Comptage des éléments d'un tableau à l'aide d'un dictionnaire

 # Position du problème : il s'agit de dénombrer pour chaque élément d'une liste le nombre de ses occurrences, et de présenter le résultat dans un dictionnaire.

Exemple : on tire des billes de couleur d'un sac et on note : \['red', 'blue', 'red', 'green', 'blue', 'blue'\].

On veut en déduire : \{'blue': 3, 'red': 2, 'green': 1\} (ou une permutation puisque l'ordre n'est pas figé).

""" Méthode optimale (mais hors programme) :

Utiliser le module **collections**, qui "*implémente des types de données de conteneurs spécialisés qui apportent des alternatives aux conteneurs natifs de Python plus généraux dict, list, set et tuple.*"

(réf. https://docs.python.org/fr/3/library/collections.html).

Importer **Counter** (avec la majuscule) depuis ce module et l'appliquer à la liste, pour obtenir directement un objet de type dictionnaire dont les clés sont les objets de la liste, et les valeurs associées à chacun le nombre d'occurrences.

"""

```
from collections import Counter
print(Counter(L))
compte = dict(Counter(L)) ; print(compte)
```

Exo 1 : Méthode "gourmande"

Utiliser une définition de dictionnaire "en compréhension" en prenant chaque élément de la liste L et en lui associant le nombre de ses occurrences, obtenu par la *méthode de liste* **L.count(élément)**.

```
L = ['red', 'blue', 'red', 'green', 'blue', 'blue']
compte = {couleur: L.count(couleur) for couleur in L}
print(compte)
```

Estimer la **complexité** de la méthode proposée, en nombre d'opérations, en fonction du nombre *n* d'objets de la liste.

On imagine que la méthode 'L.count' doit parcourir toute la liste pour trouver le nombre d'apparitions de l'objet cherché, soit n valeurs à tester.

Comme on appelle 'L.count' n fois, le nb d'opérations sera prop. à n^2 ; complexité quadratique.

Exo 2 : Méthode constructiviste

(recommandée, dans l'esprit du programme) Définir un dictionnaire vide, puis parcourir la liste (une seule fois) et créer les paires *clé:valeur*, en incrémentant la valeur autant que nécessaire. NB: dico('clé') renvoie une erreur si 'clé' n'est pas dans dico.

```
compte = {}
for x in L:
    if x in compte: # il faut distinguer les cas où l'entrée 'x' existe déjà ou non
        compte[x] += 1 # si clé existante on incrémente la valeur
    else:
        compte[x] = 1 # sinon il faut créer la clé et l'initialiser à la valeur 1
print((compte))
```

amélioration :

Dans la démarche précédente, le *hic* vient de ce qu'il faut distinguer les clés existantes de celles à créer : *on ne peut pas à la fois créer, initialiser et incrémenter la valeur*.

On peut contourner le problème grâce à la méthode de dictionnaire **dico.get('clé', défaut = '...')** qui renvoie la valeur associée à une 'clé' existante, et la valeur par défaut si la clé est introuvable :

```
compte = {}
for x in L:
    compte[x] = compte.get(x,0) + 1 # si entrée absente, créée et initialisée à 0 + 1 donc 1
print((compte))
```

Complexité :

Il y a bien sûr n tours de la boucle "x in L", puis les opérations nécessitées par le test "if x in compte".

Le pire des cas est celui d'une liste de n valeurs différentes : il faut à chaque fos créer une entrée dans le dico.

La première fois qu'on le parcourt il y a une entrée, puis 2, puis 3... et la dernière fois il y aura n entrées à vérifier.

La somme de tout ça est donc de la forme 1 + 2 + ... + n, donc quadratique.

Le meilleur des cas est celui d'une séie de valeurs identiques, alors le dico n'a qu'une entrée et "if x in compte" est de coût 1 constant.

La complexité totale sera donc elle de la boucle : linéaire.

Exo 3 : Recherche du maxi et du 2e maxi dans une liste

génération d'une liste :

from random import randint

N = 5

L = [randint(0,100) for i in range(N)]

Q1 :

```
L = [9,-4,2,-7,1,12,0,-3,4,-11]
N = len(L)
ind = 0 ; mini = L[0]
for i in range(1,N):
    if L[i] < mini :
        ind = i ; mini = L[i]
print(mini)
```

Q2

```
def minimum(v):
    N = len(v)
    ind = 0 ; m = L[0]
    for i in range(1,N):
        if v[i] < m :
            ind = i ; m = v[i]
    return m
```

Q3

```
L = [9,-4,2,-7,1,12,0,-3,4,-11]
def minimum(v):
    N = len(v)
    ind = 0
    m = v[0]
    for i in range(1,N):
        if v[i] < m :
            ind = i
            m = v[i]
    return m,ind # cela renvoie un tuple

minima = []
mini,ind = minimum(L) # depaquetage du tuple dans deux varables
minima.append(mini)
L.pop(ind) # on enlève le plus petit élément
# on cherche le min dans ce qui reste de L :
mini,ind = minimum(L)
minima.append(mini)
print(minima)
```

Q4

```
L = [9,-4,2,-7,1,12,0,-3,4,-11]
print(L)
Lc = []
while(len(L)>0):
    mini,ind = minimum(L)
    Lc.append(mini)
    L.pop(ind)
print(Lc)

L = [9,-4,2,-7,1,12,0,-3,4,-11]
print(L)
Ld = []
while(len(L)>0):
    mini,ind = minimum(L)
    Ld = [mini]+Ld
    L.pop(ind)
print(Ld)
```

Q5 : on a vidé la liste L. Si on veut la conserver, il faut la dupliquer : L2 = L[:]

Q6 : complexité quadratique : le nombre de calculs effectués par le processeur varie comme le carré de len(L)

Q7 On crée une fonction maximum

```
def maximum(v):
    N = len(v)
    ind = 0
    m = v[0]
    for i in range(1,N):
        if v[i] > m :
            ind = i
            m = v[i]
    return m,ind

L = [9,-4,2,-7,1,12,0,-3,4,-11]
print(L)
Lc = []
while(len(L)>0):
    maxi,ind = maximum(L)
    Lc.append(maxi)
    L.pop(ind)
print(Lc)
```

Exo 4 : Recherche des 2 éléments de valeurs les plus proches dans une liste

Q1

```
L = [9,-4,2,-7,1,12,0,-3,4,-11]

im,jm = 0,1 # initialise 2 indices
em = abs(L[im]-L[jm]) # initialise l'écart mini
for i in range(N):
    for j in range(i):
        e = abs(L[i] - L[j])
        if e < em:
            em = e
            im = i
            jm = j
valeurs = (L[im],L[jm])
print('les valeurs les plus proches sont :'+str(valeurs) + ' aux indices : '+str((im,jm)))
```

Q2 : complexité quadratique : le nombre de calculs effectués par le processeur varie comme le carré de len(L)

Exo 5 : Simplification d'un labyrinthe parfait

 # Q3 une boucle while car on ne sait pas a priori combien d'impasses on va rencontrer.

Q4

```
E=[1,1,2,1,2,1,1,1,1,2,1,0] # explore en tournant toujours à gauche
while 2 in E: # tant qu'il a rencontré une impasse
    ind = E.index(2)
    E[ind-1:ind+2]=[-1] # remplace les [1,2,1] par [-1]

    for i in range(1,len(E)-1):
        if E[i-1:i+2]==[-1,2,1] or E[i-1:i+2]==[1,2,-1]:
            E[i-1:i+2]=[2] # remplace les [-1,2,1] et les [1,2,-1] par [2]
    print(E)
```

Q5

```
Ed=[-1,-1,2,-1,2,-1,-1,-1,2,-1,0] # explore en tournant toujours à droite
while 2 in Ed:
    ind = Ed.index(2)
    Ed[ind-1:ind+2]=[1] # remplace les [-1,2,-1] par [1]

    for i in range(1,len(Ed)-1):
        if Ed[i-1:i+2]==[-1,2,1] or Ed[i-1:i+2]==[1,2,-1]:
            Ed[i-1:i+2]=[2] # remplace les [-1,2,1] et les [1,2,-1] par [2]
    print(Ed)
```

Q6

```
E=[1,1,2,1,2,1,1,1,1,2,1,0] # Labyrinthe exploré en tournant toujours à gauche
Ed=[-1,-1,2,-1,2,-1,-1,-1,2,-1,0] # Labyrinthe exploré en tournant toujours à droite

def Simplifie(L):
    for i in range(len(L)):
        if L[i] != 2: # dès qu'on trouve autre chose qu'une impasse,
            s = L[i] # affecter cette valeur au sens retenu pour l'exploration
            break

    while 2 in L:
        ind = L.index(2)
        L[ind-1:ind+2]=[-s] # remplace les [-1,2,-1] par [-s]

        for i in range(1,len(L)-1):
            if L[i-1:i+2]==[s,2,-s] or L[i-1:i+2]==[-s,2,s]:
                L[i-1:i+2]=[2] # remplace les [-1,2,1] et les [1,2,-1] par [2]
    print(L)

Simplifie(E)
```