Leçon d'informatique : listes

S. Benlhajlahsen - PCSI$_1$

Sommaire

Introduction Les types `int`, `float` et `bool` sont des types élémentaires : une variable d'un tel type est constituée d'une seule valeur. Nous avons aperçu le type `str`, qui est différent : une seule variable de type chaîne peut contenir plusieurs caractères. C'est un type **composite**. Nous allons étudier le plus important et le plus polyvalent des types composites offerts par Python : le type `list`, tout simplement appelé "liste" en Français. En TP, vous passerez beaucoup de temps à manipuler des listes !

I Définition et synthaxe

I.A Définition

À retenir : Une variable de type list est une collection d'éléments telle que :

- C'est une collection **ordonnée** : on peut repérer les éléments par leur position (appelé indice) dans la liste ;
- Chaque élément peut être d'un type différent, y compris `list` (listes imbriquées) ;
- Une liste est **mutable** : on peut modifier un élément de la liste après sa création ;
- Une liste est **dynamique** : on peut ajouter ou enlever des éléments après sa création.

I.A.1 Syntaxe de la valeur d'une liste

Synthaxe : La valeur d'une variable de type `list` est délimitée par des crochets. Entre ces crochets, les éléments sont séparés par des virgules. En particulier, `[]` est la liste vide.

Exemple Dans l'exemple de la figure 1, on définit une liste à 5 éléments composée de deux entiers, d'une chaîne de caractères, d'un booléen et d'un flottant.

```
1 L1 = [2, 4, "abc", True, 3.42]
```

FIGURE 1 – Exemple de liste.

I.B Une liste est ordonnée

Exemple Dans l'exemple de la figure 2, on comprend que deux listes composées des mêmes éléments mais dans des ordres différents sont deux variables différentes.

```
1 >>> [1,2] == [2,1]
2 False
```

FIGURE 2 – Une liste est un ensemble **ordonné**.

Synthaxe : indice Soit `L1` une liste de n éléments. Les règles gouvernant l'indice de position sont :

- Les éléments sont numérotés (ou indexés) de 0 à $n-1$.
- `L1[i]` désigne l'élément numéro i. En particulier, `L1[n-1]` désigne le dernier élément.
- `L1[i:j]` désigne la sous-liste de $j-i$ éléments allant de l'élément i à l'élément $j-1$ [a].

Signalons que la même syntaxe fonctionne avec les chaînes de caractères.

a. Cette opération est souvent appelée *slicing* ou hachage de liste.

```
>>> L1 =  [ 2, 4, "abc", True , 3.42 ]
>>> L1[0]
2
>>> L1[4]
3.42
>>> L1[5]
IndexError: list index out of range
>>> L1[2:4]
['abc', True]
```

FIGURE 3 – Récurépération d'éléments sur une liste.

Remarque Pour ce dernier exemple, l'opération de *slicing* ou hachage va conserver tous les élements compris entre les deux "cloisons" d'indices 2 et 4 (voir figure 4).

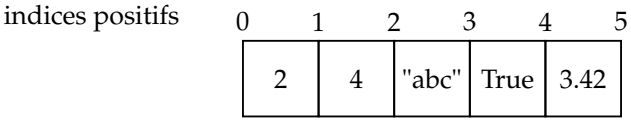

FIGURE 4 – Slicing ou hachage.

Remarque Dans cette opération de hachage `L1[i:j]`, les deux indices autour des deux points sont facultatifs. Si on omet i, il sera remplacé par 0 et si on omet j, il sera remplacé par n (la longueur de la liste). On pourra se reporter à l'exemple de la figure 5

```
>>> L1[:3]
[2, 4, 'abc']
>>> L1[2:]
['abc', True, 3.42]
```

FIGURE 5 – Slicing sur une liste.

I.C Les éléments peuvent être de types différents

Remarque La liste précédente était composée d'éléments de type différents. En outre, un élément peut être lui-même de type `list`. Voir l'exemple de la figure 6.

```
>>> L2  = [25, [True, 8, 4]]
>>> L2[0]
25
>>> L2[1]
[True, 8, 4]
```

FIGURE 6 – Liste dont l'un des éléments est une liste.

Synthaxe : La fonction `len` renvoie le nombre d'éléments d'une liste ou encore sa "longueur" (voir figure 7).

```
>>> L2  = [25, [True, 8, 4]]
>>> len(L2)
2
>>> len(L2[1]
3
```

FIGURE 7 – Méthode `len`.

I.D Une liste est mutable

À retenir : Un type de variable est dit **mutable** si on peut modifier sa valeur après sa création sans recréer la variable. Les types vus jusqu'ici (`int`, `float`, `bool` et `str`) sont non mutables (voir figure 8).

```
>>> L3 = [False, False, False]
>>> L3[0] = True
>>> print(L3)
[True, False, False]
```

FIGURE 8 – Une liste est mutable.

Remarque On peut aussi remplacer des sous-listes (voir figure 9).

```
>>> L4 = [1, 3, 5, 7, 9]
>>> L4[2:4] = [-11, -12]
>>> print(L4)
[1, 3, -11, -12, 9]
```

FIGURE 9 – On peut remplacer des sous-listes.

I.E Une liste est dynamique

Méthode `append` La méthode `append` ajoute un élément **à la fin** d'une liste, qui compte alors un élément de plus. Mais sa syntaxe est très particulière (voir figure 10).

```
>>> L5 = [10, 20]
>>> L5.append(30)
>>> print(L5)
[10, 20, 30]
```

FIGURE 10 – Synthaxe de la méthode `append`.

Attention ! La méthode `append` modifie directement la liste mais **ne renvoie rien**. La synthaxe `L = L.append(élément)` n'a aucun sens (voir figure 11) !

```
>>> type(L5.append(40))
<class 'NoneType'>
>>> L6 = L5.append(50)
>>> print(L6)
None
```

FIGURE 11 – La méthode `append` ne renvoie rien.

II Opérations sur les listes

Synthaxe Les listes se concatènent comme les chaînes de caractères avec l'opérateur `+` (voir figure 12).

```
>>> [1, -10.7, True] + [False , 7]
[1, -10.7, True, False, 7]
```

FIGURE 12 – Concaténation de liste.

Remarque : temps d'éxécution Pour ajouter un unique élément au bout d'une liste, faut-il préférer `append` ou `+`? Le script de la figure 13 permet de calculer la durée totale de chaque boucles. Sur mon ordinateur en 2019, j'ai mesuré :

- 29 secondes pour la concaténation ;
- 21 millisecondes pour la méthode `append`.

La concaténation doit créer une nouvelle liste contenant les éléments des deux listes concaténées, alors que `append` ajoute l'élément au bout de la liste existante sans toucher aux autres éléments.

```
1  from time import time
2  N = 100000
3
4  # Essai par concaténation
5  L = []
6  start = time()
7  for i in range(N) :
8      L = L+[0]
9  stop = time()
10 duree = stop - start
11 print('{:.10f} secondes'.format(duree))
12
13 # Essai avec append
14 L = []
15 start = time()
16 for i in range(N) :
17     L.append(0)
18 stop = time()
19 duree = stop - start
20 print('{:.10f} secondes'.format(duree))
```

FIGURE 13 – Temps d'éxécution.

II.A Concaténation

II.B Répétition sur une liste

Synthaxe de répétition L'opérateur `*` permet la répétition de liste. Ainsi, `3*[1,2]` renvoie `[1, 2, 1, 2, 1, 2]`.

II.C Conversions

Méthode `list` La méthode `list` convertit certaines variables en listes. Ainsi :

- `list("abc")` renvoie : `['a', 'b', 'c']`
- `list(range(4))` renvoie : `[0, 1, 2, 3]`

Par contre, cela ne fonctionnent pas avec les type élémentaires (`int`, `float` ou `bool`). Ainsi, `list(2.3)` renvoie le message d'erreur `'int' object is not iterable`.

Conversion d'une liste en chaîne de caractères Pour convertir une liste en chaîne de caractères, on propose plusieurs méthodes en figure 14.

```
1  >>> L = ["a", "b", "c"]
2  ### méthode 1
3  >>> chaine = ""
4  >>> for x in L:
5          chaine += x
6  ### méthode 2
7  >>> chaine = ""
8  >>> for i in range(len(L)):
9          chaine += L[i]
10 ### méthode 3 hors-programme
11 >>> "".join(L)
```

FIGURE 14 – Conversions de listes en chaînes de caractères.

II.D Copier une liste

Copie de variables élémentaires On considère la figure 15. Comme on pouvait s'y attendre, les deux variables `a` et `b` sont totalement indépendantes. On peut en modifier une sans implication sur l'autre. On parle de **"vraie" copie** ou **copie profonde**.

```
1  >>> a = 1
2  >>> b = a
3  >>> print(a,b)
4  1 1
5  >>> a = 2
6  >>> print(a,b)
7  2 1
```

FIGURE 15 – Copies de variables élémentaires.

Copie superficielle de liste On considère la figure 17. A présent, on constate que les deux listes ne sont pas indépendantes et la modification de `L1` entraîne une modification `L2`. On parle de **copie superficielle**[a].

a. ou *shallow copy*

```
>>> L1 = [1,2]
>>> L2 = L1
>>> print(L1, L2)
[1, 2] [1, 2]
>>> L1[0] = 100
>>> print(L1, L2)
[100, 2] [100, 2]
```

FIGURE 16 – Copie superficielle de liste.

À retenir : Quand on utilise la syntaxe habituelle d'affectation entre deux listes, on ne fait que donner deux noms à une seule et même liste. Manipuler la liste sous n'importe lequel des deux noms revient au même.

Solution 1 Pour créer une vraie copie, on peut le faire *à la main*. Cependant, cette méthode ne marche évidemment que s'il n'y a pas de sous-listes.

```
>>> L1  = [1,2,3]
>>> L2 = []
>>> for x in L1:
        L2.append(x)
```

FIGURE 17 – Vraie copie.

Solution 2 On peut utiliser le module `copy` et la méthode `deepcopy` (voir figure 18). On va bien que la modification de `L1` n'a pas modifié `L2`.

```
>>> from copy import deepcopy
>>> L1 = [1,2]
>>> L2 = deepcopy(L1)
>>> print(L1, L2)
[1, 2] [1, 2]
>>> L1[0] = 100
>>> print(L1, L2)
[100, 2] [1, 2]
```

FIGURE 18 – Copie profonde.

III Trucs et astuces sur les listes

III.A Parcourir une liste à l'envers

Remarque Pour la méthode `range(start, stop, step)`, on pouvait avoir un pas `step` positif ou négatif. On peut faire la même chose avec les listes.

Synthaxe : `liste[-1]` désigne le dernier élément d'une liste. C'est pratique car cela dispense de connaître la valeur de l'indice de position du dernier élément ! Ensuite, `liste[-2]` est l'avant-dernier élément, et ainsi de suite : si n est la longueur de la liste, `liste[i]` et `liste[i-n]` désignent le même élément (voir figures 19 et 20).

```
>>> print(L6[-1])
-2
>>> print(L6[-3])
6
```

FIGURE 19 – Exemples d'indices négatifs.

III.B Boucles `for` sur une liste

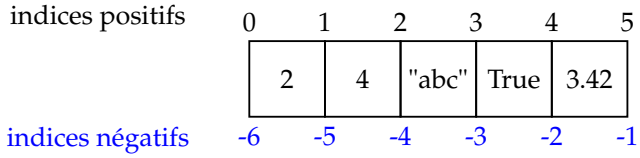

FIGURE 20 – Indices négatifs d'une liste. Ici, la liste est de longueur $n = 5$.

Somme des termes d'une liste Soit `L` une liste composée uniquement d'entiers ou de flottants. Pour calculer la somme des éléments de la liste, on propose deux solutions :

- on itère sur les indices de la liste (voir colonne gauche de la figure 20) ;
- on itère sur les éléments de la liste (voir colonne droite de la figure 20) ;

Bien entendu, cela ne fonctionne que si tous les éléments de la liste soient tous de même type !

```python
def somme_liste(L):
    res = 0
    for i in range(len(L)):
        res += L[i]
    return res
```

```python
def somme_liste(L):
    res = 0
    for a in L:
        res += a
    return res
```

FIGURE 21 – Boucle `for` sur une liste. Somme des termes d'une liste.

Parcourir la liste à l'envers Pour parcourir une liste à l'envers, on pourra se référer aux solutions proposées en figure 22.

```python
# solution 1
L = [0,1,2,3]
n = len(L)
for x in L[::-1]:
    print(x)
# solution 2
for i in range(n):
    print(L[n-i-1])
# solution 3
for i in range(n-1,-1,-1):
    print(L[i])
# solution hors-programme
for x in reversed(L):
    print(x)
```

FIGURE 22 – Parcourir la liste à l'envers.

III.C Listes imbriquées.

Vers les matrices Considérons la liste `L =[[10,20,30],[40,50,60]]`. C'est une liste de longueur 2^a. Chacune des sous-listes est de longueur 3^b. On peut alors la représenter comme ci-dessous :

```
[[  10  ,  20  ,  30  ],
 [  40  ,  50  ,  60  ]]
```

ind. de colonne $j \rightarrow$	0	1	2
ind. de ligne $i \downarrow$			
0	10	20	30
1	40	50	60

Ainsi, `L[i][j]` va chercher le j-ième élément de la i-ième sous-liste ou encore l'élément qui se situe en ligne d'indice i et de colonne j. `L[1][2]` renvoie l'élément qui se trouve en deuxième ligne[c] et troisième colonne[d] (voir figure 23).

a. `len(L)` renvoie 2
b. `len(L[0])` ou `len(L[1])`.
c. indice $i = 1$.
d. indice $j = 2$.

```
1 >>> L =[[10,20,30],[40,50,60]]
2 >>> L[1][2]
3 60
```

FIGURE 23 – Liste de listes.

IV Listes comme argument d'entrée d'une fonction

Rappel sur les variables locales On considère le script de la figure 24. La variable locale `a` reçoit à sa naissance la valeur de la variable globale `x` mais elle est par la suite une variable indépendante de `x`. Donc la ligne 2 modifie `a`, qui meurt quand la fonction se termine, et ne touche pas à `x`.

```
1 >>> def f(a):
2 >>>     a += 1
3 >>> x = 1
4 >>> f(x)
5 >>> print(x)
6 1
```

FIGURE 24 – Variable locale.

Synthaxe : Comme expliqué précédemment, quand un argument d'entrée est de type `int`, `float`, `bool` ou `str`, une variable locale est créée pour en recevoir la valeur. Toute modification de cette variable locale laisse inchangées les variables globales.

Synthaxe : Quand une liste est passée en argument d'entrée, il n'y a pas de création d'une variable locale. C'est la même variable des deux côtés, donc si la fonction modifie la liste, la modification est répercutée en dehors. Le fait que la liste porte des noms différents dans la fonction et en dehors n'y change rien : la liste est repérée par son adresse dans la mémoire, pas par son nom. On dit que les deux variables pointent vers la même liste en mémoire (voir figure 25).

```
1 >>> def f(L):
2 >>>     L[1] = 0
3 >>> L2 = [1,2]
4 >>> f(L2)
5 >>> print(L2)
6 [1, 0]
```

FIGURE 25 – Liste comme argument de fonction.

Remarque Bien sûr, si vous forcez la création d'une variable locale de même nom, la règle habituelle[a] s'applique (voir figure 26). Cette fois la ligne 2 crée une variable locale `L`, de sorte que la ligne 3 modifie cette variable locale. La liste `L2` n'est donc jamais modifiée.

a. les variables locales sont prioritaires

```
1 def f(L):
2         L=[0,1]
3         L[1] = 0
4 >>> L2 = [1,2]
5 >>> f(L2)
6 >>> print(L2)
7 [1, 2]
```

FIGURE 26 – Liste comme argument de fonction.

Remarque Créer une variable locale portant le même nom qu'un argument d'entrée est considéré comme une faute de goût. Cela rend votre code moins lisible et peut être source d'erreur (confusion).

V Listes en compréhension

V.A Synthaxe sans condition

Remarque Cette appellation, qui n'a guère de sens en Français, est la traduction littérale de *list comprehension*. Une traduction plus correcte pourrait être "liste générée formellement", mais l'usage en a décidé autrement.

Exemple On souhaite générer la liste des carrés des entiers de 0 à 6. On peut procéder de manière "brutale" (colonne gauche de la figure 27) mais Python permet d'utiliser une syntaxe beaucoup plus concise et lisible (colonne droite de la figure 27). Cette synthaxe est à rapprocher d'une écriture mathématique de la forme

$$L = \left\{ i^2,\ i \in [\![0,6]\!] \right\}$$

```
n=6
L=[]
for i in range(n+1):
    L.append(i**2)
```

```
n=6
L=[i**2 for i in range(n+1)]
```

FIGURE 27 – Liste des carrés. A gauche, construction par concaténation et, à droite, en compréhension.

V.B Synthaxe avec condition

Exemple On souhaite générer la liste contenant tous les entiers pairs de 0 à 12. Là encore, une boucle `for` fait le travail (voir figure 28) mais on peut faire plus concis avec une liste en compréhension. C'est à rapprocher d'une écriture mathématique de la forme

$$L = \{ i,\ i \in [\![0,12]\!] \text{ et } i\%2 = 0 \}$$

```
1 # solution 1
2 n = 12
3 L = []
4 for i in range(n+1):
5     if i%2 == 0:
6         L.append(i)
7 # solution 2
8 n = 12
9 L = [i for i in range(n+1) if i%2==0]
```

FIGURE 28 – Liste des entiers.

VI Exercices

VI.A Enoncés

Exercice 1 : Écrivez une fonction `maximum` qui reçoit en entrée une liste de nombres et renvoie en sortie le plus grand d'entre eux. Vous ferez une version utilisant l'indice de position dans la liste et une version sans (voir solution en figure 29).

Exercice 2 : Écrivez une fonction `moyenne` qui reçoit en entrée une liste de nombres et renvoie en sortie la moyenne arithmétique de ses éléments. Vous ferez une version utilisant l'indice de position dans la liste et une version sans (voir solution en figure 30).

Exercice 3 : Écrivez une fonction `nb_occurences` qui prend en entrée une liste et un nombre, et renvoie en sortie le nombre d'occurrences de ce nombre dans la liste (voir solution en figure 31).

Exercice 4 : Raffinement de l'exercice précédent. Écrivez une fonction `occurrences` prenant les mêmes entrées, mais renvoyant en sortie la liste des indices de chaque occurrence du nombre dans la liste (voir solution en figure 32).

VI.B Solutions

```python
def maximum1(L):
    res = L[0] # initialisation sur le
        premier element
    n = len(L)
    for i in range(1,n):
        if L[i]>res:
            res = L[i]
    return res

def maximum2(L):
    res = L[0] # initialisation sur le
        premier element
    n = len(L)
    for x in L:
        if x>res:
            res = x
    return res
```

FIGURE 29 – solution de l'exercice 1.

```python
def moyenne1(L):
    res = 0 # initialisation de la moyenne
    n = len(L)
    for i in range(n):
        res+= L[i]
    return res/n

def moyenne2(L):
    res = 0 # initialisation de la moyenne
    n = len(L)
    for x in L:
        res += x
    return res/n
```

FIGURE 30 – solution de l'exercice 2.

```python
def nb_occurences(L, a):
    # nombre d'occurences de a dans L
    res = 0
    for x in L:
        if x==a:
            res += 1
    return res
```

FIGURE 31 – solution de l'exercice 3.

```python
def occurences(L,a):
    res = []
    # liste des indices des occurences
    n = len(L)
    for i in range(n):
        if L[i]==a:
            L.append(i)
    return res
```

FIGURE 32 – solution de l'exercice 4.