b) Utilisation de fixtures

Les « fixtures » sont des fonctions attachées aux tests et exécutées avant ces mêmes fonctions de test.
Celles-ci sont utilisées afin d'injecter des données en arguments de tests telles que celles obtenues après une connexion à une base de données, des URLs, des fichiers .txt ou autres (chaîne de caractères, JSON, listes..).
On peut indiquer à pytest qu'une fonction est une fixture en la décorant avec « @pytest.fixture ».

Utilisation du décorateur

Faisons quelques tests simples, toujours avec la class Deg créé précedemment :

```
//test_fixture.py
import pytest

class Deg:
    def __init__(self, val):
        self.val = val

    def __eq__(self, other):
        return self.val == other.val

@pytest.fixture //on a indiqué que boiling_temp est une fixture
def boiling_temp(): // retourne l'objet Deg qui a pour valeur 100
        return Deg(100)

@pytest.fixture //list of temps est aussi une fixture
def add_temp_in_list(boiling_temp): //on utilise la fixture boiling_temp en
argument : insère en argument la valeur retournée par boiling_temp
        return [Deg(0),boiling_temp]


def test_list(boiling_temp,add_temp_in_list): //on réutilise les 2 fixtures
définies précédemment en argument
        assert boiling_temp in add_temp_in_list
```

L'exécution de pytest ne donnera pas d'erreurs car Deg(100) est bien présent dans la liste retournée par add_temp_in_lis.

En quelque sorte, si nous n'avions pas mis de fixtures, il aurait fallu écrire à la main :

```
def boiling_temp():
        return Deg(100)

def add_temp_in_list(boiling_temp):
        return [Deg(0),boiling_temp]
```

```
def test_list(boiling_temp,add_temp_in_list):
        assert boiling_temp in add_temp_in_list

obj_boiling_temp = boiling_temp()
list_of_temps = add_temp_in_list(obj_boiling_temp)
test_list(obj_boiling_temp, list_of_temps)
```

Auto-use fixtures

Parfois, on peut avoir besoin de définir des **fixtures pour lesquels on sait d'avance qu'on en aura besoin pour tous les tests**.
L'avantage est qu'on aura donc pas à l'appeler et l'exécuter une nouvelle fois à chaque test.
Ainsi, **une fixture peut s'exécuter automatiquement sans qu'on ait besoin de l'appeler grâce au décorateur @pytest.fixture(autouse=true)**

```
//test_autouse.py
import pytest


@pytest.fixture
def first_entry():
    return "a"


@pytest.fixture
def order(first_entry):
    return []


@pytest.fixture(autouse=True) //s'execute sans qu'on le demande
def append_first(order, first_entry):
    return order.append(first_entry)


def test_string_only(order, first_entry):
    assert order == [first_entry]
```

On peut voir dans le code ci-dessus que la fonction append_first n'est jamais appelé par une autre fonction.
Si l'on exécute le test sur la fonction test_string_only, celui-ci devrait donc échouer car order devrait être une liste vide alors que [first_entry] contiendrait « a ».
Cependant, ça ne sera pas le cas et le test fonctionnera sans aucun soucis.
En effet, le décorateur @pytest.fixture(autouse=true) va exécuter automatiquement append_first, ainsi order va bien valoir [a] conformément à ce qui est attendu.

Scope : Définir la portée des fixtures

Dans le cas où on utiliserait plusieurs fois des fixtures extrêmement couteuses ou longues à charger, il serait dommage de le faire à chaque fois qu'on en a besoin.

Ainsi, **il est possible de ne le charger qu'une fois dans un fichier de test et d'étendre sa porter à un(e) ou plusieurs classes, modules, packages, sessions.**

Pour cela, **@pytest.fixture** pourra prendre un attribut scope pouvant valoir : **function, class, module, package, session**

Scope : function

Il s'agit du scope par défaut, sans même qu'on ait à le définir.
Comme son nom l'indique, la portée de la fixture sera limitée à la fonction dans laquelle elle est utilisée.

Exemple

```
import pytest
import json
import logging
@pytest.fixture()
def read_config():
    with open("app.json") as f:
        config = json.load(f)
    logging.info("Read config")
    return config

def test1(read_config):
    logging.info("Test function 1")
    assert read_config == {}
def test2(read_config):
    logging.info("Test function 2")
    assert read_config == {}
# test/test_code.py::test1
# -------- live log setup --------
# INFO     root:test_code.py:10 Read config
# -------- live log call --------
# INFO     root:test_code.py:15 Test function 1
# PASSED   [ 50%]
# test/test_code.py::test2
# -------- live log setup --------
# INFO     root:test_code.py:10 Read config
# -------- live log call --------
# INFO     root:test_code.py:20 Test function 2
# PASSED   [ 100%]
```

Comme on peut le voir dans les logs ci-dessus, le fichier json est rechargé à chaque test.. c'est assez embêtant, surtout si celui-ci est lourd.

Scope : class

La fixture est exécutée une fois pour toute une classe entière.

Exemple

```
@pytest.fixture(scope="class")
def dummy_data(request):
    request.cls.num1 = 10
    request.cls.num2 = 20
    logging.info("Execute fixture")

@pytest.mark.usefixtures("dummy_data")
class TestCalculatorClass:
    def test_distance(self):
        logging.info("Test distance function")
        assert distance(self.num1, self.num2) == 10

    def test_sum_of_square(self):
        logging.info("Test sum of square function")
        assert sum_of_square(self.num1, self.num2) == 500
# test/test_code.py::TestCalculatorClass::test_distance
# ------- live log setup -------
# INFO     root:test_code.py:59 Execute fixture
# ------- live log call -------
# INFO     root:test_code.py:65 Test distance function
# PASSED
[ 50%]
# test/test_code.py::TestCalculatorClass::test_sum_of_square
# ------- live log call -------
# INFO     root:test_code.py:69 Test sum of square function
# PASSED

# source code
def distance(num1, num2):
    return abs(num1 - num2)

def sum_of_square(num1, num2):
    return num1 ** 2 + num2 ** 2
```

Au cas où il y aurait plusieurs fixtures avec une même scope et qu'en plus de cela on aurait besoin que d'une seule d'entre elles dans une certaine classe, le marqueur **@pytest.mark.usefixtures("fixture-name")** permet de le préciser.

Cette fois-ci, imaginons que l'on utilise cette fixture pour une base de données avec un scope="class" :

```
@pytest.fixture(scope="class")
def prepare_db(request):
    # pseudo code
    connection = db.create_connection()
    request.cls.connection = connection
    connection = db.close()
```

Cela ne fonctionnerait pas car la base de données sera fermée avant même que les tests ne s'effectuent.
C'est à ce moment là que le mot-clef **« yield »** intervient.
Ainsi, tout ce qui se situe **avant « yield » est exécuté avant les tests, et tout ce qui se situe après « yield » est exécuté à la toute fin**.

On réécrira le script précédent de la manière suivante :

```
@pytest.fixture(scope="class")
def prepare_db(request):
    # pseudo code
    connection = db.create_connection()
    request.cls.connection = connection
    yield // On ne veut pas fermer la base de donnée qu'après l'exécution de
tous les tests
    connection = db.close()
```

Scope : module and package

Le 1er exemple (cf scope :function) chargeait le fichier à chaque test.
L'intérêt d'utiliser un scope=module ou package nous permettra de ne **le charger qu'une seule fois**.

Exemple

```
@pytest.fixture(scope="module")
def read_config():
    with open("app.json") as f:
        config = json.load(f)
        logging.info("Read config")
    return config

def test1(read_config):
    logging.info("Test function 1")
    assert read_config == {}

def test2(read_config):
    logging.info("Test function 2")
    assert read_config == {}

# test/test_code.py::test1
# ----------- live log setup -----------
# INFO     root:test_code.py:88 Read config
# ----------- live log call -----------
# INFO     root:test_code.py:93 Test function 1
# PASSED
[ 75%]
# test/test_code.py::test2
# ----------- live log call -----------
# INFO     root:test_code.py:98 Test function 2
# PASSED
```

En regardant les logs, on peut voir que « Read config » n'apparaît qu'une fois au lieu de 2 dans le tout 1er exemple.

Scope = session

La fixture sera exécutée une seule fois pour toute une session **(ie : à chaque fois qu'on lancera la commande pytest)**.
On l'utilisera principalement lorsqu'on chargera une **grosse série de données (via une base de données ou un fichier lourd par exemple)**.

On notera qu'au lieu d'utiliser un scope=session, on pourra utiliser la fixture dans conftest.py.
Pour rappel, toutes les fixtures définies dans conftest.py seront disponibles dans tous les tests du répertoire sans avoir à être importées.

Exemple

```
# test/conftest.py
@pytest.fixture(scope="session")
def read_config():
    with open("app.json") as f:
        config = json.load(f)
        logging.info("Read config")
    return config

# test/test_code1.py
def test1(read_config):
    logging.info("Test function 1")
    assert read_config == {}

def test2(read_config):
    logging.info("Test function 2")
    assert read_config == {}

# test/test_code2.py
def test3(read_config):
    logging.info("Test function 3")
    assert read_config == {}

def test4(read_config):
    logging.info("Test function 4")
    assert read_config == {}
# test/test_code1.py::test1
# -------- live log setup --------
# INFO     root:conftest.py:10 Read config
# -------- live log call --------
# INFO     root:test_code1.py:93 Test function 1
# PASSED
```

```
[ 50%]
# test/test_code1.py::test2
# -------- live log call --------
# INFO     root:test_code1.py:98 Test function 2
# PASSED
[ 66%]
# test/test_code2.py::test3
# -------- live log call --------
# INFO     root:test_code2.py:5 Test function 3
# PASSED
[ 83%]
# test/test_code2.py::test4
# -------- live log call --------
# INFO     root:test_code2.py:10 Test function 4
# PASSED
```

Ordre d’exécution des scopes

Si on utilise des fixtures ayant chacune un scope différent, ils s’exécuteront dans un ordre bien précis :

- **session > package > module > class > function**