TRAVAUX PRATIQUES n° 7
Algorithmes sur les listes et les chaînes de caractères

On propose d'étudier trois situations classiques : **compter**, **tester une propriété**, **chercher un élément**. Les fonctions correspondantes existent souvent déjà en Python, mais on s'interdit de les utiliser (`sum`, `max`, `min`, `in` sur une liste, `liste.index`, `liste.count`, `liste.sort`). On travaille avec des boucles `for`, l'accès aux indices et des conditions booléennes.

Les mêmes méthodes s'appliquent aux **chaînes de caractères** : pour une chaîne `s`, la longueur est `len(s)` et le i-ième caractère est `s[i]`, numéroté de 0 à $n-1$. Une boucle `for` parcourt donc les caractères un par un. Un caractère seul s'écrit `"a"`, `"b"`, etc.

I. Compter

Une situation de base est de **compter** les éléments vérifiant une propriété. On déclare un **compteur** initialisé à 0, que l'on augmente de 1 à chaque fois. Le modèle est le suivant :

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

L'alignement des blocs est crucial : `c = c + 1` n'est exécuté que si `L[i] >= 0` ; le `return c` est exécuté **après** avoir parcouru toute la liste. Si aucun terme n'est positif, `c` reste à 0.

Exercice 1 : Tester la fonction `compte_positifs(L)` sur `[-1, 0, 4, -3, 2]`, puis l'adapter pour ne compter que les termes **strictement** 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`.

```
>>> compte([1, 3, 1, 2, 1], 1)
3
>>> compte([1, 3, 1, 2, 1], 4)
0
```

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`.

```
>>> differences([3, 7, 6, 5, 3], [3, 8, 6, 5, 4])
2
```

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`, majuscules ou minuscules) dans `s`.

```
>>> compte_voyelles("Bonjour")
3
```

II. Tester une propriété

À retenir

L'instruction `return` arrête la fonction, même au milieu d'une boucle. Pour tester une propriété sur **toute** la liste, on s'arrête dès que le contraire est vérifié, et on met `return True` **après** la boucle : si on y arrive, c'est que le contraire n'a jamais été rencontré.

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.

```
>>> binaire("011101011")
True
>>> binaire("011201011")
False
```

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

```
>>> est_croissante([1, 2, 2, 5])
True
>>> est_croissante([1, 3, 2])
False
```

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.

III. Chercher

Pour chercher un élément, le canevas est :

```
1 def cherche(L, x):
2     for i in range(len(L)):
3         if ... :
4             ...
5     ...
```

- ⋆ On veut ici l'indice `i` tel que `L[i] == x`.
- ⋆ Si on s'arrête dès que `x` est trouvé, on obtient la **première** occurrence. S'il n'apparaît pas, il faut aller au bout de la liste.
- ⋆ S'il n'y a pas d'occurrence, on renvoie `None` (absence de valeur).

Exercice 8 : Écrire une fonction `cherche(L, x)` qui renvoie le premier indice de la liste où `x` apparaît, et `None` s'il n'apparaît pas.

```
>>> cherche([4, 1, 7, 1], 1)
1
>>> cherche([4, 1, 7, 1], 3)
None
```

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.

```
>>> premier_negatif([2, -1, -4, 0])
-1
>>> premier_negatif([2, 1, 4, 0])
None
```

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 auquel 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.

```
>>> indice_differents("ACGTGATAA", "ACGTCATTA")
4
>>> indice_differents("ABC", "ABC")
None
```

IV. Synthèse

Exercice 11 : On suppose que la liste `L` ne contient que des nombres entiers entre 0 et 9. 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.

Écrire cette fonction, qu'on appellera `compte_tout(L)`.

```
>>> compte_tout([1, 0, 1, 3, 1])
[1, 3, 0, 1, 0, 0, 0, 0, 0, 0]
```

Exercice 12 : On considère qu'un mot de passe valide est formé uniquement des caractères parmi ceux-ci : `"abcdefghijklmnopqrstuvwxyz0123456789"`

Pour cet exercice, on autorise l'opérateur `in` pour tester si un caractère appartient à cette chaîne fixe de caractères valides.

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.

V. Pour aller plus loin

Exercice 13 (*) : On considère des listes constituées uniquement de nombres 0 et 1 et on souhaite compter le nombre de 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 !)