# TRAVAUX PRATIQUES n° 8
## Algorithmes sur les listes et les chaînes de caractères

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 chaînes de caractères.

Exercice 1 : Écrire une fonction `compte_positifs(L)` qui prend en argument une liste `L` et qui compte combien de termes de `L` sont positifs.

Exercice 2 : Écrire une fonction `compte(L, x)` qui prend en argument une liste `L` et un nombre `x` et qui compte combien de fois `x` apparaît dans la liste `L`.

Exercice 3 : Écrire une fonction `differences(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 chaînes de caractères se manipulent de la même manière : pour une telle chaîne `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 chaîne. Un caractère seul s'écrit entre guillemets doubles `"a"`, `"b"`, etc.

Exercice 4 : Écrire une fonction `compte_voyelles(s)` qui prend en argument une chaîne de caractères `s` et compte le nombre de voyelles (lettres parmi `a`, `e`, `i`, `o`, `u`, `y`) dans `s`.

Exercice 5 : Écrire une fonction `binaire(m)` qui prend en argument une chaîne 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 6 : Écrire une fonction `est_croissante(L)` qui renvoie `True` si la liste `L` est rangée par ordre croissant et `False` sinon.

Exercice 7 : Écrire une fonction `est_monotone(L)` qui renvoie `True` si la liste `L` est monotone, c'est-à-dire soit croissante soit décroissante et `False` sinon.

Exercice 8 :

1) 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`.
2) Compléter la fonction ci-dessus pour que `cherche(L, x)` renvoie le premier indice de la liste où `x` apparaît, et `None` s'il n'apparaît 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_differents(s, t)` qui prend en argument deux chaînes de caractères, supposées de même longueur, et qui renvoie le premier indice auxquel les chaînes diffèrent, et `None` si elles sont égales. Par exemple, les chaînes `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.

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 apparaît chaque chiffre, en renvoyant une liste `C` de longueur 10 telle que `C[x]` donne le nombre de fois où le chiffre `x` apparaît 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 `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 chaîne 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.

Exercice 13 (*) : 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 14 (*) : Une permutation de longueur $n$ est une liste de longueur $n$ où chacun des nombres de 0 à $n-1$ apparaît 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 !)