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 précédents et est résumé sur le cours distribué au début d'année.

On propose d'étudier un certain nombre de situations classiques sur les listes : compter les éléments, tester une propriété, chercher un élément. 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. Les méthodes sont les mêmes pour traiter des listes ou bien des chaines de caractères.

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**. Dans les exercices qui vont suivre, pour vérifier que sa fonction est correcte, il est nécessaire de la tester avec des valeurs pour lesquelles tout marche bien mais aussi avec des valeurs qui pourraient mettre en échec le programme. **Regarder attentivement les tests proposés** et **observer les résultats du programme** tout en **réfléchissant bien** !

I Compter et accumuler

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 (ou nuls) 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 que l'instruction `return c` est exécutée à la toute fin après avoir parcouru *toute* la liste. Ainsi ce programme compte bien, parmi la liste toute entière, le nombre de termes qui sont positifs.

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 qui compte combien de fois `x` apparait dans la liste `L`.

Exercice 2
Écrire une fonction `différences(L, M)` qui prend en argument deux listes `L`, `M`, supposées de même longueur (on ne demande pas que la fonction vérifie cette condition) et compte à combien d'indices les éléments `L[i]` et `M[i]` sont différents.

Par exemple les listes `L = [3, 7, 6, 5, 3]` et `M = [3, 8, 6, 5, 4]` sont différentes à 2 indices, pour $i = 1$ et $i = 4$.

Les chaines de caractères se manipulent de la même manière : pour une telle chaine `s`, alors la longueur est `len(s)`, et le i-ème caractère est `s[i]`, numéroté de 0 à $n - 1$. Ainsi une boucle `for` comme les précédentes va parcourir les caractères uns par uns de la chaine. Un caractère seul s'écrit entre guillemets doubles `"a"`, `"b"`, etc.

Exercice 3
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 4
Écrire une fonction `somme(L)` qui calcule la somme de tous les termes de la liste `L`.

II Tester

Une autre situation courante est de vouloir vérifier si 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 (ou nuls), et `False` sinon.

Une seule fonction est correcte parmi les propositions ci-dessous.

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

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

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

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

Exercice 5
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 6
Écrire une fonction `binaire(m)` qui prend en argument une chaine de caractères `m` (par exemple `m = "011101011"`), et qui renvoie `True` si `m` est bien composée uniquement de caractères `0` ou `1`, et `False` sinon.

Exercice 7
Écrire une fonction `est_croissante(L)` qui renvoie `True` si la liste `L` est rangée par ordre croissant et `False` sinon.

À retenir

Quand elle est rencontrée, l'instruction `return` arrête la fonction, même dans une boucle. Mais si la propriété à tester dépend aussi de la suite de la liste, on ne peut pas se contenter de s'arrêter là, et on ne peut pas non plus dire « sinon, continuer à tester »… La bonne façon de faire est alors de **s'arrêter si la condition contraire est vérifiée**, et sinon, le `return True` se situe **à la fin et en dehors de la boucle** : si on est arrivé jusque là, c'est qu'on ne s'est pas arrêté avant, donc que la condition fausse n'a pas été rencontrée, donc que c'est vrai !

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

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

mais ensuite… On fait les remarques suivantes :

- 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`).
- L'élément `x` peut apparaitre plusieurs 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…
- 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 8
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 9
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.

Exercice 10
Écrire une fonction `indice_différents(s, t)` qui prend en argument deux chaines de caractères, supposées de même longueur, et qui renvoie le premier indice auxquel les chaines diffèrent, et `None` si elles sont égales. Par exemple, les chaines `s = "ACGTGATAA"` et `t = "ACGTCATTA"` sont de même longueur 9 et diffèrent aux indices 4 (`s[4] = "G"` et `t[4] = "C"`) et 7 (`s[7] = "A"` et `t[7] = "T"`) donc la fonction doit renvoyer 4.

À retenir

Dans d'autres situations, on peut tout à fait quitter la fonction dès qu'on a trouvé ce qu'on voulait, c'est donc une bonne idée d'avoir une instruction `return` qui est bel et bien dans la boucle (et même, il ne sert à rien de continuer la boucle !) Éventuellement, en fin de fonction et en dehors de la boucle, on traite le cas où on n'a pas trouvé ce qu'on voulait.

IV D'autres exercices

Exercice 11
On suppose que la liste `L` ne contient que des nombres entiers 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 12
On considère qu'un mot de passe valide sera formé uniquement des caractères parmi ceux-ci :
`"abcdefghijklmnopqrstuvwxyz0123456789"`

1. Écrire une fonction `caractère_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 une lettre et un chiffre.

   Le plus efficacement possible.

Exercice 13
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 14 (*)
On considère des listes constituées uniquement de nombres `0` et `1` et on souhaite compter le nombre blocs de `1` consécutifs. Par exemple pour

```
L = [0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1]
```

on compte 4 blocs de `1`, ayant pour tailles respectives 2, 3, 1, 2.

Écrire la fonction `compte_blocs(L)` qui prend en argument une telle liste et renvoie le nombre de blocs.

Attention à ce qu'elle fonctionne correctement dans tous les cas, que les blocs soient calés au début de la liste ou à la fin ou pas du tout.

Exercice 15 (*)
Une **permutation de longueur n** est une liste de longueur n où chacun des nombres de 0 à $n-1$ apparait exactement une fois. Par exemple `[3, 1, 0, 2]` est bien une permutation de longueur 4.

1. Pourquoi suffit-il que chacun de ces nombres apparaisse *au moins* une fois ? Ou bien *au plus* une fois ?
2. Écrire une fonction `appartient(L, x)` qui renvoie `True` si le nombre `x` est présent dans la liste `L` et `False` sinon.
3. En utilisant la fonction précédente, écrire une fonction `est_permutation(L)` qui renvoie `True` si `L` est bien une permutation, et `False` sinon.

Une autre possibilité qui est *plus rapide* mais nécessite *plus de mémoire* est la suivante. Pour tester si la liste `L` est bien une permutation, on crée une liste de booléens `M` de même taille que `L`, et on parcourt une seule fois `L`, mais on « coche » les nombres qu'on a vus. Ainsi `M[x] = True` est à interpréter comme « `x` est bien présent dans `L` » alors que `M[x] = False` signifie que `x` n'a pas encore été rencontré.

4. En utilisant cette méthode, écrire une fonction `est_permutation_2(L)`.

Pour ceux qui ont fini trop vite :

5. Écrire une fonction `permutations(n)` qui renvoie la liste de toutes les permutations de longueur n (une liste de listes !)