Patching External Resources

In the previous reading unit, we saw how a mock object can be used as an argument for a function or method call, which treats the mock object as the 'real' object.

In this reading unit, we'll use this technique to mock other objects used by the function, not just the objects passed in as arguments. In particular, we'll be mocking those objects that are associated with external resources (such as files, databases and internet urls).

We'll be using the mock.patch decorator. There are two common patterns for patching over some 'name' (such as a module, function, method or class) with some mock resources that we have written.

The first pattern uses two arguments in the @patch decorator:

- the first is a string holding the name to patch (the target)
- the second is the name of the 'new' object use instead of the target

We can use this pattern for a simple patch of one function with another function:

```
import mock

# 'target': a string holding the the name to patch, written as package.module.name where
applicable
# new_target: the replacement name to use, e.g. some function that we have written

@patch('target', new_target) # new_target will be used instead of target
def test_some_function_that_uses_target():

    some_function_that_uses_target()  # will use new_target instead

test_some_function_that_uses_target()
```

The second pattern omits the second argument from the decorator, only providing the 'target' string. In this case, the mock target is provided to us, as an implicit argument for the decorated function (similar to the 'self' implicit argument for instance methods). The mock_target is a 'MagicMock' object*.

Use this pattern when greater control and flexibility is required over the 'mock_target':

```
import mock

# target: string holding the the name to patch, written as package.module.name where
applicable

@patch('target')
def test_some_function_that_uses_target(mock_target):  # mock_target is added as an
implicit argument

    mock_target.some_method.return_value = [0,1,2,3,5] # we can specify return values for
arbitrary methods

    # mock_target is a 'MagicMock' object that will be used below, instead of target

    some_function_that_uses_target()

test_some_function_that_uses_target() # no arguments, since mock_target is supplied by
@patch
```

Example: Patching os.listdir

Specification for Production Code:

Write a function 'get_submission_path' taking two arguments, a folder string and a input_filename string (defaulting to 'submission.ipynb'):

- If the folder contains the input_filename, then return the full path of the input_filename (i.e. including the folder)
- If there is only one notebook file in the folder, then return the full path of that filename
- Otherwise, return False

Strategy for writing unit tests:

Patch the os.listdir name with our function that will supply a list of filenames in the test folder.

Solution using first pattern (two arguments in the decorator):

```
def listing_for_test_a(*args, **kwargs):
    return ['submission.ipynb', 'README.md', '.git']

@mock.patch('os.listdir', new=listing_for_test_a)
def test_a():
    folder = os.path.join('virtual', 'folder')
    expected_filename = 'submission.ipynb'
    expected_filepath  = os.path.join(folder, expected_filename)

    actual_filepath =  get_submission_path(folder, expected_filename)
    assert expected_filepath == actual_filepath
```

Here is that same unit test, using the second pattern, i.e. only one argument in the decorator:

```
def listing_for_test_a(*args, **kwargs):
    return ['submission.ipynb', 'README.md', '.git']

@mock.patch('os.listdir')
def test_a(mock_listdir):
    mock_listdir.side_effect = listing_for_test_a
    folder = os.path.join('foo', 'bar')
    expected_filename = 'submission.ipynb'
    expected_filepath  = os.path.join(folder, expected_filename)

    actual_filepath =  get_submission_path(folder, expected_filename)

    assert expected_filepath == actual_filepath
    # Also, MagicMock object stores number of times it is called
    assert mock_listdir.called ==1
```

We've included another check in the above unit test: in the last line, we are checking that our production code has called 'os.listdir' exactly once. The 'MagicMock' object keeps tracks of the number of times it is called.

Your tasks to complete this example:

- Write the production code that passes the above tests.
- Add further tests that check the business logic for other combinations of files present in the folder.
- Run the tests, making changes to the production code (and tests) as necessary.
- Add a further test to check that the production function still returns False when passed in a non-existent folder path.
- Make changes to the production code as necessary to ensure that all tests pass
- Refactor some of your tests by decoratoring a single test function with @pytest.mark.parametrize, passing in a callable that returns the folder listing, and the expected file path.

Concept Check: Writing Tests for Components using External Resources

At Goldman Stanley, a data product is being designed that provides users with a dashboard of recent financial information, using reports delivered from a machine learning pipeline. These reports are stored as json files in a folder, which is then monitored for new items.

For naming the reports in a given folder, the following convention is used:

'report_<x>.json', where <x> is the next integer after the highest positive integer currently found in the folder.

For example, if the folder contained report_1.json and report_2.json, then the next filename should be 'report_3.json'

- If there are no files in the folder corresponding to the template 'report_<x>.json' (where x is a positive integer), then the function should return 'report_1.json'.
- If the folder doesn't exist then the function should return False.

Use Test-Driven Development to write a function called 'get_next_report_filename(folder)', which returns a string (or False) as specified above.

Conclusion and Next Steps

We have learned how to decorate our test functions with mock.patch, which optionally provides us with a MagicMock* object. In the production code, that object will be called, instead of the 'target' name that we've patched.

We used the MagicMock object in three ways:

- We specified the 'side effect', meaning the value that would be returned if we called this target e.g. as a function
- In the previous reading unit, we added methods to the MagicMock object, with return values, to patch the real methods
- We inspected the 'called' attribute of the MagicMock object, which indicates the number of times this object was called. This can be used if we need to ensure that production code does (or doesn't!) make specific calls on the target name.

Here are some further steps to investigate:

- The mock.patch decorator takes an optional keyword argument spec, which can be used to specify the mock object that is created (rather than the default MagicMock* type)
- There is the more powerful 'autospec' option, that automatically generates all the attributes and methods from the target.

In the next (and final) reading unit, we will make a more general survey of what other technology is available for automated testing.

* The type of the patched object will be MagicMock, unless we are mocking an Async function, and then it is an AsyncMock object. But this is beyond the scope of this tutorial.