c) Utilisation de mocks

NB: Ce document s'inspire d'openclassroom et de la documentation officielle de pytest

Partons sur une utilisation un peu plus poussée des tests : l'utilisation de **monkeypatck/mock**

Commençons par une définition de « Mock », ça nous aidera à comprendre la suite :
« Mock », qui signifie « imitation » en français, **désigne des fonctions qui ont comme objectif d'imiter le comportement d'autres objets**.

Mise en contexte:

Imaginons qu'un script interroge un API et inscrit les résultats dans un fichier Json.

Plusieurs problèmes peuvent se poser si nous n'utilisons pas de « Mockers ». En effet, chaque lancement de ce script écrit dans le même fichier. Or nous ne voulons pas changer les données de ce fichier, fait pour héberger de vraies données, dans le but de faire nos tests. Normalement ce fichier ne doit se recharger que si l'utilisateur le veut et non si nous lançons nos tests. Par ailleurs, chaque appel http va allonger le temps d'exécution du test (qui est un temps précieux car on souhaiterait que les tests soient le plus rapide possible).

L'idéal serait donc de pouvoir tricher un peu en disant à Pytest : "Je veux que tu testes la fonction mais, plutôt que de lancer un appel 'pour de vrai', utilise plutôt ces données-là !".

Ceci s'appelle un **'mock'**.

De manière générale, les mocks sont utilisés pour imiter le comportement de librairies ou de modules que nous ne souhaitons pas tester.

Monkeypatch possibles

La fixture « monkeypatch » va nous aider à créer ces imitations.

- **Modification du comportement d'une fonction ou de la propriété d'une classe** (API call ou database connection)

monkeypatch.setattr(obj, name, value, raising=True)
monkeypatch.delattr(obj, name, raising=True)

- **Modification des valeurs d'un dictionnaire**

monkeypatch.setitem(mapping, name, value)
monkeypatch.delitem(obj, name, raising=True)

- **Modification des variables d'environnement**

monkeypatch.setenv(name, value, prepend=False)
monkeypatch.delenv(name, raising=True)
monkeypatch.chdir(path)

Exemple : On peut utiliser monkeypatch.setenv("PATH", value, prepend=os.pathsep) pour modifier $PATH, et monkeypatch.chdir pour changer le répertoire de travail courant.

- **Modification de sys.path**

monkeypatch.syspath_prepend(path)

NB: "raising" permet d'indiquer si les exceptions KeyError or AttributeError seront levés si "name" n'existe pas dans "obj"

Imitation d'une requête HTTP

1ère tentative d'explication

Imaginons qu'on ait l'app suivante qui va faire un request d'un API et nous retourne un fichier json

```
//app.py
#! /usr/bin/env python
import json
import urllib2

def get_agents():

    response = urllib2.urlopen("http://pplapi.com/batch/1/sample.json")
    agents = json.loads(response.read().decode("utf8"))

    return agents
```

Cette fonction nous donne le retour suivant :

```
<http.client.HTTPResponse object at 0x10636d3c8>
[
  {
    "age": 27,
    "agreeableness": -1.330478164740173,
    "conscientiousness": -0.13936508775709072,
    "country_name": "Belarus",
    "country_tld": "by",
    "date_of_birth": "1990-03-29",
    "extraversion": -0.7695016063711204,
    "id": 6864578729,
    "id_str": "bSI-dKD",
    "income": 14798,
    "internet": true,
    "language": "Belarusian",
    "latitude": 52.98282724600774,
    "longitude": 28.003545708677223,
    "neuroticism": 0.4034048869311393,
    "openness": 0.4024561283710326,
    "religion": "other",
    "sex": "Female"
  }
]
```

C'est là qu'il faut être attentif afin d'en comprendre tout l'intérêt !

Désormais, **nous souhaitons juste vérifier que get_agents retourne bien un json**.

Comme nous pouvons le voir, get_agents est décomposé de la manière suivante :

- response = urllib2.urlopen("http://pplapi.com/batch/1/sample.json") va charger le contenu de la page

- agents = json.loads(response.read().decode("utf8")) va le transformer en json

response fait appel à un module externe.. **pourquoi le tester car ce n'est pas moi qui l'ai développé** ?
Il faudra donc **imiter son comportement afin qu'on puisse tester la suite du code**.
Par ailleurs, **dans notre cas, nous n'avons pas besoin de connaître le contenu exact du json qui est réellement retourné** car nous avons juste besoin d'un **retour du même type que get_agents (une liste de json)**.

On pourra donc commencer le module de test par les lignes suivantes :

```
//test app.py
import app
import urllib2
from io import BytesIO
import json

def test_http_return():

    results = [{
            "age": 84,
            "agreeableness": 0.74
          }
        ]
```

Jusque là, ça va ! On veut voir si get_agents nous retourne bien une liste de json, peu importe son contenu.

Maintenant qu'on connaît le résultat, on veut tester que app.get_agents() nous retourne bien le contenu de la variable "results" qu'on a défini.

```
def test_http_return(monkeypatch):
    results = [{
            "age": 84,
            "agreeableness": 0.74
          }
        ]


    assert app.get_agents() == results
```

Ouai, mais là on va réellement faire appel à la fonction urllib2.urlopen("http://pplapi.com/batch/1/sample.json").. On a dit qu'on ne va pas le tester..
Ducoup on va comparer le contenu de "http://pplapi.com/batch/1/sample.json" au results que l'on a défini, ça ne va donc pas marcher !
C'est là qu'intervient le 'Mock' ! Imitons cette fonction !

```
def test_http_return(monkeypatch):
    results = [{
            "age": 84,
            "agreeableness": 0.74
          }
```

```
        ]

    def mockreturn(request):
        return results

    monkeypatch.setattr(urllib2, 'urlopen', mockreturn)

    assert app.get_agents() == results
```

Qu'est-ce qui se passe là dedans ???

- mockreturn(obj) prendra en paramètre l'objet sur lequel nous allons appeler la méthode
- monkeypatch.setattr(urllib2, 'urlopen', mockreturn), écrit autrement on a : urllib2.urlopen = mockreturn. On change la valeur renvoyée par urllib2.urlopen.

Si on exécute le test, qui ne fonctionnera pas.. on aura :

```
___________________________________________________________ test_http_return
___________________________________________________________

monkeypatch = <_pytest.monkeypatch.MonkeyPatch object at 0x1065e7f50>

    def test_http_return(monkeypatch):
        results = [{
                "age": 84,
                "agreeableness": 0.74
              }
            ]
        def mockreturn(request):
            return results
        monkeypatch.setattr(urllib2, 'urlopen', mockreturn)
>       assert app.get_agents() == results

test_app.py:19:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

    def get_agents():
        response = urllib2.urlopen("http://pplapi.com/batch/1/sample.json")
>       agents = json.loads(response.read().decode("utf8"))
E       AttributeError: 'list' object has no attribute 'read'

app.py:8: AttributeError
====================================== 1 failed in 0.14 seconds
======================
```

L'erreur nous indique que : 'list' object has no attribute read.
Cela nous indique que l'imitation de app.get_agents a bien fonction (ah bon ??).
Lors du passage dans la fonction app.agents, **monkeypatch a redéfini le résultat de la variable response** (car nous avons demandé à ce que urllib2.urlopen renvoie requests qui est une liste).

En effet, urllib.requests ne retourne pas vraiment une liste. Il faut l'adapter un peu.

Utilisons la classe BytesIO pour créer un objet sur lequel nous pourrons appeler la méthode read(). Enfin, puisque la requête renvoie du json encodé et que nous le décodons, il nous faut imiter une réponse en json et encodée :

```
def mockreturn(url):
        return BytesIO(json.dumps(results).encode())
```

On réexécute et tout fonctionne !!

2ème tentative d'explication

Une 2ème tentative d'explication sur un exemple quasi-identique.
Cette fois-ci, on part d'un code déjà tout fait.

```
import requests

def get_json(url):
    """Takes a URL, and returns the JSON."""
    r = requests.get(url)
    return r.json()
```

```
//test_app.py
import requests
import app

class MockResponse:

    @staticmethod
    def json():
        return {"mock_key": "mock_response"}


def test_get_json(monkeypatch):

    def mock_get(*args, **kwargs):
        return MockResponse()

    monkeypatch.setattr(requests, "get", mock_get)

    result = app.get_json("https://fakeurl")
    assert result["mock_key"] == "mock_response"
```

Réfléchissons ensemble sur le fonctionnement du test:

Dans test_app.py:

- on a défini une fonction mock_get qui retourne la classe MockResponse
- MockResponse contient une **méthode json** qui retourne {“mock_key”: “mock_response”}
- par la suite, on rencontre monkeypatch qui à priori cherchera à attributer mock_get à requests.get

- un appel à app.get_json est effectué

On passe à app.py :

- on retrouve r = requests.get(url) **(ça match avec le monkeypatch !)**
- ainsi **r est redéfini** et vaut MockResponse() (aucun réel appel à url n'est effectuer)
- on retourne r.json() –> MockResponse.json() –> {“mock_key”: “mock_response”}

On repasse à test_app.py: :

- result vaut {“mock_key”: “mock_response”}
- l'assertion est vérifiée

Un mock en fixtures ? Et oui c'est possible..

Si on reprend les fixtures qu'on a défini dans le chapitre précédent, on peut écrire :

```
import pytest
import requests
import app

@pytest.fixture
def mock_response(monkeypatch):
    """Requests.get() mocked to return {'mock_key':'mock_response'}."""

    def mock_get(*args, **kwargs):
        return MockResponse()

    monkeypatch.setattr(requests, "get", mock_get)

# on utilise la fixture en argument, et pas monkeypatch
def test_get_json(mock_response):
    result = app.get_json("https://fakeurl")
    assert result["mock_key"] == "mock_response"
```

Imitation d'un dictionnaire

Si le principe du mocking a été compris pour l'exemple précédent, il le sera sans aucun doute pour celui-ci.

```
#app.py
DEFAULT_CONFIG = {"user": "user1", "database": "db1"}

def create_connection_string(config=None):
    config = config or DEFAULT_CONFIG
    return f"User Id={config['user']}; Location={config['database']};"
```

Dans le code ci-dessus, on a le dictionnaire DEFAULT_CONFIG qui sert à se connecter à une base de donnée.
Cependant, pour effectuer des tests, on ne voudra pas se connecter à la base opérationnelle.

Vous me voyez venir de loin.. On pourra utiliser un monkeypatching pour modifier ses valeurs.

```
//test_app.py
import app

def test_connection(monkeypatch):

    monkeypatch.setitem(app.DEFAULT_CONFIG, "user", "test_user")
    monkeypatch.setitem(app.DEFAULT_CONFIG, "database", "test_db")

    # résultat attendu
    expected = "User Id=test_user; Location=test_db;"

    result = app.create_connection_string()
    assert result == expected
```

Commençons par test_app.py :

- monkeypatch.setitem(app.DEFAULT_CONFIG, “user”, “test_user”) : on va chercher à attribuer la valeur test_user à app.DEFAULT_CONFIG.user
- monkeypatch.setitem(app.DEFAULT_CONFIG, “database”, “test_db”) : on va chercher à attribuer la valeur test_db à app.DEFAULT_CONFIG.database
- on connait le résultat attendu : “User Id=test_user; Location=test_db;”
- result = app.create_connection_string()

On passe à app.py :

- config = DEFAULT_CONFIG
- comme par hasard DEFAULT_CONFIG contient les clefs “user” et “database”, le monkeypatch fait son boulot et modifie les valeurs
- config vaut donc {“user”: “test_user”, “database”: “test_db”}

On repasse à test_app.py:

- le test fonctionne car l'assertion est vérifiée

C'était beaucoup plus simple !
Et d'après vous.. est-ce que ce test va fonctionner ?

```
//test_app.py
import app

def test_connection(monkeypatch):

    result = app.create_connection_string()

    monkeypatch.setitem(app.DEFAULT_CONFIG, "user", "test_user")
    monkeypatch.setitem(app.DEFAULT_CONFIG, "database", "test_db")

    # résultat attendu
    expected = "User Id=test_user; Location=test_db;"
```

```
    assert result == expected
```

Réponse

Eh ben non, **les monkeypatch ne regardent pas en arrière**.
Les objets qui seront modifiés par les monkeypatch doivent absolument être définis après eux !

Imitation de l'écriture dans un fichier

Lorsqu'on veut tester une fonction qui va écrire dans un fichier spécifique, on n'a pas très envie de polluer nos répertoires avec des fichiers de test.
L'idéal serait d'avoir un fichier temporaire qui se détruira tout seul.
J'espère que vous me voyez venir, sinon vous avez raté quelques notions ! On va utiliser “tmpdir”.

```
//app.py
import argparse
import urllib2


def parse_args(args=None):
    parser = argparse.ArgumentParser(description="Ecriture dans un fichier")
    parser.add_argument("-d", "--dest", help="Destination file. If absent,
will print to stdout")
    return parser.parse_args(args)

def main(command_line_arguments=None):
    args = parse_args(command_line_arguments)

    result = "Hello world"

    with open(args.dest, 'w') as out_f:
            out_f.write(result)
```

Dans la fonction ci-dessus, on va créer un fichier.txt dans lequel on va insérer du contenu (ie. “Hello World !”).
Le chemin du répertoire du fichier sera précisé via l'argument “-d” de la fonction main()

On voudra donc tester si le contenu du fichier est bien “Hello World”, mais on le fera **via un fichier temporaire**.

```
//test_app.py
import app

def test_http_return(tmpdir, monkeypatch):

    p = tmpdir.mkdir("program").join("test.txt")

    app.main(["--dest", str(p)])

    expect = "Hello World"
    with open(str(p),'r') as f:
```

```
        text = f.readlines()[0].strip()

    assert expect == text
```

En détail :

- On va ajouter “tmpdir” en argument de la fonction de test (ce directory se détruira tout seul).
- On va définir un chemin temporaire “p”
- On exécute la fonction d'écriture du texte dans le fichier ayant pour directory p
- On l'ouvre et on vérifie que le texte obtenu est le même que celui qui est attendu