INFINITYWORKS ACADEMY

Generation

Unit Testing 3

Overview

- Using a Testing Framework
- Intro to `Mock()`
- Intro to `Patch()`
- Exercise

Learning Objectives

- Be able to use a testing framework
- Know how to use an alternative approach to Dependency Injection

Re-cap

- In the first session we learned how to write some basic unit-tests for our `add_two_numbers` function.
- In the second session we learned how to inject *functional* dependencies and mock their return values with stubbed data.

Consider

What happens when our *unit* depends on a module such as `random`?

```python
from random import randint, random

def add_two_random_numbers():
    return randint(1, 10) + random()
```

We could mock parts of the module manually

```
class mock_random():
    def random():
        return 1

    def randint(a, b):
        return 5
```

But we'd likely have to

- Create mocks for each test case
- Modify each one to return the desired result

Is there a better way?

What about a Testing Framework?

pytest & unittest

- Provides a framework upon which to write and run our tests
- Includes helper objects and functions for versatile mocking, and spying
- Provides a test-runner for test detection and verbose results
- Includes additional assertions for diverse testing scenarios

INFINITYWORKS ACADEMY

Generation

Installing `pytest`

You can install it globally with:

```
$ pip install pytest
```

Alternatively, you can add it to your `requirements.txt` inside your virtual environment.

Running `pytest`

1. File names should begin or end with `test`, as in `test_example.py` or `example_test.py`.
2. Function names should begin with `test_`. So for instance: `test_example`.
3. If tests are defined as methods on a class, the class should start with `Test`, as in `TestExample`.
4. You can run `pytest --collect-only` to see which tests `pytest` will discover, without running them.

Example

```python
# test_additions.py
def add_two_numbers(a, b):
    return a + b

def test_add_two_numbers():
    expected = 5
    actual = add_two_numbers(4, 1)
    assert expected == actual
```

Copy the code to a Python file, run `python -m pytest` and watch the output. Hopefully you should see some information about 1 test passing.

`Mock()`

- `Mock()` allows us to create a new object which we can use to replace dependencies in our code
- We can use it to mock primitive functions or entire modules without having to be fully aware of the underlying architecture of the thing we're trying to mock
- Each method / function call is automagically replaced with another `Mock()` object whenever our *unit* tries to access it.

Configuring our Mock ⚙️

`Mock()`

- `return_value`: Specifies the return value when the mock is called (*stub*)
- `side_effect`: Specifies some other function when the mock is called. For example: Raise an `Exception` when testing an unhappy path

—

Example

```
# Mocking a Function
mock_function = Mock()
mock_function.return_value = True
mock_function() # True

# Mocking a Class / Object
mock_class = Mock()
mock_class.some_method.return_value = 1
mock_class.some_other_method.return_value = "Hello World!"
# etc...
```

INFINITYWORKS ACADEMY

Generation

Example Implementation

```python
# With Mock
from unittest.mock import Mock

def test_add_two_numbers():
    # Creates a new mock instance
    mock_get_random_number = Mock()
    mock_get_random_number.return_value = 5

    expected = 10
    actual = add_two_numbers(5, mock_get_random_number)
    assert expected == actual
```

Spying on our Mock 🕵️

Spying allows us to record the behaviour of our mocks and it's parameters which we can use later to make better assertions.

`Mock()`

- `call_count`: Returns the amount of times the mock has been called
- `called_with`: Returns the parameters passed into the mock when called
- `called`: Returns a `bool` indicating if the mock has been called or not

Example

```
mock_function = Mock()
mock_function.return_value = True
mock_function() # True
mock_function.call_count # 1
```

Making Assertions ✓

`Mock()`

- `assert_called()`: Fails if mock is not called
- `assert_not_called()`: Fails if mock is called
- `assert_called_with(*args)`: Fails if the mock is not called with the specified params
- `reset_mock()`: Resets mock back to the initial state. Useful if testing one mock under multiple scenarios

Example

```
mock_function = Mock()
mock_function.return_value = True
mock_function() # True
mock_function.call_count # 1
mock_function() # True
mock_function.reset_mock()
mock_function.assert_called() # Fails
```

What if we don't use Dependency Injection

- We have a legacy app and don't have the resources to restructure it for DI
- We only want to inject certain dependencies, but not built-ins like `print` or `input`

`patch()`

- `patch()` allows us to mock a dependency when we can't, or choose not to inject it.
- It works by intercepting calls to the dependency we've patched and replacing it with a `Mock()`.
- In order to use it we have to *decorate* our test with `patch()`.
- The mocks are then available to use for spying, or making assertions.

INFINITYWORKS ACADEMY

Generation

Example

```python
from unittest.mock import patch

def hello_world(): # No DI
    print("Hello World!") # Dependency

@patch("builtins.print")
def test_prints_hello_world(mock_print):
    hello_world() # Act
    mock_print.assert_called_with("Hello World!") # Passes
```

Example 2

```python
from unittest.mock import patch

def print_name(): # No DI
    name = input("Please enter your name: ")
    print(f"Hello {name}!") # Dependency

@patch("builtins.input")
@patch("builtins.print")
def test_print_name(mock_print, mock_input):
    # Arrange
    mock_input.return_value = "John"

    # Act
    print_name()

    # Assert
    mock_print.assert_called_with("Hello John!") # Passes
    assert mock_input.call_count == 1
    assert mock_print.call_count == 1
```

Configuring our Patch

- `@patch("path.to.module.method")`
- `@patch("src.module.method")`
- `@patch("builtins.input")`

—

Exercise

Duplicate tests from `unit-testing-2` and refactor to make use of `pytest`, `Mock()` and `patch()`.

Learning Objectives Revisited

- Be able to use a testing framework
- Know how to use an alternative approach to Dependency Injection

Terms and Definitions Recap

- `Mock`: A piece of *fake* code standing in to replace some *real* code.
- `Stub`: Dummy data serving to replace real data usually returned from an external source.
- `Dependency`: A piece of code relied upon by another piece of code.
- `Dependency Injection`: A Software Development paradigm in which dependencies are passed as inputs into the function or class which invokes them.

Further Reading

- Dependency Injection
- Handbook: unittest.mock