Aix*Marseille université
Initiative d'excellence
Année 2018-19

Diplôme Inter-Universitaire
Enseigner l'informatique au lycée

Semaine 2 : TP 3

Tests

1 Tutoriel utilisant Pycharm

1.1 premiers tests

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

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

1.1.1 Un test sans bibliothèque

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.

Attention : Les fonctions de tests doivent toutes commencer par `test_`.

Ajoutez la méthode ci-après à votre fichier :

```
def test_double():
  assert double(0) == 0
  assert double(21) == 42
```

et exécutez-la en fin de programme : `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/mathematiques.py", line 10, in <module>
    test_double()
  File "/Users/francois/Documents/pycharm/essai_tests/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).

1.1.2 Utilisation de l'environnement de test avec Pycharm

Nous allons demander à l'environnement py.test d'exécuter nos tests. Si ce n'est pas encore fait, Il faut installer le module `pytest` (`pip3 install pytest` ou via pycharm).

Il nous donnera plus d'informations sur les tests réussis ou échoués (une application normale contient des centaines de tests). Puis nous allons demander à Pycharm d'exécuter `mathematiques.py` à l'aide de notre environnement de test.

Pour cela, créez 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 `mathematiques.py`

Une fois ceci configuré, cliquez sur le bouton `OK`.

Un nouvel environnement de tests est créé dans le menu `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.

1.1.3 Séparer code et tests

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

Dans le module d'exécution de test, vous pouvez maintenant donnez le répertoire de votre projet comme source de tests. Pytest prendra automatiquement tous les fichiers qui commencent par `test_` comme fichiers de tests et à l'interieurs de ceux ci les fonctions qui commencent par `test_` comme des tests. Ceci vous permet d'exécuter en un click tous les tests de votre projet répartis en plusieurs fichiers de tests.

1.1.4 Les tests en ligne de commande

La bibliothèque `http://pytest.org` peut directement s'exécuter depuis le terminal. En supposant que votre fichier de test s'appelle `test_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 PyCharm.

2 Utilisation des tests

2.1 Analyse d'algorithme

Exemple du tri par sélection

L'algorithme fonctionne ainsi : à l'étape i l'algorithme place le ième plus petit élément en position $i-1$ du tableau

Que devrions nous tester ? Cas général des tris :
— tableau simple à un élément : normalmenet les algorithmes ne font rien
— tableau déjà trié de petite taille : les algorithmes *travaillent* mais ne doivent rien faire
— tableau trié à l'envers : parfois donne les cas limites
— tableau non trié

Question 1. En utilisant le code de l'algorithme ci-dessous, créez les 4 tests proposés. Vérifiez que vos tests sont bien ok en ajoutant un test qui plante, puis en modifiant l'algorithme pour qu'il ne trie plus.

```
def selection(tab):
    for i in range(len(tab) - 1):
        min_index = i
        for j in range(i + 1, len(tab)):
```

```
            if tab[j] < tab[min_index]:
                min_index = j

        tab[i], tab[min_index] = tab[min_index], tab[i]
```

Quelques règles :

— un tri doit être reproductible. On n'utilise donc pas le hasard dans les tests (le tableau ne doit pas être mélangé par shuffle par exemple). Sinon un bug peut apparaitre de façon aléatoire.

— pas trop de tests. Mais si on est pas sur, autant le rajouter. On pourra toujours l'enlever plus tard. Ici, est ce que tester des tableau avec des égalités à un sens ?

2.2 Preuve du programme

Exemple sur le code de césar. Les tests permettent :

— aux élèves de chercher à comprendre un bout de code : on donne le code d'une fonction et on demande aux élèves d'écrire les tests pour comprendre le fonctionnement.

— aux enseignants de vérifier que le code produit par les élèves marche juste en exécutant les tests (code de césar)

2.2.1 Compréhension du code : exemple de la suppression d'accent

Il est parfois utile se supprimer tous les accent d'un texte pour avoir moins de lettres différentes.

On utilise pur cela un *truc* unicode, la normalisation *'NFD'*. En unicode, les caractères accentués peuvent s'écrire sous deux formes : soit la glyphe `é` soit une combinaison de glyphes : la glyphe `e` suivie de la glyphe d'accent `'`.

Dans un texte en français, la lettre de base est une lettre présente dans le code ascii, mais pas l'accent. En transformant alors un texte français normalisé en NFD en ascii on supprime les caractères qu'il ne connait pas, donc tous les accents.

Exemple de la transformation :

```
import unicodedata

s = "éçà ü ?"
print(s)
s2 = unicodedata.normalize('NFD', s)
print(s2)  # pas de différence, c'est toujours de l'utf-8 par encodé différemment
s3 = s2.encode('ascii', 'ignore')
print(s3)  # c'est du bytes maintenant
s_sans_accent = s3.decode('utf-8')
print(s_sans_accent)  # de l'utf-8 sans accent
```

Question 2. Testez le bout de code ci-dessus. Créez une fonction (dans `transformations.py`) que vous nommerez `sans_accent` qui prend un texte en paramètre et le rend sans accent. Testez la fonction `sans_accent` en vérifiant qu'une chaine avec accent est bien transformée en une chaine sans accent. Créez une fonction (dans `transformations.py`) que vous nommerez `majuscules_sans_accent` qui prend un texte en paramètre et le rend sans accent et en majuscule (méthdoe `upper` des listes). Testez la fonction `majuscules_sans_accent`.

2.2.2 Production de fonction testés : exemple du code de César

Cette façon de coder des messages très ancienne, César l'aurait utilisée, repose sur une *substitution mono-alphabétique* particulière, c'est à dire remplacer une lettre de l'alphabet par une autre.

Le principe en est simple, chaque lettre de l'alphabet possède une valeur de 1 à 26 (A vaut 1, B vaut 2, ..., Z vaut 26), que l'on code en lui rajoutant une constante modulo 26.

On a alors coutume de considérer que la clé permettant de coder le texte est la lettre correspondant au codage de 'A'. Ainsi, l'ajout de la constante 4 correspond à la clé 'E', comme le montre la table suivante :

			clé E																					
initiale :	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
codée :	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

Le texte `"LE VENDREDI C'EST ALGORITHMIE"` devient alors `"PI ZIRHVIHM G'IWX EPKSVMXLQMI"` avec `"E"` comme clé (les caractères qui ne sont pas des majuscules sont ignorées dans le codage).

Question 3. Dans `cesar.py`, créez une fonction que vous appellerez `code_cesar` qui prend une lettre en majuscule en paramètre (le code) et rend un dictionnaire dont les clés sont les lettres en majuscules et la valeur l'encodage de cette lettre selon le code donné en paramètre. Testez votre méthode. Elle doit rendre `{'A': 'E', 'B': 'F', 'C': 'G', 'D': 'H', 'E': 'I', 'F': 'J', 'G': 'K', 'H': 'L', 'I': 'M', 'J': 'N', 'K': 'O', 'L': 'P', 'M': 'Q', 'N': 'R', 'O': 'S', 'P': 'T', 'Q': 'U', 'R': 'V', 'S': 'W', 'T': 'X', 'U': 'Y', 'V': 'Z', 'W': 'A', 'X': 'B', 'Y': 'C', 'Z': 'D'}` pour un paramètre valant `'E'`.

2.3 Refactoring

Une fois qu'une fonction est crée et testée, il est aisé de la modifier sans la casser. Ceci est particulièrement utile pour séparer un algorithme compliqué en plusieurs parties plus simple.

En cours, cela permet aux élèves d'avoir une compréhension plus fine de l'algorithme – souvent décrit en un seul tenant dans les livre et sur wikipédia – en le découpant en parties fonctionnelles.

Exemple du tri par sélection. Ce tri effectue 2 sous-tâches pour fonctionner : recherche du minimum et échange de deux valeurs.

Nous allons extraire ces deux sous-fonctions. Comme notre méthode de tri est testée, on saura tout de suite si cela rate ou pas.

Question 4. En vérifiant régulièrement, en exécutant vos tests, que tout fonctionne encore :

```
* Créez une méthode nommée `échange` qui à partir d'un tableau et de deux indices, les échange.
* Testez-la.
* Remplacez l'échange dans le tri par sélection par la nouvelle fonction.
```

Question 5. De même :

```
* Créez une méthode nommée `min_indice_tableau` qui à partir d'un tableau et d'un indice, rend l'inc
* Testez-la.
* Remplacez l'échange dans le tri par sélection par la nouvelle fonction.
```

2.4 Calcul de puissances

Question 6. Proposez des tests pour l'algorithme du calcul de la puissance :

```
def puissance(nombre, exposant):
    resultat = 1
    compteur = exposant
    while compteur > 0:
        resultat *= nombre
        compteur -= 1
    return resultat
```

Question 7. Adaptez le code précédent pour implémenter une exponentiation rapide. Procédez par étapes : chaque étape doit être simple et accompagnée d'un test si nécessaire.

2.5 Création par les tests

Partie la plus difficile mais également la plus utile et gratifiante une fois qu'on l'a comprise et qu'on l'utilise. On peut créer un algorithme ou un programme compliqué juste avec des tests. En ajoutant petit à petit de la complexité dans l'algorithme sans le casser (puisque les tests passent). L'exemple que l'on va utiliser est simple, mais cela fonctionne admirablement bine avec des problèmes bien plus compliqué dont, souvent, on a pas la bonne réponse immédiatement.

Question 8. Utilisez la programmation par les tests pour réaliser l'algorithme glouton de rendu de monnaie vu en première semaine.

L'algorithme doit prendre en entrée deux paramètres :

— le système de pièce sous forme d'une liste

— la valeur à rendre

Comme à chaque fois, commencez petit et ajouter des cas simples (et leurs tests) qui complexifieront votre code.