Advanced Python testing

Fixtures, mocks and patching

Tim Guicherd - ACIT2515

Fixtures

- In tests, it is common to create an object and run all the different methods to check their behaviours.
- Each test function will test a specific feature of the class.
- You may end up creating the same object over and over in all of your test functions.
- *Fixtures* are meant to solve this problem: they allow you to define objects that can be reused in different test functions.

```python
import pytest

@pytest.fixture
def bank():
    bank = Bank("BCIT Bank")
    return bank

def test_bank(bank):
    assert bank.name == "BCIT Bank"
```

Pytest fixtures

- Defined using the decorator `@pytest.fixture`
- They are regular functions
- The value returned is the "fixture" that you can use in your test function
- To make a test function use a fixture, use it as an argument

```python
import pytest

@pytest.fixture
def tim():
    return Customer("Tim", "A0001")

@pytest.fixture
def checking(tim):
    return CheckingAccount(tim)
```

> You can "nest" fixtures!

MOCKS AND PATCHING

The problem we are trying to solve

```
class BankAccount:
    [...]
    @property
    def balance(self):
        """ Return the balance of the account """
    [...]
class BankCustomer:
    def __init__(self):
        self._accounts = list()
    def add_account(self): [...]
    def total_balance(self):
        return sum([account.balance for account in self._accounts])
```

How to test `BankCustomer` without testing `BankAccount` at the same time?

Unit tests

- Unit tests should only test a specific class or function, and not the **dependencies** of that code.
- There are some elements that you do not need to isolate: you don't have to test if `super()` works, or if a `list` is really a list, for instance.

> There are different ways to make it happen. In Python, we generally use **mocks** or **patch** our functions / methods / classes.

Example: broken property in `BankAccount`

```
def test_bank_customer_total_balance():
    tim = BankCustomer("Tim")

    account = BankAccount("Test", 1000)
    tim.add_account(account)

    assert tim.total_balance == 1000
```

- Let's imagine there is a bug in the `balance` property of the BankAccount class.
- This test will fail, because of the bug in **another class**.
- We need to remove the dependency between `BankCustomer` and `BankAccount`.

Monkeypatching

- The `monkeypatch` fixture is available by default in Pytest.
- You can use it in your test functions to change the behaviour of certain objects or classes.
- Its use is very similar to `setattr` in Python.
- In our case, we want to **monkeypatch** the `balance` attribute for the BankAccount class.
- We are going to monkeypatch it with a standard integer:

```python
def test_bank_customer_total_balance(monkeypatch):
    tim = BankCustomer("Tim")

    account = BankAccount("Test", 1000)
    tim.add_account(account)

    monkeypatch.setattr(BankAccount, "balance", 1000)
    assert tim.total_balance == 1000
```

> This allows the test to pass, without fixing the code in a different Python module!
> Note that the `monkeypatch` is a fixture received as argument to the test function.

Mocks

- Monkey patching is very useful and efficient in simple cases.
- It allows you to change attributes on the fly to make your tests pass.
- Sometimes it is not possible to monkey patch specific attributes, and you need to replace your entire classes.
- Sometimes you may want to change the behaviour of "builtin" functions, such as `input` or `open`.
- Mocks are used in this case. They are pieces of code that replace the behaviour of another piece of code.
- The process of "injecting" mocks into the code is called patching.

Patching the `input` method

```
@patch("builtins.input", side_effect=["abc", "def"])
def test_example(mock_input):
    value1 = input()
    value2 = input()
    assert value1 == "abc"
    assert value2 == "def"
```

Patching with `patch`

- The code patches the builtin `input` method.
- It replaces it with a mock (available in the test function as `mock_input`, see arguments).
- The mock has a *side effect*
  - it returns "abc" the first time it is called
  - and returns "def" the second time it is called

> The `mock_input` replaces the `input` method, makes it non-interactive and predictable!

Patching the `open` function

- It is very common to patch the `open` method in Python.
- The tests should not depend on local files whose content you don't control.
- Unit tests will patch `open` to control the "content" on which the program operates.
- It can be complicated to create a mock object every time - especially because `open` is a standalone function, but can also be used as a context manager (`with open [...]`).
- Python comes with a preexisting mock called `mock_open` .
- Use it in combination with `patch` !

Python with `mock_open`

```
from unittest.mock import mock_open, patch

FILE_CONTENTS="line1\nline2\nline3\n"

@patch("builtins.open", new_callable=mock_open, read_data=FILE_CONTENTS)
def test_open(mock_file):
    with open("my_file.txt", "r") as fp:
        data = [line.strip() for line in fp.readlines()]

    assert data[0] == "line1"
    assert data[1] == "line2"
```