a) Notions de base

Pourquoi tester ?

Lorsqu'un module est constitué de nombreuses fonctionnalités ou que nous sommes plusieurs à contribuer à son développement, il devient fortement nécessaire d'y inclure une procédure de tests. Généralement, il peut être simple de déceler manuellement des problèmes lorsque la console le retourne explicitement ou que nous connaissons intuitivement le type de résultat attendu. Cependant, même si rien n'est retourné, cela ne veut pas dire que le programme fonctionne à la perfection.

Dans le cas de la boîte à outils de l'équipe, imaginons tout simplement qu'une procédure de post-process a été développée et a pour objectif de calculer la somme d'un paramètre sur un intervalle de validité prédéfini (ex : entre 1-7h). En exécutant le programme, on verra d'un côté qu'aucune erreur ne sera retournée, donc pas de soucis du côté de python et d'un autre côté que la valeur retournée a bien l'ordre de grandeur d'un cumul sur 6h. À priori, tout va bien.. mais si l'on creuse un peu plus, on se rendra compte que le cumul est en réalité pris entre 0 et 6h (et pas 1-7h).

Et puis même si nous avons tout testé manuellement, cette procédure peut aussi être malencontreusement modifiée quelques années plus tard; surtout lorsque celle-ci est développée à plusieurs : un signe + modifié en un signe – est vite arrivé !

Ainsi, lorsque qu'un outil est développé à plusieurs et devient conséquent , il est difficile de tout tester manuellement à la perfection car cela devient vite chronophage voire même impossible. C'est donc à ce moment là qu'interviennent les procédures de tests.

Quels utilitaires

Différentes procédures de tests existent pour python :

- Nose
- Doctest
- Unittest
- Pytest

Après plusieurs recherches, il s'avère que **pytest est celui qui est le plus largement utilisé** de par sa facilité d'utilisation, de compréhension et le large éventail de possibilités qu'il offre. Pour n'en citer que quelques-unes :

- **Il n'utilise pas de boilerplate** (je viens d'apprendre ce mot).

En gros, au lieu d'écrire :

```
Self.assertEqual(1+1,2) //unittest.py
```

On va écrire :

```
Assert 1 + 1 == 2 //pytest.py
```

Certes, on diminue en précision dans l'écriture de l'erreurs mais pytest fera tout seul l'effort de comprendre l'erreur et de pondre un rapport d'erreurs tout aussi exhaustif et précis qu'avec l'écriture en unittest.

- **Rapports d'échecs intelligents**

```
def test_eq_dict(self):
>       assert {'a': 0, 'b': 1, 'c': 0} == {'a': 0, 'b': 2, 'd': 0}
E       assert {'a': 0, 'b': 1, 'c': 0} == {'a': 0, 'b': 2, 'd': 0}
E       Omitting 1 identical items, use -v to show
E         Differing items:
E         {'b': 1} != {'b': 2}
E         Left contains more items:
E         {'c': 0}
E         Right contains more items:E         {'d': 0}
E         Use -v to get the full diff
```

- **Utilisation de décorateur/fixtures** (on verra ça après !)
- **Présence de nombreux plugins** (dont code coverage)
- **Et surtout.. Pytest est activement maintenu**

Procédures de base avec pytest

Tout d'abord, **on regroupera tous les tests dans un même dossier**.
Par ailleurs, pytest attend à ce que tous les **tests soient contenus dans des fichiers de type test_*.py ou *_test.py**.
Toutes les procédures de test devront commencer par test_ (ex : def test_conversion_deg_to_kelvin())

Définition d'un test

Tentons de créer un test sur un programme qui va convertir les kelvin en degrés.

```
// script.py

def kelvToDeg(x) :
    deg = x – 273.15
    return deg
```

```
//test_script.py

from script import *

def test_kelv_to_deg() :
    assert kelvToDeg(273.15) == 0 // verifions que la conversion de 273.15
kelvin est bien 0 degré
```

Maintenant, exécutons ce premier test en **lançant la commande suivante dans le répertoire contenant les test_*.py** :

```
pytest (ou python -m pytest si ça ne fonctionne pas)
```

On aura donc le retour suivant indiquant tout simplement que le test a fonctionné :

```
============================= test session starts ======================
platform darwin -- Python 2.7.16, pytest-4.6.11, py-1.10.0, pluggy-0.13.1
rootdir: /Users/kerylclain/Desktop/test
collected 1 item

test_script.py . [100%]  // Pourcentage de couverture du test

========================== 1 passed in 0.01 seconds ======================
```

Maintenant, imaginons que j'insère malencontreusement une erreur dans mon code de conversion après une longue semaine de boulot.
(je ne vais pas vous faire l'affront de préciser où est l'erreur) :

```
//script.py
def kelvToDeg(x) :
    deg = x – 272.15
    return deg
```

Le test retournera l'erreur suivante :

```
=================================================== test session starts
====================================
platform darwin -- Python 2.7.16, pytest-4.6.11, py-1.10.0, pluggy-0.13.1
rootdir: /Users/kerylclain/Desktop/test
collected 1 item

test_script.py F [100%]

==================== FAILURES =======================================
_____________________________________ test_kelv_to_deg
_____________________________________

    def test_kelv_to_deg():
>    assert kelvToDeg(273.15) == 0
E    assert 1.0 == 0 //1.0 est la valeur calculée, 0 est la valeur attendue
E     +  where 1.0 = kelvToDeg(273.15)

test_script.py:4: AssertionError
============================================================ 1 failed in
0.06 seconds
====================================================================
```

La console va rougir et nous dire que le résultat obtenu n'est pas correct (càd, différent que celui attendu dans l'assert).

À retenir :

- On rassemblera tous les tests dans un même dossier
- Dans ce dossier, chaque fichier de tests commencera par test_ ou finira par _test.py
- Dans chaque fichier, les fonctions devront commencer par test_ ou finir par _test
- Dans chaque fonction, le test que l'on va effectuer est défini **assert**

Error raises

Par la suite, imaginons qu'une chaîne de caractère soit utilisée en entrée du programme de conversion. (Il va de soi que le programme ne fonctionnera pas.)
C'est pour cela qu'il peut donc être intéressant de vérifier qu'une erreur soit bien levée si le format utilisé n'est pas conforme au format attendu.

```
//test_script.py

def test_raise_error_string_argument():
        with pytest.raises(Exception): //On vérifie qu'une exception est
bien levée..
                assert kelvToDeg('dix-huit') //lorsqu'on tente de convertir
une chaîne de caractère
```

En lançant le programme de tests, nous allons vérifier que l'ajout d'une chaîne de caractère en entrée lève bien une exception.
Ainsi, l'exécution nous donnera le résultat suivant :

```
============================= test session starts
=====================================
platform darwin -- Python 2.7.16, pytest-4.6.11, py-1.10.0, pluggy-0.13.1
rootdir: /Users/kerylclain/Desktop/test
collected 2 items

test_script.py ..
[100%]

====================== 2 passed in 0.07 seconds
==================================
```

En effet, **le test fonctionne car une erreur est bien levée** par le programme lorsqu'on tente de convertir la chaîne de caractère en degré Celsius.
Cependant, dire qu'une exception est levée n'est pas très informatif car il existe un large panel d'erreurs (ex: AssertionError, AttributeError...).
Il peut donc être intéressant de la spécifier de la manière suivante:

```
//test_script.py

def test_raise_error_string_argument():
        with pytest.raises(Exception) as e: //on stocke l'exception dans la
```

```
variable e
                assert kelvToDeg('dix-huit') //test qui doit lever une
erreur
        assert str(e.value) == "Must be int or float" //on vérifie que
l'erreur levée par script.py est conforme à celle qui est attendue "Must be
int or float"
```

On vérifie donc que le message d'erreur « Must be int or float » soit affiché lorsqu'une chaîne de caractère est utilisée en entrée.
Pour cela, il faut que **script.py** retourne ce message d'exception :

```
//Script.py

def kelvToDeg(kelv):
        if isinstance(kelv,(int,float)):
                deg = kelv - 273.15
                return deg
        else://On lève une erreur si l'argument n'est ni un entier, ni un
flottant
                raise Exception("Must be int or float")
```

En exécutant le test, tout fonctionne car l'exception affiche bien « Must be int or float ».
Le test échouera dans le cas où cette exception spécifique n'est pas levée (c'est à dire, une exception qui a un message autre que "Must be int or float").

À retenir :

- On peut vérifier qu'une erreur sera bien levée avec "with pytest.raises(ErreurQuiSeraLeve): assert fonctionTest(argumentQuiDoitLeverUneErreur)"
- Pour tester une erreur spécifique, on peut vérifier que le message retourné par l'erreur est bien conforme à ce qui est attendu avec : assert str(e.value) == "Message de l'erreur attendu"

Error raises : redéfinir les retours

Si le contenu des erreurs levées n'est pas très parlant, on peut les redéfinir comme bon nous semble. Ainsi, il est possible d'ajouter nos propres explications à l'échec d'une assertion en ajoutant la fonction **pytest_assertrepr_compare() dans un fichier conftest.py**.

Lorsqu'un fichier conftest.py (**convention d'écriture**) est inséré dans un répertoire, **son contenu devient disponible pour tous les sous-répertoires du répertoire dans lequel il est présent (des imports ne sont pas nécessaires**). Cela sera pratique lorsque nous définirons des fixtures qui pourront être utilisés par plusieurs tests. (cf https://docs.pytest.org/en/stable/fixture.html#conftest-py)
À titre d'exemple, définissons le test suivant :

```
//test_newassert.py:

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

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


def test_compare():
    f1 = Deg(1) //on instancie les objets Deg
    f2 = Deg(2)
    assert f1 == f2 //on ne teste pas les objets mais les valeurs avec
lesquelles elles ont été initialisées
```

En lançant un test, on obtient le résultat suivant :

```
_____________________________________________ test_compare
_____________________________________________________________________________

    def test_compare():
        f1 = Deg(1)
        f2 = Deg(2)
>       assert f1 == f2
E       assert <test_newassert.Deg instance at 0x10b4e7248> ==
<test_newassert.Deg instance at 0x10b4e7320>
```

L'exception levée n'est pas très claire n'est-ce pas..
On va donc venir **la redéfinir dans un fichier conftest.py** (convention d'écriture) situé dans le même dossier que le test ou dans un des dossiers parents.

```
//conftest.py
from test_newassert import Deg

def pytest_assertrepr_compare(op, left, right): //en argument op="operation
de l'assert", left et right sont les valeurs situées respectivement à gauche
et à droite de "op"
    if isinstance(left, Deg) and isinstance(right, Deg) and op == "==":
        return [
            "Comparing Degrees instances:",
            "   vals: {} != {}".format(left.val, right.val), //retour voulu
en cas d'échec
        ]
```

Dorénavant, le retour sera plus compréhensible :

```
_____________________________________ test_compare
_____________________________________

    def test_compare():
        f1 = Deg(1)
        f2 = Deg(2)
>       assert f1 == f2
E       assert Comparing Degrees instances:
E            vals: 1 != 2 //Retour défini dans conftest.py

test_newassert.py:12: AssertionError
============================== 1 failed, 2 passed in 0.05 seconds
=======================================
```

À retenir :

- On peut redéfinir le message d'erreur retourné dans la console à partir d'un fichier conftest.py (nomenclature du fichier à respecter)
- Le contenu de ce fichier conftest sera disponible dans tous les fichiers du répertoire dans lequel il est situé et les sous-répertoires
- Des fonctions pré-existantes peuvent être utilisées pour redéfinir les erreurs issues d'asserts (exemple : pytest_assertrepr_compare)

Regroupement de tests

Les fonctions de tests peuvent aussi être organisées par class.
Cette structure trouvera vite son utilité car plus le programme est conséquent, plus le nombre de tests le sera aussi.
Tout comme pour les fichiers ou fonctions qui doivent avoir test_ comme suffixe ou préfixe, **les classes doivent aussi respecter une convention d'écriture : commencer par Test**.

```
//script test_classes.py //préfixe test_ pour le nom de fichier python

class TestClass: //la classe commence par Test
    def test_one(self): //préfixe test_ pour la fonction
        x = "this"
        assert "h" in x

    def test_two(self):
        x = "hello"
        assert hasattr(x, "check")
```

L'exécution du test de ce script donnera aussi un retour un peu plus structuré :

```
================================== FAILURES ===========================================
_____________________________________________ TestClass.test_two ______________________________________________________

self = <test_class.TestClass instance at 0x106b188c0>

    def test_two(self):
        x = "hello"
>       assert hasattr(x, "check")
E       AssertionError: assert False
E        +  where False = hasattr('hello', 'check')

test_class.py:8: AssertionError
```

Le regroupement des tests par Class doit être utilisé avec parcimonie car le principe d'un test unitaire est d'être isolé au maximum.
Ainsi, la définition de plusieurs tests dans une même Class peut être contre-productif.

tmpdir

```
#test_tmpdir.py
def test_needsfiles(tmpdir):
    print(tmpdir)
    assert 0
```

En exécutant le script précédent, on aura le retour suivant :

```
================================ FAILURES ================================
_____________________________ test_needsfiles ______________________________

tmpdir = local('PYTEST_TMPDIR/test_needsfiles0')

    def test_needsfiles(tmpdir):
        print(tmpdir)
>       assert 0
E       assert 0

test_tmpdir.py:3: AssertionError
--------------------------- Captured stdout call ---------------------------
PYTEST_TMPDIR/test_needsfiles0
======================== short test summary info =========================
```

On peut voir que le retour nous fournit un tmpdir.
Mais où est-ce que celui-ci est défini ?
pytest possède plusieurs fixtures prédéfinies, dont on peut retrouver la liste ici :
https://docs.pytest.org/en/stable/builtin.html

Parmi ces fixtures, on retrouvera le tmpdir utilisé en argument de la fonction def test_needsfiles().
Celui-ci va créer un répertoire temporaire unique à chaque fois qu'il est appelé.

Exemple d'utilisation :

```
#test_tmpdir.py
def test_needsfiles(tmpdir):
    p = tmpdir.mkdir("sub").join("hello.txt") //on créé sub/hello.txt dans
le dossier temporaire qui a été créé
    p.write("content") //on y insère du contenu
    assert p.read() == "content" //on vérifie que celui-ci possède bien le
contenu défini
```

À retenir :

- Pour avoir des informations plus poussées sur ces dossiers temporaires, c'est par ici
- Le répertoire par défaut peut être redéfini grâce à la commande : “pytest –basetemp=mydir”