TP1

Elements de complexité et programmation par tests (niveau 1).

Les tests

Nous devons être certains que toutes les méthodes, fonctions ou modules que nous créons soient corrects. On écrira donc des tests pour être moralement sûrs que nos programmes fonctionnent (la plupart du temps une preuve de code est illusoire).

Pour éviter de retaper tous ces tests à chaque modification du code (ce qui arrive souvent lorsque un algorithme ou une application est utilisée longtemps) ou à chaque découverte de bug, ils sont conservés dans un fichier à part. Ceci nous permettra d'exécuter ces tests à loisir (c'est à dire très souvent) et d'être sûrs que **tous** les tests seront exécutés. Ces tests sont dit unitaires et sont essentiels dans toutes les pratiques courantes de code.

Environnement de tests avec pyCharm

De nombreux environnements de tests existent pour pycharm, nous allons utiliser py.test.

Premier exemple

Créez un nouveau projet avec pycharm que l'on pourra appeler `essai_tests`, puis ajoutez-y un fichier que vous nommerez `aide_mathematiques.py`. Ce fichier contiendra le code suivant :

```
def double(entier):
  return 2 * entier
```

Pour tester ce code, j'imagine que si les deux conditions suivantes sont remplies :

- double(0) vaut 0,
- double (21) vaut 42

Ma méthode sera exacte.

On utilise le mot clé assert pour créer notre fonction de test.

 Les fonctions de tests doivent toutes commencer par `test_`

Ajouter la méthode ci-après à votre fichier :

```
def test_double():
  assert double(0) == 0
```

```
  assert double(21) == 42
```

et executez là :

```
test_double()
```

Si tout s'est passé comme prévu, il ne s'est rien passé. Normal, l'`assert` était vérifié. Changez un des `assert` de la fonction `test_double` pour que le résultat soit faux (par exemple `assert double(0) == 7`). Le programme doit maintenant s'arrêter sur une exception. Chez moi, j'obtiens ça :

```
Traceback (most recent call last):
  File
"/Users/francois/Documents/pycharm/essai_tests/aide_mathematiques.py", line
10, in <module>
    test_double()
  File
"/Users/francois/Documents/pycharm/essai_tests/aide_mathematiques.py", line
6, in test_double
    assert double(0) == 7
AssertionError
```

Ainsi, si tout se passe bien, nos tests sont passés, si le programme s'arrête sur une exception de type `AssertionError`, nos tests ne correspondent pas à la réalité. Nous sommes en face d'un bug (qu'il faut corriger).

Séparer code et tests

Placez la fonction de test (et son exécution) dans un fichier que vous nommerez `test_aide_mathematiques.py`.

Faites en sorte qu'il s'exécute sans problème (attention aux `import`).

On séparera toujours les tests du code. Tout fichier de test commence par `test_`.

Utilisation de l'environnement de test

Nous allons demander à l'environnement py.test d'exécuter nos tests. Il nous donnera plus d'informations sur les tests réussis ou échoués (une application normale contient des centaines de tests).

Commencez par supprimer l'exécution de `test_double` dans le fichier `test_aide_mathematiques.py`.

Un fichier de test ne doit contenir que des fonctions.

Puis nous allons demander à pycharm d'exécuter `test_aide_mathematiques.py` à l'aide de notre environnement de test. Pour cela, suivez les instructions de la partie Ajouter un environnement d'exécution et créez une configuration *pyhton test* > *py.test*. Ici, les paramètres dont nous aurons besoin sont :

- le champ `name`, qui donne un nom à notre contexte. Par exemple `mes tests`
- le champ `target`, qui spécifie quel script utiliser. Cliquez tout à droite de ce champ sur un petit bouton avec **…** puis choisissez le fichier `test_aide_mathematiques.py`

Une fois ceci configuré, cliquez sur le bouton OK.

Un nouvel environnement de test est créé dans l'onglet `run`. Exécutez le. Vous devriez voir une nouvelle fenêtre en bas de l'écran pycharm apparaître et vos tests s'exécuter. Si tout s'est bien passé, une barre verte doit apparaître.

Pour finir cette partie :

- séparez votre fonction de tests en 2 fonctions (chaque fonction de test ne doit contenir qu'une chose à tester, donc a priori qu'un seul `assert`,
- exécutez votre nouvel environnement
- ajoutez une fonction de test qui plante. Exécutez votre environnement de test. Voyez la barre rouge. Supprimez ce test non valide.

Les tests en ligne de commande

La bibliothèque py.test peut directement s'exécuter depuis le terminal. En supposant que votre fichier de test s'appelle `test_aide_mathematiques.py` et que vous vous trouviez dans le bon répertoire, la commande : `python3 -m pytest test_aide_mathematiques.py` va exécuter vos tests, comme vous le feriez depuis yCharm.

Calcul de Puissance

On cherche à calculer x^y

Itératif et récursif naïf

Codez un algorithme itératif et un algorithme récursif naïf permettant de calculer la puissance de deux entiers et leurs tests associés.

Exponentiation rapide

Coder l'algorithme d'exponentiation rapide et ses tests.

Calcul de complexité

Pour mesurer ce temps on pourra utiliser la méthode process_time du module time de python (si votre python3 est vieux, utilisez la méthode clock de time).

```
import time

temps_depart = time.process_time()
#ce que l'on veut mesurer
delta_temps = time.process_time() - temps_depart
```

Vérifiez que :

- le temps pris par l'algorithme itératif augmente suivant la valeur de y,
- le temps mis par l'algorithme itératif et rapide ne dépend pas de x,
- le rapport entre le temps mis pour résoudre une exponentiation avec l'algorithme rapide et celui mis avec l'algorithme naïf tend vers 0.

Mesurer précisément le temps mis pour exécuter un algorithme est compliqué. Les oscillations sont normales car le système, l'ide et même python peuvent faire des choses en parallèle. La mesure de temps utilisée n'est donc pas rigoureusement proportionnelle à la complexité de l'algorithme mais en est une bonne approximation.

Affichage de courbes

Utilisez le code ci-dessous pour afficher une courbe avec matplotlib où l'abcisse est y et l'ordonnée le temps mis pour calculer x^y.

```
import matplotlib.pyplot
from math import log


coordonnees_abcisses = range(2, 101)

x_fois_2 = []
x_carre = []
x_log_x = []

for x in coordonnees_abcisses:
    x_fois_2.append(x * 2)
    x_carre.append(x * x)
    x_log_x.append(x * log(x))

matplotlib.pyplot.ylabel("axe des ordonnees")
matplotlib.pyplot.xlabel("axe des abcisses")
```

```
matplotlib.pyplot.plot(coordonnees_abcisses, x_fois_2, color="#ff0000")
matplotlib.pyplot.plot(coordonnees_abcisses, x_carre, color="#00ff00")
matplotlib.pyplot.plot(coordonnees_abcisses, x_log_x, color="#0000ff")

matplotlib.pyplot.show()
```

Supperposez les courbes pour les 3 algorithmes. Conclusions ?