TP03 : Listes – 2 [version du 13/10/2022]

1 Parcours de liste et applications

Pour explorer et exploiter, à l'aide de Python, une liste `L` déjà définie en mémoire, la méthode par défaut consiste à « **parcourir** » la liste *L*, ce qui se fait à l'aide des deux méthodes de parcours suivantes :

- **Parcours par les indices** :
  À l'aide d'un appel à l'itérateur `range`, on fait prendre à l'indice de boucle (*k* ici) toutes les valeurs de position des éléments de la liste *L* (*k* prend les valeurs de 0 à `len(L)-1`), ce qui permet d'accéder à tous les éléments de la liste *L*, dans l'ordre.
  On peut, par exemple, afficher tous les éléments de *L* dans l'ordre où ils apparaissent dans *L* :

```
L = [45, 24, 78, 13, 100, 2]
# parcours de L par les indices (ici pour afficher les éléments de L)
for k in range(len(L)):
    print(L[k])
```

- **Parcours par les valeurs** :
  La construction « `for val in L :` » permet d'accéder à toutes les valeurs stockées dans `L`, dans l'ordre où elles apparaissent dans `L`. À chaque itération, la variable `val` prend la valeur d'un élément de `L`.

```
L = [45, 24, 78, 13, 100, 2]
# parcours de L par les valeurs (ici pour afficher les éléments de L)
for val in L:
    print(val)
```

Important : le parcours par les valeurs est moins « puissant » que le parcours par les indices. Il faut déjà retenir que ce que permet un parcours par les valeurs peut toujours être réalisé par un parcours par les indices, et plus encore retenir que **de nombreux algorithmes nécessitant de parcourir une liste ne sont pas implémentables** (réalisables) **à l'aide d'un parcours par les valeurs** (en tout cas pas de façon simple), à savoir : **tous ceux qui requerraient de modifier (*en place*) la liste** ou **d'utiliser la position des éléments**.

À titre d'exemple, un parcours par les valeurs ne permet pas :

- de multiplier par 2 toutes les valeurs d'une liste (il s'agit ici de modifier *en place* la liste) ;
- rechercher de la position du maximum d'une liste (il faut ici connaître la position des valeurs lues).

À titre d'illustration de ce qu'il serait difficile d'obtenir à l'aide d'un parcours par les valeurs, mais qu'il est facile d'obtenir à l'aide d'un parcours par les indices, on peut donner aussi l'exemple de l'affichage des couples d'éléments consécutifs dans une liste :

```
L = ["a", "b", "c", "d", "e"]
## affichage des couples d'éléments consécutifs dans L avec leurs positions
for k in range(len(L)-1):
    print((L[k], L[k+1]), "couple des éléments en position", (k, k+1))
```

On obtient l'affichage

```
('a', 'b') couple des éléments en position (0, 1)
('b', 'c') couple des éléments en position (1, 2)
('c', 'd') couple des éléments en position (2, 3)
('d', 'e') couple des éléments en position (3, 4)
```

Il est par ailleurs sur cet exemple essentiel de noter que l'on a fait prendre à l'indice de boucle *k* les valeurs 0 à `len(L) - 2` (valeurs de 0 à 3 sur l'exemple) en utilisant **`range(len(L)-1)`** et non `range(len(L))`, afin que la plus grande valeur prise par *k* soit `len(L)-2` (position de l'avant-dernier élément dans `L`, 3 sur l'exemple) afin que l'accès, `L[k+1]`, à la valeur en position $k+1$ dans *L* ne déclenche pas une erreur « `index out of range` » lorsque *k* prend sa plus grande valeur.

Ce dernier exemple est une source d'inspiration pour l'exercice classique suivant :

- énoncé :

Définir une fonction `est_croissante`, prenant en argument une liste d'entiers ou de flottants et renvoyant un booléen, `True` ou `False`, selon que la liste passée en argument est croissante ou non. Proposer quatre tests pour cette fonction.

- une solution :

```
def est_croissante(L):
    for k in range(len(L)-1):
        if L[k] > L[k+1]:
            return False # (note 1)
    return True # (note 2)cette instruction est exécutée
                # uniquement si la boucle for a été jusqu'à son terme
                # c'est-à-dire ici uniquement si la valeur False n'a pas été déjà
                # renvoyée
```

Note 1 : si l'expression `L[k] > L[k+1]` est évaluée à `True`, l'instruction `return False` est exécutée, ce qui a pour conséquences :

1. la boucle `for` est interrompue sur le champ ;
2. la valeur `False` est renvoyée, ce qui « termine » la fonction.

Note 2 : l'instruction `return True`, située **en dehors** (désindentation) et **après** la boucle `for`, n'est exécutée uniquement si la boucle `for` a été jusqu'à son terme (toute la liste a été parcourue, l'indice de boucle a pris toutes les valeurs de 0 à `len(L)-2`), c'est-à-dire ici uniquement si la valeur `False` n'a pas été déjà renvoyée.

- Propositions de tests pour la fonction `est_croissante` :
  1. l'appel `est_croissante([4, 8, 9, 10])` doit renvoyer `True` (on teste ici que la fonction renvoie la bonne valeur pour une liste croissante) ;
  2. l'appel `est_croissante([2.1, 3.3, -1e-5, 10e10])` doit renvoyer `False` (on teste ici que la fonction renvoie la bonne valeur pour une liste non-croissante) ;
  3. l'appel `est_croissante([])` doit renvoyer `True` (on teste ici un cas particulier potentiellement problématique ou non pris en charge) ;
  4. l'appel `est_croissante([4])` doit renvoyer `True` (on teste ici un cas particulier potentiellement problématique ou non pris en charge).

1.1 Application 1 : calcul de la somme des termes d'une liste

Sous forme de script (programme composé d'une séquence d'instructions enregistré dans un fichier textuel d'extension **.py**) :

```
L = [10, 20, 30, 40, 50]
## calcul de la somme des termes de L
s = 0 ## initialisation d'une variable s à 0 (ACCUMULATEUR)
for k in range(len(L)):
    s = s + L[k] ## chaque valeur rencontrée est ajoutée à l'accumulateur
```

En fin d'exécution, la variable `s` aura pour valeur $10 + 20 + 30 + 40 + 50 = 150$.

Sous forme d'une fonction :

À l'aide d'un parcours par les indices	**À l'aide d'un parcours par les valeurs**

`def somme1_liste(L):`
`  s = 0`
`  for k in range(len(L)):`
`  s = s + L[k]`
`  return s`
	`def somme2_liste(L):`
	`  s = 0`
	`  for val in L:`
	`  s = s + val`
	`  return s`

Les appels `somme1_liste([10, 20, 30, 40, 50])` et `somme2_liste([10, 20, 30, 40, 50])` renverront tous deux la valeurs 150.

Exercice 1. Dans un script **ex1.py**, définir deux fonctions de paramètre `L`, prenant en argument une liste d'entiers ou de flottants, et renvoyant, respectivement, la somme des valeurs positives de cette liste et la somme des valeurs négatives de cette liste. On nommera `f1` et `f2` ces deux fonctions, la première opérera à l'aide d'un parcours de *L* par les valeurs, la seconde à l'aide d'un parcours de *L* par les indices.
Proposer trois tests pour chaque de ces deux fonctions.
Effectuer un test complémentaire en générant une liste de 100 entiers tirés aléatoirement entre 1 et 100, et en comparant la somme obtenue par appel aux deux fonctions `f1` et `f2` et la somme envoyée par appel à la fonction prédéfinie `sum` (disponible nativement dans Python) (`sum(L)` renvoie la somme des éléments d'une liste *L* d'entiers ou de flottants).

1.2 Application 2 : filtrage d'une liste

Le **filtrage** d'une liste *L* consiste à construire une nouvelle liste, constituée uniquement des éléments de *L* vérifiant une certaine condition.

- exemple : extraire d'une liste d'entiers, uniquement ceux qui sont supérieurs strictement à 50

```
L = [45, 24, 78, 13, 100, 2]
## filtrage des éléments supérieurs strictement à 50
Lacc = [] ## initialisation d'une liste Lacc à une liste vide
          ## (Lacc sert d'ACCUMULATEUR)
for k in range(len(L)): # parcours de la liste L
    if L[k] > 50:       # comparaison de chaque valeur dans L à 50
        Lacc.append(L[k]) # ajout de la valeur dans l'accumulateur Lacc si > 50
```

À la fin de l'exécution de ce script, la liste `Lacc` est la liste `[78, 100]`.

Exercice 2. Dans un script **ex2.py**, définir une fonction `pairs_impairs`, prenant en argument une liste d'entiers *L* et renvoyant sous la forme d'un tuple de listes, la liste des éléments de *L* qui sont pairs, et la liste des éléments de *L* qui sont impairs.
Proposer trois tests pour cette fonction.

Exercice 3. **[N.B. : ce qui est demandé dans cet exercice n'est pas ce que l'on appelle un filtrage de la liste.]**

Dans un script **ex3.py**, définir une fonction `rangs_pairs_impairs`, prenant en argument une liste *L* et renvoyant sous la forme d'un tuple de listes, la liste des éléments de rang pair de *L*, et la liste des éléments de rang impair de *L*.
Proposer une solution utilisant les compréhensions de liste.
Proposer trois tests pour cette fonction.

1.3 Application 3 : recherche du maximum d'une liste non vide

La recherche du maximum d'une liste non vide se fait à l'aide de l'algorithme implémenté ci-dessous sous la forme d'une fonction `rechmax(L)` :

```
def rechmax(L):
    """ L est supposée non vide"""
    vmax = L[0]# initialisation de la valeur la plus grande rencontrée jusque là
    for k in range(1, len(L)):
        if L[k] > vmax:
            vmax = L[k] # actualisation de la plus grande valeur rencontrée
    return vmax # renvoi de la valeur maximale rencontrée
```

- Variante possible pour une liste de flottants (moins usitée, permettant un parcours par les valeurs et prenant en charge le cas d'une liste vide)

On recourt ici à la valeur spéciale flottante, renvoyée par l'appel `float('inf')`, qui renvoie une valeur flottante, affichée `inf`, qui a la particularité d'être supérieure strictement à toute autre valeur de type `float`.

```
>>> float('inf')
inf
>>> 10e300 > float('inf')
False
```

Corollairement, `-float('inf')` produit une valeur flottante ayant la propriété d'être inférieure strictement à tout autre valeur flottante.

- Implémentation de l'algorithme :

```
def rechmaxfloat(L):
    """ L est une liste de flottants éventuellement vide"""
    vmax = - float('inf') # valeur la plus grande rencontrée jusque là
    for val in L:
        if val > vmax:
            vmax = L[k] # actualisation de la plus grande valeur rencontrée
    return vmax # renvoi de la valeur maximale rencontrée
```

Exercice 4. Dans un script **ex4.py**, recopier le code de la fonction `rechmax(L)` et écrire quatre tests pour cette fonction sur des exemples bien choisis.
Faire de même pour la fonction `rechmaxfloat(L)`.

Exercice 5. Dans un script **ex5.py**, définir une fonction `rechposmax` en adaptant le code de la fonction `rechmax` de l'**exercice 4**, afin que la fonction `rechposmax` envoie non pas la valeur maximale de liste passée en argument, mais la position d'une des occurrences de cette valeur maximale.
Avec l'implémentation choisie, quelle position renvoie la fonction si la liste en argument est composée de valeurs toutes égales ? quelle position renvoie la fonction si la valeur maximale est présente exactement deux fois dans la liste ?

Exercice 6. Dans un script **ex6.py**, définir, en adaptant le code de la fonction `rechmax` de l'**exercice 6**, une fonction `rechmin` renvoyant non pas la valeur maximale de liste passée en argument, mais la valeur minimale dans cette liste.

Exercice 7. Dans un script **ex7.py**,

1. Écrire une fonction `rechmax1max2` prenant en argument une liste L comportant au moins deux éléments, et renvoyant la valeur et la position de ses deux premiers maximum (valeur la plus grande et seconde valeur la plus grande, éventuellement égales) ;
2. Combien de comparaisons sont effectuées lors d'un appel à `rechmax1max2` sur la liste $[1, 3, 2]$?
3. Combien de comparaisons au maximum sont effectuées lors d'un appel à `rechmax1max2` sur une liste à 5 éléments ? Lors de l'appel sur une liste à n éléments ?

1.4 Application 4 : « mappage d'une liste par une fonction »

Exercice 8. Dans un script **ex8.py**,

1. Définir une fonction `map1(L, f)`, qui prend en argument une liste L et une fonction f, et qui, si $L = [x_0, x_1, \dots, x_n]$, renvoie une nouvelle liste, égale à $[f(x_0), f(x_1), \dots, f(x_n)]$.

Tester votre fonction avec la liste des $(x_k)_{0\leq k\leq 20} = \left(\frac{k\pi}{2}\right)_{0\leq k\leq 20}$, construite à l'aide d'une compréhension de liste, et la fonction cosinus.

2. Reprendre dans le même script, **ex8.py**, l'exercice **8** et définir une fonction `map2(L, f)`, qui prend en argument une liste L et une fonction f, et qui, si $L = [x_0, x_1, \dots, x_n]$, modifie la liste L, afin qu'elle soit égale à $[f(x_0), f(x_1), \dots, f(x_n)]$.

Tester votre fonction sur le même exemple qu'en question **8.**

3. Dans les **questions 1.** et **2.** peut-on indifféremment utiliser un parcours par les indices ou un parcours par les valeurs ?

2 Copie de liste et *Slicing*

Exercice 9. Dans un script **ex9.py**,

1. Définir une fonction `copie(L)` renvoyant la copie d'une liste L.
2. Définir une fonction `slicing(L, a, b, c)` prenant en argument une liste L, et renvoyant une liste égale à `L[a:b:c]`.
3. Pour l'une et l'autre de ces fonctions, un parcours par les valeurs peut-il convenir ?