TP 6
Algorithmes sur les listes

À partir de ce TP nous n'apprenons pas de concepts nouveaux du langage Python lui-même. Tout le cours nécessaire a été accumulé dans les TP et est résumé sur le cours distribué au début d'année.
On propose d'étudier un certain nombre de situations classiques. Pour certaines d'entre elles les fonctions existent déjà dans les bibliothèques de base de Python, ou s'obtiennent facilement à partir d'elles, mais **on s'interdit de les utiliser**. On travail donc essentiellement avec des boucles `for`, l'accès aux indices d'une liste, les conditions booléennes classiques.
Enfin ce TP vient avec un fichier pré-rempli à compléter. L'intérêt, en plus du gain de temps, est surtout que le fichier comprend de nombreux **tests** pour aider à décider si la fonction est bien correcte ou non. Dans les exercices qui vont suivre, il faut en effet être capable de tester la fonction dans des situations diverses et pas seulement avec l'exemple le plus simple. Regarder attentivement les valeurs avec lesquelles sont testés les programmes et observer les résultats tout en réfléchissant bien !

I Compter

Une situation de base est de *compter* les éléments d'une liste vérifiant une certaine propriété. Pour cela il est nécessaire de déclarer une variable appelée **compteur** (en effet...) initialisée à 0, et qui augmente de 1 à chaque fois. Le modèle de base est la fonction suivante qui compte le nombre de termes positifs d'une liste de nombres :

```
def compte_positifs(L):
    c = 0
    for i in range(len(L)):
        if L[i] >= 0:
            c = c + 1
    return c
```

Remarquons que l'alignement des blocs d'instructions est crucial. Il indique que l'instruction `c = c + 1` est exécutée uniquement quand la condition juste au dessus `L[i] >= 0` est vérifiée, et c'est donc cela qui va être compté. L'instruction `return c`, elle, est exécutée à la toute fin après avoir parcouru *toute* la liste.
Si aucun terme de la liste n'est positif, alors l'incrémentation de `c` n'a jamais lieu et à la fin `c` vaut 0 comme au début, ce qui est cohérent.

Exercice 1. Écrire une fonction `compte(L, x)` qui prend en argument une liste `L` et un nombre `x` et compte combien de fois `x` apparait dans la liste `L`.

Exercice 2. Écrire une fonction `compte_voyelles(s)` qui prend en argument une chaine de caractères `s` et compte le nombre de voyelles (lettres parmi `a`, `e`, `i`, `o`, `u`, `y`) dans `s`.

La situation pour sommer les termes d'une liste n'est pas très différente. Ici il n'y a plus de variable compteur mais une **variable accumulatrice**.

Exercice 3. Écrire une fonction `somme(L)` qui calcule la somme de tous les termes de la liste `L`.

II Tester

Une autre situatione courante est de vouloir vérifier si *tous* les termes d'une liste vérifient une certaine propriété. Par exemple on souhaite écrire une fonction prenant en argument une liste `L` et qui renvoie `True` si tous les éléments de `L` sont positifs, et `False` sinon.
Une seule fonction est correcte parmi les propositions ci-dessous.

```python
def tous_positifs_1(L):
    for i in range(len(L)):
        if L[i] >= 0:
            return True
        else:
            return False
```

```python
def tous_positifs_2(L):
    for i in range(len(L)):
        if L[i] >= 0:
            return True
    return False
```

```python
def tous_positifs_3(L):
    for i in range(len(L)):
        if L[i] < 0:
            return False
        else:
            return True
```

```python
def tous_positifs_4(L):
    for i in range(len(L)):
        if L[i] < 0:
            return False
    return True
```

Exercice 4. Laquelle des fonctions ci-dessus teste bien si tous les éléments de `L` sont positifs ? Observer le code, tester les exemples du fichier et **justifier précisément**.

Exercice 5. Écrire une fonction `binaire(m)` qui prend en argument une chaine de caractères `m`, et qui renvoie `True` si `m` est composée uniquement de caractères `"0"` ou `"1"`, et `False` sinon.

Exercice 6. Écrire une fonction `est_croissante(L)` qui renvoie `True` si la liste `L` est rangée par ordre croissant et `False` sinon.

III Chercher

On souhaite maintenant écrire une fonction `cherche(L, x)` qui prend en argument une liste `L` et un objet `x` et cherche l'élément `x` dans la liste `L`. Encore une fois, il faut parcourir la liste ; le démarrage est donc nécessairement

```python
def cherche(L, x):
    for i in range(len(L)):
        if ... :
    ... ... ...
    ... ... ...
```

mais ensuite... On fait les remarques suivantes :

1. On peut s'intéresser soit à l'élément lui-même, soit à son indice dans la liste. Ici, on veut son indice (la variable `i` telle que `L[i]` soit égal à `x`).
2. L'élément `x` peut apparaitre plusieur fois dans la liste. Mais si on arrête la fonction *dès que* `x` est trouvé, alors on obtiendra l'indice de **la première** apparition de `x` dans la liste. Au contraire, si `x` n'apparait pas du tout alors il faut bien aller au bout de la liste pour le savoir...
3. Enfin, contrairement à la situation « compter », il n'y a pas de bonne valeur cohérente à renvoyer si `x` n'est pas dans la liste. On propose de renvoyer l'objet spécial `None` qui indique l'absence de valeur.

Exercice 7. Compléter la fonction ci-dessus pour que `cherche(L, x)` renvoie le premier indice de la liste où `x` apparait, et `None` s'il n'apparait pas.

Exercice 8. Écrire une fonction `premier_negatif(L)` qui renvoie le premier élément de `L` qui est strictement négatif (l'élément, pas son indice), et `None` s'il n'y a pas de tel élément.

IV D'autres exercices

Exercice 9. On suppose que la liste `L` ne contient que des nombres entre 0 et 9. Dans ce cas, on souhaite compter combien de fois apparait *chaque* chiffre, en renvoyant une liste `C` de longueur 10 telle que `C[x]` donne le nombre de fois où le chiffre `x` apparait dans `L`. Si on s'y prend bien, on peut le faire en parcourant la liste une seule fois, au lieu d'appeler 10 fois une fonction pour compter...
Écrire cette fonction, qu'on appellera `compte_tout(L)`.

Exercice 10. On considère qu'un mot de passe valide sera formé uniquement des caractères parmi ceux-ci : `"abcdefghijklmnopqrstuvwxyz0123456789"`

1. Écrire une fonction `caractere_valide(x)` qui teste si `x` est un caractère valide ou non.
2. En déduire une fonction `motdepasse_valide(m)` qui teste si `m`, une chaine de caractères, représente un mot de passe valide.
3. Bonus : écrire une fonction `motdepasse_fort(m)` qui teste si `m` est valide et contient au moins un caractère parmi `"0123456789"`.
   Le plus efficacement possible.

Exercice 11. Écrire une fonction `est_monotone(L)` qui renvoie `True` si la liste `L` est soit croissante soit décroissante, et `False` sinon.
Sans écrire séparément des fonctions annexes `est_croissante(L)` et `est_decroissante(L)`.

Exercice 12. On s'intéresse à compter le nombre de mots dans une chaine de caractères. Pour simplifier on suppose que seuls les caractères espaces sont utilisés pour séparer les mots… Mais qu'il peut y avoir plusieurs espaces consécutifs ! Le texte suivant

```
"    jaime     programmer en    Python   "
```

comporte bien quatre mots mais on voit qu'on ne peut pas se contenter de compter le nombre de caractères espace, qui n'a pas de sens ici.
Pour résoudre ce problème, on itère sur la chaine de caractères en utilisant une variable booléenne `le_precedent_est_espace` qui indique si le caractère précédent était un espace ou non. Ainsi on compte un mot de plus uniquement dans le cas où on lit un caractère non-espace **et** le caractère précédent était un espace.
Écrire la fonction `compte_mots(s)` qui compte comme décrit le nombre de mots dans la chaine `s`.