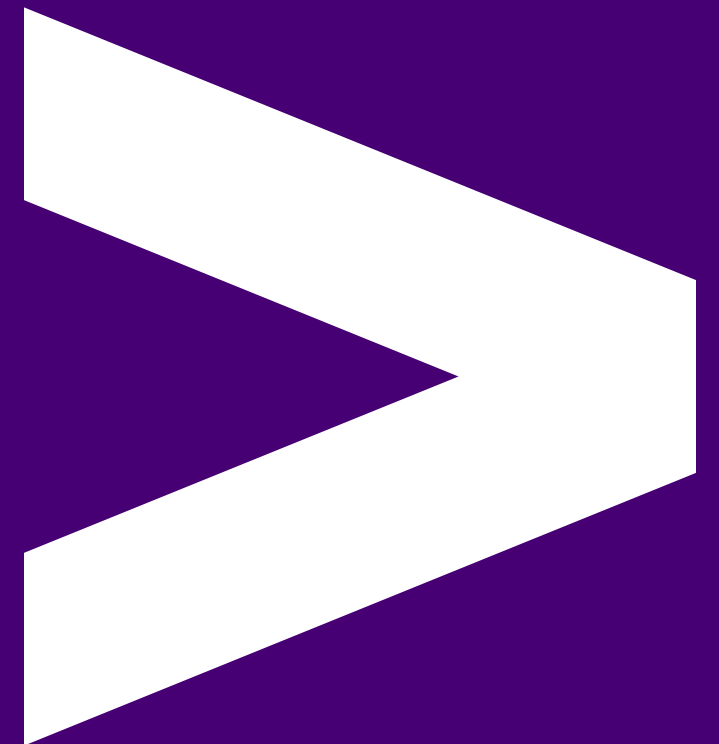


Unit Testing 2



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 - Unit testing

In the previous session we unit-tested a method like this:

```
def add_two_numbers(num1, num2)  
    return num1 + num2
```

Consider - Scenario 1

Let's add more complexity to our function, adding a dependency on something else:

```
from random import randint # generates random numbers

def add_number_and_random(num1):
    return num1 + randint(1, 10)
```

...this adds a dependency to another function.

What problems might we face testing `add_number_and_random`?

Consider - Scenario 2

What about this one?!

```
from random import randint # generates random numbers
def get_random_number():
    return randint(1, 10)

def add_number_and_random(num1)
    return num1 + get_random_number()
```

What problems might we face testing `get_random_number` or `add_number_and_random`?

Consider - Scenario 2

- When we run `add_number_and_random` is also runs `get_random_number`
- And what if `get_random_number` had it's own dependencies?
- We don't necessarily know (without looking) what `get_random_number` itself is going to depend on
- If we leave it as it is, our test will also indirectly test the dependencies, and dependencies of dependencies, which is *Integration Testing*
- We want to test *only* the `add_number_and_random` 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?

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? If our code can be changed to a useful shape, then:

- **Yes:** Mock it (**Today's topic**)

But if not:

- **No:** *Patch* it, then *Mock* it (For which see Unit Testing 3)

Dependency Injection (DI)

By *injecting* the *dependency*, the caller of our function is responsible for providing the `get_random_number` logic:

```
from random import randint
def get_random_number():
    return randint(1, 10)


def add_number_and_random(num1, generator_function)
    return num1 + generator_function()

# to call it we would use:
print(add_number_and_random(123, get_random_number))
```

Which means that

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

Another example



Lets look at example using an external API to load some data.

Here's a function that calls out to an external service:

```
def get_todays_price_per_cake():  
    api_url = "https://www.random.org/integers/?num=1&min=5&ma  
    response = requests.get(api_url)  
    if response.status_code == 200:  
        return json.loads(response.content)[0]  
    else:  
        return None
```

Here's a function using it to calculate today's cost of a box of cakes, yummy:

```
def get_price_of_box(number_of_cakes):  
    price_per_cake = get_todays_price_per_cake() # Execute dep  
    return number_of_cakes * price_per_cake  
  
print(get_price_of_box(15))
```

With dependency injection

Lets reorganise that to use DI (dependency injection):

```
def get_price_of_box(number_of_cakes, price_getter):  
    price_per_cake = price_getter() # Execute dependency  
    return number_of_cakes * price_per_cake  
  
print(get_price_of_box(15, get_todays_price_per_cake))
```


The Mock Function

There are two common uses of the word *Mock*, and this can get a bit confusing:

- A Mock function you write yourself
 - These are also called "dummy" functions, which can help disambiguate
- Using the `Mock()` testing tool (more on this later)

The Mock or "Dummy" Function

We make a function (in our test file) that takes the place of the real dependency:

```
def mock_price_per_cake():  
    return 7  
  
# or just as common is  
def dummy_price_per_cake():  
    return 7
```

Emoji Check:

Do you feel you understand the need for dependency injection? 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

Code-Along Example 1 (10 mins)

Let's write a unit test for `get_price_of_box`, using a "dummy function" we write ourselves

Lets open VSCode and do this together

Example 1 - Solution

```
def dummy_price_per_cake():  
    return 8  
  
def test_get_price_of_box_happy_case():  
    number_cakes = 10  
    expected = 80  
  
    result = get_price_of_box(number_cakes, dummy_price_per_ca  
  
    assert expected == result, f'Expected {expected} but was {'
```

Code Along - Example 2

- Let's try to understand what the `random_list_generator` function does, then refactor it to use DI, 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
```

Example 2 - Solution (part 1/2)

First we update the `random_list_generator` function to include Dependency Injection:

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

Example 2 - Solution (part 2/2)

Then we can provide a mock random number generator for our unit test:

```
def mock_random_number_generator():  
    return 7  
  
def test_random_list_generator():  
    list_length = 3  
  
    expected = [7, 7, 7]  
  
    result = random_list_generator(list_length,  
                                   mock_random_number_generato  
  
    assert expected == result, f'Expected {expected} but was {'
```


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 prep

Instructor to give out the zip file of exercises for `unit-testing-2`

Everyone please unzip the file

Exercise time

From the zip, you should have a file `exercises/unit-testing-2-numbers-exercise.md`

Let's all do the exercises included in this file

If you finish early, have a look at the file `exercises/unit-testing-2-countries-exercise.md`

Emoji Check:

How did the exercises go? Is dependency injection 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

Improving our approach

Writing our own mock functions works, but we'd likely have to:

- Create individual mock functions for each test case
- Modify each one to return the desired result

Is there a better way?

What about tools from the testing Frameworks?

Mock()

- The `unittest` framework has a `Mock` function
- `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 automatically replaced with another `Mock()` object whenever our *unit* tries to access it.

Using the Mock utility

We import the utility from the `unittest` library - this comes with `pytest`:

```
from unittest.mock import Mock
```


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 = 123
print(mock_function()) # 123

# Mocking a Class / Object
mock_class = Mock()
mock_class.some_method.return_value = "Hello World!"
mock_class.some_other_method.return_value = True
# 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, f'Expected {expected} but was {'
```

Spying on our Mock

The test tools record the behaviour of our mocks (how they are called, or not) and it's parameters, which we can use later to make better assertions. We can thus "spy" on what happened in our code:

`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
```

Emoji Check:

Do you feel you understand the Mock function? 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

Example 1 - Refactor

- Try to write a unit-test for `get_price_of_box`. Use unittest's `Mock` class in your tests.

Example 1 - Refactor Solution

We no longer need to create a separate mock function.

```
from unittest.mock import Mock

def test_get_price_of_box_happy_case():
    number_cakes = 10
    mock_get_price = Mock() # Mock created as part of unit tes
    mock_get_price.return_value = 9

    expected = 90
    result = get_price_of_box(number_cakes, mock_get_price)

    assert expected == result, f'Expected {expected} but was {
    mock_get_price.assert_called() # Can spy on the mock
```

Example 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
```

Example 2 - Refactor Solution

First we update the `random_list_generator` function to include Dependency Injection again for the `random` module:

```
import random

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

To actually call this function, we could for example use:

```
random_list_generator(6, random)
```

Example 2 - Refactor Solution

Then we can provide a mock random module for our unit test:

```
from unittest.mock import Mock

def test_random_list_generator():
    list_size = 4

    mocked_random = Mock()
    # This time we add a return_value on the
    # randint method of our mock
    mocked_random.randint.return_value = 3

    expected = [3, 3, 3, 3]

    result = random_list_generator(list_size, mocked_random)

    assert result == expected
    mocked_random.randint.assert_called()
```

Exercise time (Optional extra)

Go back over your solutions from Unit Testing 1 (the previous session) and use `Mock()` instead.

Emoji Check:

How did the exercises go? Is Mock 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

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.
- **Spy**: examining how a Mock was used to check if our code did the right thing

Overview - recap

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

Learning Objectives - recap

- 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.

Further Reading

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

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