title: Unit Testing 2

Unit Testing 2

Overview

- Dependency Injection
- Mocking
- Intro to `Mock()`

Learning Objectives

- To be able to explain what *Dependency Injection* is and why we do it.
- To gain experience *Mocking* in order to write well tested code.

Re-cap

In the previous session we learned how to write some unit-tests for our `Rectangle` class:

```
class Rectangle:
    def __init__(self, width, length):
        self.width = width
        self.length = length

    def get_area(self):
        return self.width * self.length
```

do it live

Consider - Scenario 1

Lets add more complexity to our class and try to write unit-tests for `get_price` method.

```
def get_todays_price_per_unit():
    return 500

class Rectangle:
    def __init__(self, width, length):
        self.width = width
        self.length = length

    def get_price(self):
        price_per_unit = get_todays_price_per_unit() # Dependency
        return self.width * self.length * price_per_unit
```

do it live

Consider - Scenario 2

What about this one?!

```
import time

def get_todays_price_per_unit():
    time.sleep(100)
    return 500

class Rectangle:
    def __init__(self, width, length):
        self.width = width
        self.length = length

    def get_price(self):
        price_per_unit = get_todays_price_per_unit() # Dependency
        return self.width * self.length * price_per_unit
```

Are you enjoying your time waiting for the test result?

What problems we may face testing `get_price` method? Write a simple test case for it and run it. Point out: - When we run this function, it also runs `get_todays_price_per_unit`. - `get_todays_price_per_unit` may also have it's own deps, and thus execute those. - We don't necessarily know (without looking) what `get_todays_price_per_unit` itself is going to depend on. - If we leave it as it is, our test will also indirectly test the deps, and deps of deps, which is _Integration Testing_. - We want to test _only_ the `get_price` function.

What happens when our *unit* depends on the outcome of some other piece of code? How can we then test our *unit* in isolation?

This is what the rest of this session is all about!

What is a Dependency

Our *units* may depend upon other functions, libraries or external services in order to do their job. We call these `dependencies`.

Example dependencies:

- REST API
- MySQL Database
- File Store
- Print / Input / Math etc
- Any more?

How do we do that then?

Can you do dependency injection?

- Yes: Mock it (Today's topic)
- No: Patch it, then Mock it

Dependency Injection (DI)

By injecting the dependency, the caller of our function is responsible for providing the `get_todays_price_per_unit` logic.

```
 # Inject price_getter_function dependency
def get_price(self, price_getter_function):
    price_per_unit = price_getter_function() # Execute dependency
    return self.width * self.length * price_per_unit
```

Please take a few moment and think how this change (DI) is going to help us with testing?

Which means that

- When we call `get_price` in our application, we inject the real `get_todays_price_per_unit` function
- When we call `get_price` in our test, we inject a fake (*mock*) `mock_get_todays_price_per_unit` function

---

The Real Function

```
def get_todays_price_per_unit():
    api_url = "http://www.randomnumberapi.com/api/v1.0/random?min=100&max=1000"

    response = requests.get(api_url)

    if response.status_code == 200:
        return json.loads(response.content)[0]
    else:
        return None

class Rectangle:
    def __init__(self, width, length):
        self.width = width
        self.length = length

    #Inject price_getter_function dependency
    def get_price(self, price_getter_function):
        price_per_unit = price_getter_function() # Execute dependency
        return self.width * self.length * price_per_unit

r = Rectangle(2, 10)
print(r.get_price(get_todays_price_per_unit))
```

Ask: - What do we think this code is doing at a high level? - Why would it be bad to use an external service? - Run it several times, see the changes in result? Points: - Main reason: Control. Using an external service, we depend on data that is out of our control for our test. - In this particular case the API will return the same data each time, but what if it didn't? For example an API which provides recent news articles. - Other reasons: Test execution speed, Test parallelisation, Cost to use external API, API Credential management complexity, Service usage limits...

The Mock Function

```
def mock_get_todays_price_per_unit():
    return 500
```

Ask: - How much data SHOULD we return here? - Depends what we are trying to test. In many cases, just the single data point we need might be enough - If the test is to ensure the correct country is chosen from the list then we could add more to help guard against a naive edge-case implementation, e.g. that just returns the first from the list

---

Exercise [code along] - 1

- Try to write unit-test for `get_price` of the `Rectangle` class.

- Explain Dependency Injection as a design pattern. (Also called Dependency Inversion or Inversion of Control) Gives caller control of dependencies. Dependencies are made explicit as they are described in the contract (function or method signature) - White Box vs Black Box testing - Unit testing is very strictly a White Box test - we CAN and absolutely do need to look at the internals to write a good test - and we can refactor the internals

to make the test easier - Demonstrate using Postman or similar to inspect the response data from the API and determine what is required for our test case

---

Exercise [code along] - 2

- Try to understand what `random_list_generator` function does, then do DI on it and then write unit-tests to verify its functionality.

```
import random


def get_random_number():
    return random.randint(1, 10)


def random_list_generator(n):
    result = []
    for _ in range(n):
        result.append(get_random_number())
    return result
```

Some Caveats of DI

- May require restructuring of your code if retro-fitting.
- Tests will be so easy to write you may die of boredom.
- Your colleagues will be envious of you.
- Recruiters will keep blowing up your phone.

Exercise

Instructor to distribute exercise.

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 the Testing Frameworks?

`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

```
from unittest.mock import Mock

# 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...
```

Example Implementation

```
# function to be tested
def add_two_numbers(a, random_number_getter_function):
    return a + random_number_getter_function()
```

```
# 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
```

Point out: - The use of the `Mock()` class to create a new `object` - The use of the `return_value` method to specify the return value of the mock when it is called

---

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
```

Exercise [code along] - 1 - refactor

- Try to write unit-test for `get_price` of the `Rectangle` class. Use unittest's `Mock` class in your tests.

Exercise [code along] - 2 - refactor

- Try to do DI and then write unit-tests for this function (use `Mock` class)

```
import random


def random_list_generator(n):
    result = []
    for _ in range(n):
        result.append(random.randint(1, 10))
    return result
```

Exercise 2

- Try to rewrite the previous exercise's tests using unittest `Mock` class.

Learning Objectives Revisited

- To be able to explain what *Dependency Injection* is and why we do it.
- To gain experience *Mocking* in order to write well tested code.

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/class that invokes them.

Further Reading

- YouTube: Dependency Injection (in JavaScript but still a great watch)
- Dependency Injection
- unittest.mock