School of Tech
part of accenture

Unit Testing 3

School of Tech
part of accenture

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.

School of Tech
part of accenture

How do we do that then?

Can you do dependency injection?

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

What if we don't use Dependency Injection

There are common scenarios where mocking or refactoring for dependency injection aren't always possible, for example:

- 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`

School of Tech
part of accenture

`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()` for us
- In order to use it we have to *decorate* our test with `patch()`
- The mocks are then available to configure the behaviour, to use for spying, and making assertions

School of Tech
part of accenture

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
```

School of Tech
part of accenture

Example 1 - Solution

```
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?

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

School of Tech
part of accenture

Example 2 - answer

```python
from unittest.mock import patch

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

    # Act
    hello_to_you(my_name)

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

School of Tech
part of accenture

Example 3

How could I test this function?

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

School of Tech
part of accenture

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
```

School of Tech
part of accenture

Exercise (code-along)

> Let's all write a test to verify functionality of the following function for this scenario:

Example data:

```
    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
```

School of Tech
part of accenture

Solution (Part 1)

```python
from unittest.mock import patch
from price_lister import add_price

@patch('builtins.input')
def test_add_price(mock_input):
    price_list = [10, 20]
    mock_input.return_value = 50
    expected = [10, 20, 50]

    result = add_price(price_list)

    assert expected == result
```

School of Tech
part of accenture

Solution (Part 2)

> Bonus functionality - "ignore the input if it is already available in price list"

First we write a unit test that considers this case:

```python
from unittest.mock import patch
from price_lister import add_price

@patch('builtins.input')
def test_add_price_already_in_list(mock_input):
    price_list = [10, 20]
    mock_input.return_value = 20
    expected = [10, 20]

    result = add_price(price_list)

    assert expected == result
```

Solution (Part 2)

> Bonus functionality - "ignore the input if it is already available in price list"

Then we can update our `add_price` function to pass the new test:

```
def add_price(price_list):
    value = int(input("Please enter a number: "))
    if value not in price_list:
        price_list.append(value)
    return price_list
```

Emoji Check:

Do you feel you understand what patching is? Say so if not!

1. 😢 Haven't a clue, please help!
2. 🙁 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

School of Tech
part of accenture

Example 5 - Multiple Dependencies

What if we have two dependencies?!

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

School of Tech
part of accenture

Example 5 - Solution

```
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
```

Patching multiple dependencies

Note the order of the `@patch()` decorators compared to the order of the arguments in the test function.

- The first argument for the mock returned from `@patch()` belongs to the patch directly above the function definition (in this example that's `@patch("builtins.print")`).
- It then works up the way, so the second `@patch()` decorator above the function will provide the second argument (in this case `@patch("builtins.input")` is 2 lines above the function definition, so `mock_input` is the second argument).

Configuring our Patch

For example:

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

So:

The path to the function to be patched is provided as a string to `@patch()`.

The path should match the level which would be needed if the function were to be imported into the file, e.g. `from src.module import method` -> `@patch("src.module.method")`.

Exercise prep

> Instructor to give out the zip file of exercises for `unit-testing-3`
>
> Everyone please unzip the file

Exercise time

> From the zip, you should have a file `exercises/unit-testing-3-exercises.md`
>
> Let's all do the exercises included in this file

Emoji Check:

How did the exercises go? Is patching making more sense now?

1. 😢 Haven't a clue, please help!
2. 🙁 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

School of Tech
part of accenture

Patching Gotcha - Where to Patch

Imagine you have a project with the following structure (two modules):

```python
# module_a.py
def add_nums(a, b):
    return a + b
```

and

```python
# module_b.py
from module_a import add_nums

def do_maths(a, b):
    print(add_nums(a, b))
```

If you wanted to unit test `do_maths` from `module_b.py`, you would need to patch `add_nums`. But...

School of Tech
part of accenture

Patching Gotcha - Where to Patch

It would be tempting to write the patch for where `add_nums` was created, e.g.:

```python
@patch('module_a.add_nums') # Incorrect
def test_do_maths():
    ...
```

However, as `module_b` has imported `add_nums`, you have to patch `add_nums` where it is used, **NOT** where it is defined, e.g.:

```python
@patch('module_b.add_nums') # Correct
def test_do_maths():
    ...
```

See the documentation for some more examples of this and similar situations.

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.

Overview - recap

- Intro to `patch()`
- Exercise

Learning Objectives - recap

- Know how to use an alternative approach to Dependency Injection

School of Tech
part of accenture

Further Reading

- Dependency Injection
- Handbook: unittest.mock

School of Tech
part of accenture

Emoji Check:

On a high level, do you think you understand the main concepts of this session? Say so if not!

1. 😢 Haven't a clue, please help!
2. 🙁 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively