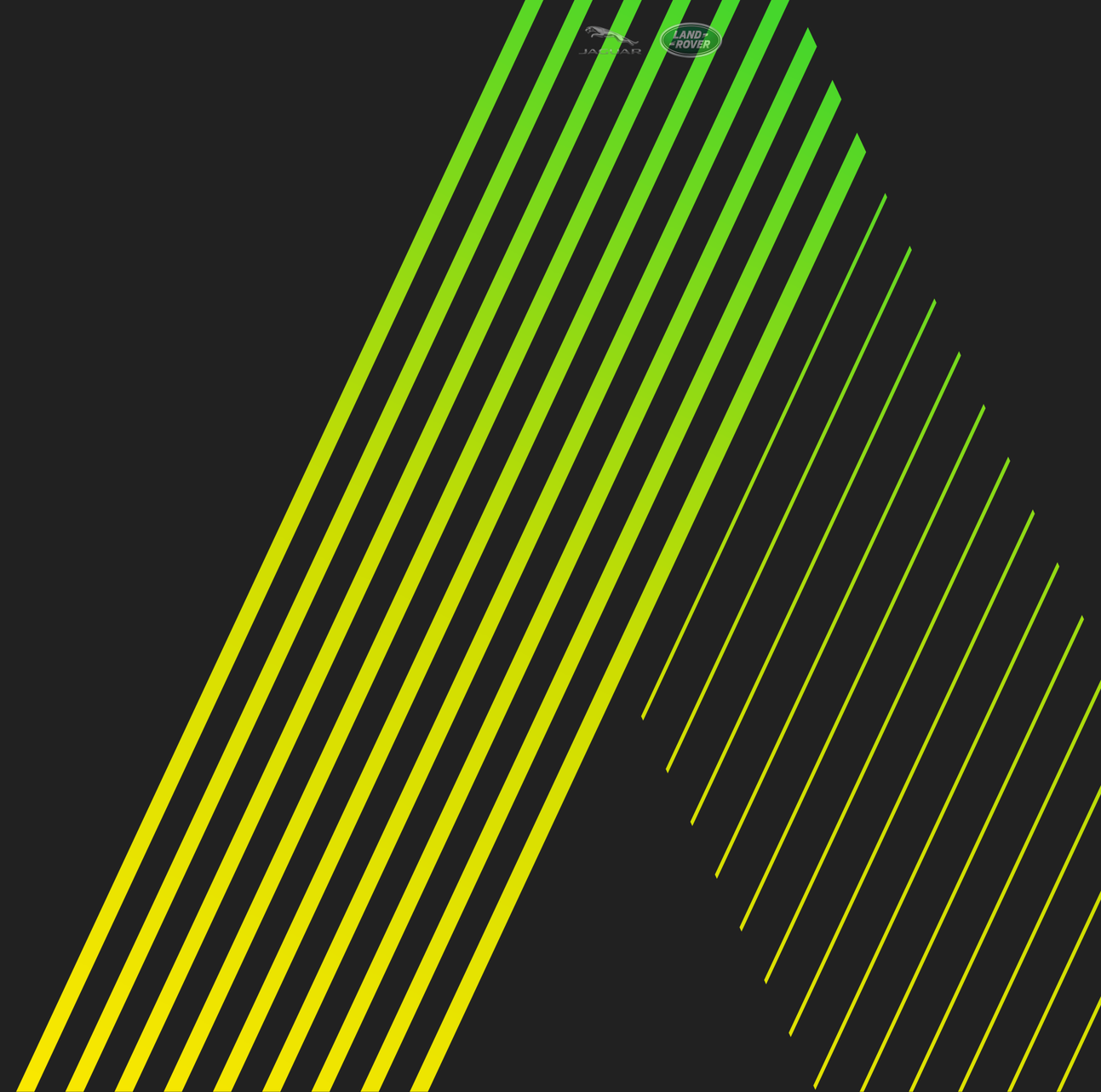
Unit Testing 2



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
```

Consider - Scenario 1

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

```python
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
```

Consider - Scenario 2

What about this one?!

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

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
```

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

```python
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))
```

The Mock Function

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

Exercise [code along] - 1

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

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.

```python
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
```

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)

```python
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