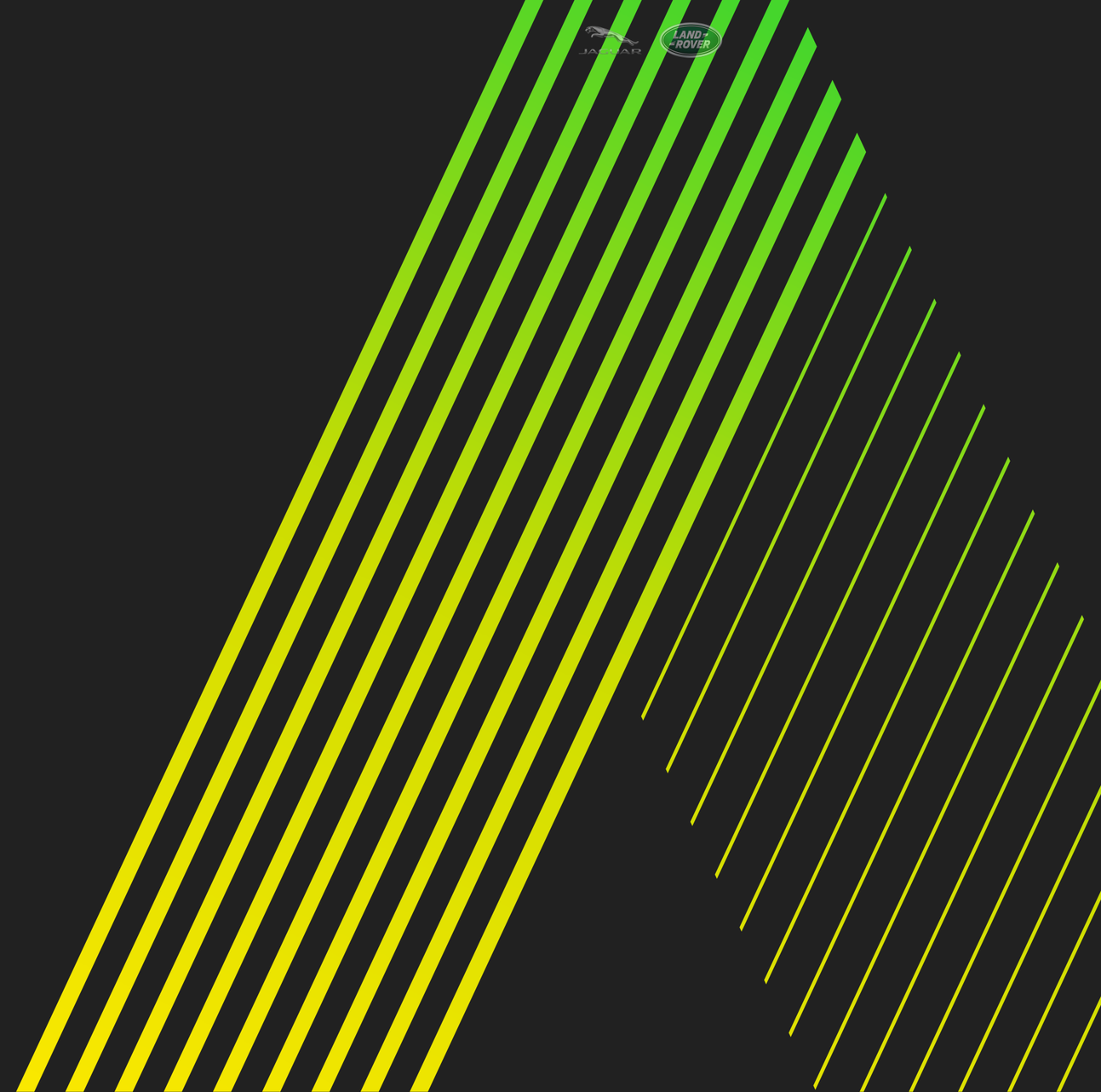
Unit Testing 3



Overview

- Intro to `Patch()`
- Exercise

Learning Objectives

- 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.

How do we do that then?

Can you do dependency injection?

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

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.

Example 1

How could I test these functions without any modification?

```python
import time


def api_call():
    time.sleep(3)
    return 9

def slow_function_with_DI(value, func_to_call):
    result = value * func_to_call()
    return result

def slow_function_without_DI(value):
    result = value * api_call()
    return result
```

Example 1 - answer

```python
from unittest.mock import Mock, patch
from app import slow_function_without_DI

@patch('app.api_call')
def test_slow_function_without_DI(mock_api_call):
    # assemble
    mock_api_call.return_value = 500
    expected = 100 * 500

    # act
    actual = slow_function_without_DI(100)

    # assert
    assert expected == actual
```

Example 2

How could I test this function?

```
# Without DI
def hello_to_you(name):
    print(f"Hello, {name}!") # Dependency
```

Example 2 - answer

```python
from unittest.mock import patch

@patch("builtins.print")
def test_prints_hello_to_you(mock_print):
    # Assemble
    my_name = "John"
    expected = "Hello, John!"

    # Act
    hello_to_you(my_name)

    # Assert
    mock_print.assert_called_with(expected) # Passes
```

Example 3

How could I test tis function?

```
# Without DI
def greeting():
    name = input("what is your name? ") # Dependency
    return 'Nice to meet you, ' + name
```

Example 3 - answer

```python
from unittest.mock import patch


@patch("builtins.input")
def test_greeting(mock_input):
    # Arrange
    mock_input.return_value = 'Jessica'
    expected = 'Nice to meet you, Jessica'

    # Act
    actual = greeting()

    # Assert
    assert actual == expected
    assert mock_input.call_count == 1
```

Exercise [code-along]

Write a test to verify functionality of the following function for this scenario:

Example scenario:

```
price_list = [10, 20]
user input = 50
expected_result = [10, 20, 50]
```

```
def add_price(price_list): # No DI
    value = int(input("Please enter a number: ")) # Dependency
    price_list.append(value)
    return price_list
```

Example 5

What if we have two dependencies?!

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

Example 5 - answer

```python
from unittest.mock import patch


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

    # Act
    print_name()

    # Assert
    mock_print.assert_called_with(expected) # 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

Instructor to distribute exercise.

Exercise [code-along]

Write a function named `add_multiple_products` with the following requirements:

- It should accept two arguments, `products_list` (list) and `number_of_new_products` (int)
- Based on the value of `number_of_new_products` it should ask user to write product names in terminal one by one.
- For any of the user input products, if the product is not already available in the `products_list` it should add it to the end of list, otherwise it should skip that product.
- At the end, the function should return the updated `products_list`.

Write unit-tests to verify the functionality of your code.

Learning Objectives Revisited

- 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