORaw"))  
print(convert_capital("dUBLIN"))
```

That all works fine, its just not very scalable or methodical.

Building a test

I create a new file called **test_formatter.py** and I import the **unittest** module, I also import the file I want to test, **formatter**.

I create a new class, inheriting **unittest.TestCase** and create functions for my tests, in this case **test_lower** and **test_upper**.

I add a main section and call **unittest.main()**

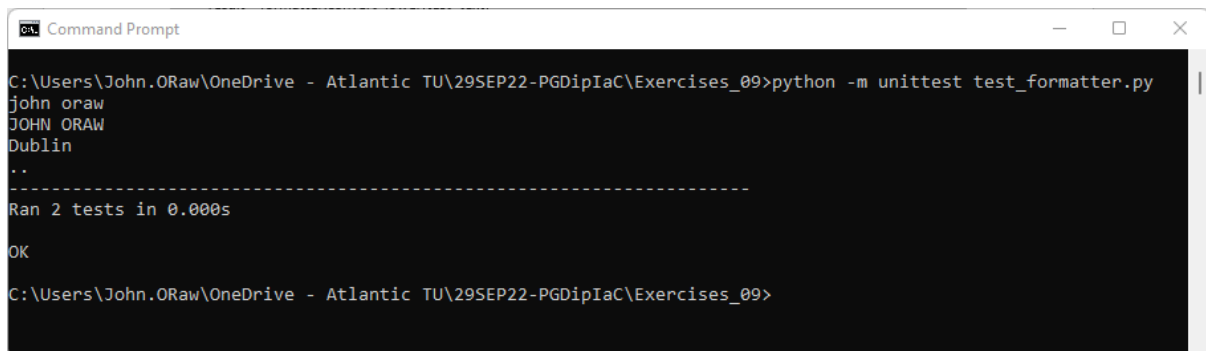
```
import unittest
import formatter

class TestFormatter(unittest.TestCase):
    def test_lower(self):
        test_text = "JOHN ORAW"
        result = formatter.convert_lower(test_text)
        self.assertEqual(result, "john oraw")

    def test_upper(self):
        test_text = "John ORaw"
        result = formatter.convert_upper(test_text)
        self.assertEqual(result, "JOHN ORAW")

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

I need to run the test from the command prompt; it passes!



```
Command Prompt
C:\Users\John.ORaw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_09>python -m unittest test_formatter.py
john oraw
JOHN ORAW
Dublin
..
-----
Ran 2 tests in 0.000s

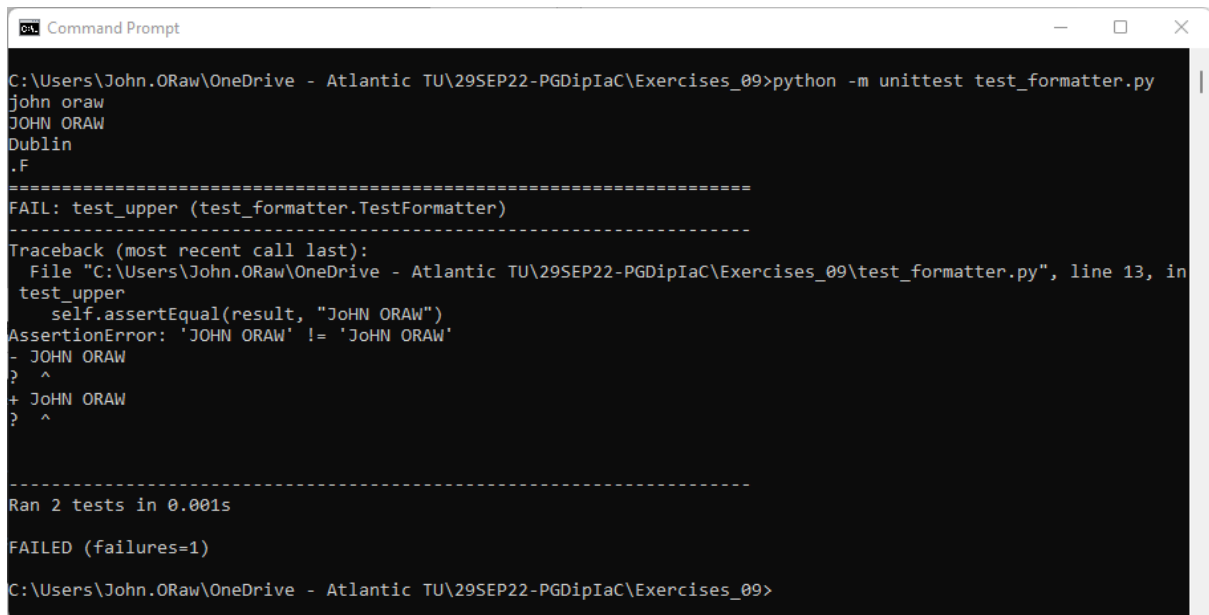
OK
C:\Users\John.ORaw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_09>
```

Note that **-m** runs a library module (unittest) as a script.

Next, I force an error. I edit **test_formatter.py** and mangle the test string.

```
def test_upper(self):
    test_text = "John ORaw"
    result = formatter.convert_upper(test_text)
    self.assertEqual(result, "JoHN ORAW")
```

The test failed as expected and the location of the error was pointed out to me.



```
Command Prompt
C:\Users\John.ORaw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_09>python -m unittest test_formatter.py
john oraw
JOHN ORAW
Dublin
.F
=====
FAIL: test_upper (test_formatter.TestFormatter)
-----
Traceback (most recent call last):
  File "C:\Users\John.ORaw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_09\test_formatter.py", line 13, in
test_upper
    self.assertEqual(result, "JOHN ORAW")
AssertionError: 'JOHN ORAW' != 'JoHN ORAW'
- JOHN ORAW
? ^
+ JoHN ORAW
? ^
-----
Ran 2 tests in 0.001s

FAILED (failures=1)
C:\Users\John.ORaw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_09>
```

There are many different test types you could use in unittest, look through and understand the flexibility you have.

Finally

The examples I used here were trivial, but they did effectively demonstrate unit testing. On a large project, we create the unit tests as part of the project. When code is edited or refactored, we rerun the unit tests and hopefully spot any runtime errors. However, there is a whole school of thinking that says we are doing this backwards.

When you are writing the specification for a software project, how do you write it unambiguously? One way to do so might be to write it in code. What more effective way could there be to do this, than to write the tests. The definition of a completed working program is that it passes the tests.

I used unittest because historically, it has always been in Python. Many programmers advocate other frameworks, like **pytest**.

Test driven development (TDD) is a key programming style, and you should read a little bit about the background to this.

At the very least, if I'm creating a project, it will be documented, structured with modules or classes, separated out as packages if necessary, and with integrated tests.

I showed you how to manually run the tests from the command prompt. Obviously, the final step would be to write a script file (DOS or Linux) to execute these tests. When we consider DevOps, the execution of tests is just one step in a deployment.