Mocking External Resources

So far, we've been writing tests for components that don't have any interaction with external resources -- such as files, folders, databases and internet connections.

For components that do interact with external resources, what's the appropriate strategy for writing unit tests?

We certainly could write tests that use these resources, but for several reasons the preferred approach is to arrange a test in which these resources are simulated or 'mocked', i.e. a mock ('pretend') version of the resource is used.

Below, we'll show how to make a 'mock' file object that can be passed as a function argument (instead of a real file object).

What's the problem with using real resources?

Let's say that we'd like to create a class CsvReader for reading in csv ('comma separated value') data. This toy example is a proxy: in the real-world we use pandas.read_csv, but we frequently need to write a data reading class for other formats, so this is a useful example.

We could prepare a series of 'real' test files: test_input1.csv, test_input2.csv and so on, and put them in a folder called 'test_resources', and write our TestCsvReader class so that CsvReader is tested on these files.

However, many teams prefer to go to the trouble of creating 'mock' resources to simulate the presence of these files. Here are some of the reasons for that preference:

- It's a hassle to split the information about the tests across two document types (the.py source test files, and the .csv input data). These file types will be stored in different folders. It's more convenient to keep the input data for the test alongside the script for the test.
- Maintaining versions and releases gets a little bit more complicated: there's an increased risk of test errors caused by a mismatch between input test files version and test script versions.
- File formats vary across operating systems. In particular, there are different character sequences used for end-of-line (Mac vs Windows). If we use real files, then we may need to support multiple versions of each test file, which is a bullet we'd prefer to dodge
- The test script will run from a given folder, and the file path (to get to the test files) will to be a relative path, starting at that given folder. If we want to run this test script from other folders, the script gets more complicated, as we'll need to adjust the path to the test files accordingly.
- The 'test runner' process that runs the test script may not have permission to access the file system in the required way for the tests. There's a chance that

it won't have 'write permission', for various reasons. This would break the tests.

- On 'Continuous Integration Servers' (such as gitlab or azure devops), access to external resources is more constrained and may need additional configuration.
- Unit tests should run quickly -- to encourage frequent repetition of the tests. Accessing external resources slows tests down significantly. With large projects, this problem gets significant: teams work hard to allow tests to complete in minutes/hours (rather than hours/days)

On the other hand...

There are some limitations to using Mock resources, and it's useful to keep these in mind alongside the advantages:

- There doesn't seem to be any alternative to using real binary files (such as images etc)
- It may also be problematic to store non-ascii set data (accents and characters from other languages) in Python files.
- About the Mac vs PC file types: although we don't need to check both file types work for every single unit test that uses input files, it's still important to have one test that both file types work, if that is what we are supporting.
- More generally, it's useful to have some tests (but not lots of unit tests) that do actually check that the real external resources are working. If there's a newer version of the files we are using, or if an internet connection is down, then we ought to know. But this falls under a different category of test (such as a smoke test or an integration test), to be performed when we need to demonstrate or deploy.

Using 'Mock' Objects as Function Arguments

So far, we've been using the pytest package and program for our tests.

Another unit testing package is called 'unittest': we will use its 'mock' module, which has a class named 'Mock'. We can use an instance of the Mock class as an argument, when calling the function to be tested. We can provide the Mock object with attributes and methods that respond in the same way as the external resources.

For example, the following code shows how a Mock object is created, and given one attribute that simulates a file object's interface (the method called 'readlines'):

```
from unittest.mock import Mock

book_data = [
    'author,title,rating\n',
    'Orwell,1984,8\n',
    'McCarthy,The Road,9\n',
    'T.C.Boyle,Tortilla Curtain,10\n']

open_file = Mock()

# we can add a method to our mock object, with its return value:

open_file.readlines.return_value = book_data

# just like reading from a real open file:

print(open_file.readlines())
```

We can pass this open_file object as an argument for the function or method that we wish to test.

Choosing an Easily Mockable Interface

When designing the our classes, it helps to choose interfaces that can easily be mocked.

For our csv example, if we design the __init__ method to accept an open file object as an argument, then we can pass in a *mocked* open file object instead.

```
class CsvReader:
    def __init__(self, open_file):
        self.columns = []
        # get the lines of csv from the file, storing each column as a separate dictionary

    def get_element(self, row, column):
        pass
```

Mocking open files using the StringIO class

StringIO objects can be written to and read from. They can be made to appear identical to a file object that is opened for reading. In this particular case, as an alternative to the 'Mock' object presented above, it makes a good mock object for an open file.

We can write a test method to jointly check the constructor and the get method, as follows:

```
import pandas as pd
import io


def get_open_test_file():

    mock_file = io.StringIO()

    mock_file.write('author,title,rating\n')
    mock_file.write('Orwell,1984,8\n')
    mock_file.write('McCarthy,The Road,9\n')
    mock_file.write('T.C.Boyle,Tortilla Curtain,10\n')

    mock_file.seek(0) # now ready for reading from the start

    return mock_file


def test_first_column():

    open_test_file = get_open_test_file()

    test_instance = CsvReader(open_test_file)

    expected_values = ['Orwell', 'McCarthy', 'T.C.Boyle']
```

Concept Check: Using Mocked Input Data

Starting with the above two code blocks:

- Complete the test_first_column test, using the 'StringIO' object to mock the open file.
- Complete the implementation for the CsvReader constructor and 'get' methods, check that this passes the above test
- Write a test for the second column of data, again using this mocked input resource.

Concept Check: replacing StringIO mock with 'Mock' mock

rewrite the above tests, replacing the 'StringIO' mock object with a unittest.Mock object, using the example code provided in the first code block at the top of this reading unit.

In fact, the Pandas DataFrame object can be initialized from an open file object, which allows us to write the following:

```
    open_test_file = get_open_test_file()
    df = pd.read_csv(open_test_file)
```

(More frequently, we use a filename as the input argument for the DataFrame constructor, but it works with an open file object too. Official docs here.)

Concept Check: Unit Testing Pandas.read_csv

Write two tests for the pd.read_csv function to show that:

- Column names with a space can be correctly read in from file
- String values containing a quote mark can be correctly read in from file.

You can decide whether to use a StringIO object or a unittest.mock.Mock object, as the input argument when making the call 'read_csv'.

For a stretch goal, refactor these two tests into a single function, decorated with @pytest.mark.parametrize.