Jeu de tests et assertions

1. Test d'une fonction

Lorsque l'on exécute une fonction, elle peut fonctionner comme prévu, mais elle peut aussi "planter", ou bien boucler indéfiniment, ou bien ne pas produire le résultat escompté. Et même si elle fonctionne comme prévu, rien ne garantit que ce soit toujours le cas quelles que soient les données d'entrée qu'on lui fournit. **Tester une fonction consiste à vérifier qu'elle fonctionne comme prévu dans un maximum de situations possibles**.
Pour cela, il faut écrire un ou plusieurs **jeux de tests** et s'assurer que les tests passent avec succès.

2. Jeux de tests

Deux des méthodes de base qui consiste à tester la fonction sont l'utilisation :

- Du **`print`** dans le programme
- Du **shell**.

```
test_et_assertion.py
1  def minutes(h, m):
2      """renvoie en minutes le temps donné en heure (h) et minutes(m)"""
3      nb_minutes = h*60 + m
4      return nb_minutes
5
6  print(minutes(1, 20))
7  print(minutes(0, 30))
Console
>>> %Run test_et_assertion.py
80
30
```

Test avec un print

```
test_et_assertion.py
1  def minutes(h, m):
2      """renvoie en minutes le temps donné en heure (h) et minutes(m)"""
3      nb_minutes = h*60 + m
4      return nb_minutes
Console
>>> %Run test_et_assertion.py
>>> minutes(1, 20)
80
>>> minutes(0, 30)
30
```

Test dans le shell

L'ensemble des tests réalisé par ces deux méthodes s'appellent un **jeu de tests**.

Ces méthodes ne laissent pas de traces des tests réalisés (si on conserve l'instruction **`print`**, on remplit de message le **shell** lors de l'exécution du programme).
Conserver une trace des tests permet d'aider à la compréhension de la fonction au travers des différents cas testés.

3. Assertions

3.1. assert

Le mot-clé **`assert`** permet de définir un test d'assertion. Sa syntaxe est la suivante :

`assert condition, texte alternatif`

- Si la condition est vraie, il ne se passe rien et le programme continue de se dérouler normalement.
- Si la condition est fausse, une erreur de type **`'AssertionError'`** est déclenchée et le texte alternatif détaille cette erreur.

```
test_et_assertion.py
1  def minutes(h, m):
2      """renvoie en minutes le temps donné en heure (h) et minutes(m)"""
3      nb_minutes = h*60 + m
4      return nb_minutes
5
6  assert minutes(1, 20) == 80, 'erreur dans la fonction'
7  assert minutes(0, 30) == 30, 'erreur dans la fonction'
Console
>>> %Run test_et_assertion.py
>>>
```

Fonction sans erreur

```
test_et_assertion.py
1  def minutes(h, m):
2      """renvoie en minutes le temps donné en heure (h) et minutes(m)"""
3      nb_minutes = h*30 + m
4      return nb_minutes
5
6  assert minutes(1, 20) == 80, 'erreur dans la fonction'
7  assert minutes(0, 30) == 30, 'erreur dans la fonction'
Console
>>> %Run test_et_assertion.py
Traceback (most recent call last):
  File "D:\NSI\Activité Term\1-Modularité\test et assertion.py", line 6, in <module>
    assert minutes(1, 20) == 80, 'erreur dans la fonction'
AssertionError: erreur dans la fonction
>>>
```

Fonction avec erreur

À chaque exécution du programme, les tests d'assertions vont être exécutés.

(cc) BY-NC-SA

Pour ne pas exécuter les tests d'assertions à chaque exécution du programme, Il est possible aussi de placer les tests d'assertions d'une fonction dans un fonction de test.
Cette fonction pourra alors être exécutée seulement lors de la phase de mise au point.

```
test_et_assertion.py
1  def minutes(h, m):
2      """renvoie en minutes le temps donné en heure (h) et minutes(m)"""
3      nb_minutes = h*60 + m
4      return nb_minutes
5
6
7  def test_minutes():
8      """test de la fonction minutes"""
9      assert minutes(1, 20) == 80, 'erreur dans la fonction'
10     assert minutes(0, 30) == 30, 'erreur dans la fonction'

Console
>>> %Run test_et_assertion.py
>>> test_minutes()
>>>
```

Fonction test_minutes

3.2. Préconditions

Il est souvent nécessaire de tester des **préconditions** sur les arguments d'une fonction, pour s'assurer que celle-ci fonctionnera bien selon le schéma voulu. Pour ceci, il est possible d'utiliser une assertion.

```
test_et_assertion.py
1  def minutes(h, m):
2      """renvoie en minutes le temps donné en heure (h) et minutes(m)"""
3      assert type(h) == int, 'h doit etre un chiffre entier'
4      assert h >= 0, 'h doit être positif ou nul'
5      assert type(m) == int and m >= 0, 'm doit être un entier positif ou nul'
6      nb_minutes = h*60 + m
7      return nb_minutes

Console
>>> %Run test_et_assertion.py
>>> minutes(1, -2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "D:\NSI\Activité Term\1-Modularité\test et assertion.py", line 5, in min
utes
    assert type(m) == int and m >= 0, 'm doit être un entier positif ou nul'
AssertionError: m doit être un entier positif ou nul
>>>
```

Essai avec un chiffre négatif pour les minutes

3.3. Test de la fonction appartient

Soit la fonction suivante :

```
def appartient(v, t):
    """renvoie True si l'entier v appartient au tableau d'entiers t"""
    for i in range(len(t)):
        if t[i] == v:
            return True
    return False
```

✍ Dans la fonction **appartient**, créer une assertion pour tester la précondition que **v** est entier.

✍ Créer une fonction **test_appartient** qui teste la fonction **appartient** dans les cas suivants :

- **t** est un tableau vide
- La valeur **v** se situant en début du tableau
- La valeur **v** se situant en fin de tableau
- La valeur **v** se situant en milieu de tableau
- La valeur **v** ne se situant pas dans le tableau

(cc) BY-NC-SA