Cours : Liste Python

Définition

Une **liste** est une structure de données qui contient une série de valeurs. Python autorise la construction de liste contenant des valeurs de types différents (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité. Une liste est déclarée par une série de valeurs (n'oubliez pas les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des **virgules**, et le tout encadré par des **crochets**. En voici quelques exemples :

```
animaux = ["girafe", "tigre", "singe", "souris"]
tailles = [5, 2.5, 1.75, 0.15]
mixte = ["girafe", 5, "souris", 0.15]
```

Lorsque l'on affiche une liste, Python la restitue telle qu'elle a été saisie.

Utilisation

Un des gros avantages d'une liste est que vous pouvez appeler ses éléments par leur position. Ce numéro est appelé **indice** (ou *index*) de la liste.

```
Liste : ["girafe",   "tigre",   "singe",   "souris"]
Indice :   0            1          2            3
```

Soyez très **attentifs** au fait que les indices d'une liste de *n* éléments commence à *0* et se termine à *n-1*. Voyez l'exemple suivant :

```
animaux = ["girafe", "tigre", "singe", "souris"]
>>> animaux[0]
'girafe'
>>> animaux[1]
'tigre'
>>> animaux[0]
'girafe'
>>> animaux[3]
'souris'
```

Par conséquent, si on appelle l'élément d'indice 4 de notre liste, Python renverra un message d'erreur :

```
>>> animaux[4]
Traceback (most recent call last):
File "<console>", line 1, in <module>
IndexError: list index out of range
```

Opération sur les listes

Tout comme les chaînes de caractères, les listes supportent l'opérateur `+` de concaténation, ainsi que l'opérateur `*` pour la duplication (par un entier) :

```
>>> ani1 = ['girafe','tigre']
>>> ani2 = ['singe', 'souris']
```

```
>>> ani1 + ani2
['girafe', 'tigre', 'singe', 'souris']
>>> ani1*3
['girafe', 'tigre', 'girafe', 'tigre', 'girafe', 'tigre']
```

L'opérateur `+` est très pratique pour concaténer deux listes.

Indiçage

1) Longueur d'une chaîne

La liste est caractérisée par un nombre d'éléments. L'instruction len() renvoie la longueur de la liste.

```
Liste          : ["girafe",   "tigre", "singe",      "souris"]
len(Liste) # renvoie 4 (attention numérotation de 0 à (len(Liste)-1)
```

2) Indiçage négatif

La liste peut également être indexée avec des nombres négatifs selon le modèle suivant :

```
Liste          : ["girafe",   "tigre", "singe",      "souris"]
Indice positif :    0          1        2             3
Indice négatif :    -4         -3       -2            -1
```

Les indices négatifs reviennent à compter à partir de la fin. Leur principal avantage est que vous pouvez accéder au dernier élément d'une liste à l'aide de l'indice `-1` sans pour autant connaître la longueur de cette liste. L'avant-dernier élément a lui l'indice `-2`, l'avant-avant dernier l'indice `-3`, etc.

3) Tranches ou slicing

Un autre avantage des listes est la possibilité de sélectionner une partie d'une liste en utilisant un indiçage construit sur le modèle `[m:n+1]` pour récupérer tous les éléments, du émième au énième (de l'élément `m` inclus à l'élément `n+1` exclu). On dit alors qu'on récupère une **tranche** de la liste, par exemple :

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> animaux[0:2]
['girafe', 'tigre']
>>> animaux[0:3]
['girafe', 'tigre', 'singe']
>>> animaux[0:]
['girafe', 'tigre', 'singe', 'souris']
>>> animaux[:]
['girafe', 'tigre', 'singe', 'souris']
>>> animaux[1:]
['tigre', 'singe', 'souris']
>>> animaux[1:-1]
['tigre', 'singe']
```

Notez que lorsqu'aucun indice n'est indiqué à gauche ou à droite du symbole deux-points, Python prend par défaut tous les éléments depuis le début ou tous les éléments jusqu'à la fin respectivement.
On peut aussi préciser le pas en ajoutant un symbole deux-points supplémentaire et en indiquant le pas par un entier.

Création d'une liste

1) Liste en place

La liste peut être créée en écrivant chacun des termes séparés par une virgule et le tout encadré par des crochets :

```
mixte = ["girafe", 5, "souris", 0.15]
```

2) Méthode append

La liste peut être créée en partant d'une liste vide et en rajoutant un à un les éléments :
Script :

```
L=[] # liste vide
for i in range(10):
        L.append(i) # rajout de l'élément i
print('L= ',L)
>>> (executing lines 1 to 4 of "exemple copie listes.py")
L= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

3) Liste par compréhension

Un autre moyen pour créer une liste est la compréhension de liste dont la syntaxe est :

```
new_list = [function(item) for item in list if condition(item)]
```

Script :

```
L= [ i for i in range(10)]
print('L= ',L)
>>> (executing lines 1 to 2 of "exemple copie listes.py")
L= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4) Exercices

- Ecrire une fonction moyenne1 qui à partir d'une liste comportant que des entiers, renvoie la moyenne des termes de la liste en parcourant les termes par leur indice.

- Ecrire une fonction moyenne2 qui à partir d'une liste comportant que des entiers, renvoie la moyenne des termes de la liste en parcourant les termes par leur valeur.

Liste de listes

1) Exemple

Chaque élément d'une liste peut être lui-même une liste :

L=[['Janvier',31],['Février',28],['Mars',31]]
L[0] renvoie : ['Janvier',31]
Et donc L[0][1] renvoie : 31

Attention à l'ordre des indices
L[1][0] renvoie : 'Février'

2) Création par compréhension

Script :

```
LL=[[x*y for x in range(1,5)] for y in range(1,3)]
print(LL)
>>> (executing lines 1 to 2 of "exemple copie listes.py")
    [[1, 2, 3, 4], [2, 4, 6, 8]]
```

Copie de listes

1) Caractère mutable

La liste est un des rares éléments mutables de Python. On peut modifier une liste de différentes manières :

- Modification d'un élément

```
x = [1, 2, 3, 5]
x[-1] = 4 # remplace le 5 par un 4
```

- Rajout d'un élément

```
x = [1, 2, 3, 5]
x.append(6)  # rajout de l'élément 6 en fin de liste
```

- Suppression d'un élément

```
x = [1, 2, 3, 5]
x.pop() # supprime par défaut le dernier élément donc le chiffre 5
x.pop(0) # supprime le premier élément donc ici le chiffre 1
```

2) Référence

Les listes sont renvoyées par **référence**. C'est-à-dire que lorsque l'on stocke une liste dans une variable, on ne stocke pas la liste elle-même mais une référence vers cette liste (l'adresse mémoire où est stockée cette liste).

```
x = [1, 2, 3, 5]
y = x
x .append(6)
 print(y) # renvoie [1, 2, 3, 5, 6]
```

Dans cet exemple, les variables x et y contiennent une référence vers une même liste ([1, 2, 3, 5]). Lorsque cette liste est modifiée, les valeurs de x et de y sont modifiées.

3) La concaténation

```
x = [1, 2, 3, 5]
y = x
x = x + [6]
 print(y) # renvoie [1, 2, 3, 5], la liste y n'a pas été modifiée.
```

Dans cet exemple, l'opération x + [6] renvoie une nouvelle liste contenant [1, 2, 3, 5, 6] mais ne modifie pas la liste anciennement référencée par x([1, 2, 3, 5]). La valeur de y ne change donc pas.

4) L'assignation augmentée

```
x = [1, 2, 3, 5]
y = x
x += [6]
 print(y) # renvoie [1, 2, 3, 5, 6], la liste y a été modifiée.
```

Dans cet exemple, l'opération x += [6] rajoute un élément à la liste x sans en créer de nouvelle. La liste y étant une copie de la liste x, contenant désormais [1, 2, 3, 5, 6]. La liste y évolue. De plus, les copies y et x ne sont pas indépendantes.

5) Fonction id()

id() est une fonction de Python qui accepte un seul paramètre et est utilisée pour renvoyer l'identité d'un objet (ou l'adresse mémoire de stockage de l'entité).

Reprenons l'exemple de la concaténation :

Script :

```
x = [1, 2, 3, 5]
print('début id(x) :', id(x))
y = x
print('id(y) :', id(y))
x = x+ [6]
print('fin id(x) :', id(x))
print('x = ', x)
print('y = ', y)
>>> (executing lines 1 to 8 of "exemple copie listes.py")
début id(x) : 61655688
id(y) : 61655688
fin id(x) : 45809992
x = [1, 2, 3, 5, 6]
y = [1, 2, 3, 5]
```

Lors que l'on concatène l'élément 6 à la liste existante x, Python crée une nouvelle identité puisque id(x) a évolué. La modification de x n'influe pas sur y comme on peut le constater.

Reprenons l'exemple d'utilisation de la méthode append :

Script :

```
x = [1, 2, 3, 5]
print('début id(x) :', id(x))
y = x
print('id(y) :', id(y))
x.append(6)
print('fin id(x) :', id(x))
print('x = ', x)
print('y = ', y)
>>> (executing lines 1 to 8 of "exemple copie listes.py")
début id(x) : 61655176
id(y) : 61655176
fin id(x) : 61655176
x = [1, 2, 3, 5, 6]
y = [1, 2, 3, 5, 6]
```

La méthode append ne modifie pas l'identité de x, les variables x et y pointent vers la même adresse mémoire. Toute modification de l'une des listes (si non création d'une nouvelle identité) entraine la modification de l'autre liste.

Dernière vérification sur le caractère mutable cette fois-ci :

Script :

```
x = []
print('1 id(x) :',id(x))
x = [1, 2, 3, 5]
print('2 id(x) :',id(x)) # création d'une nouvelle entité
x.append(6)
print('3 id(x) :',id(x)) # pas de modification de l'id()
x.pop()
print('4 id(x) :',id(x)) # pas de modification de l'id()
x += [7]
print('4 id(x) :',id(x)) # pas de modification de l'id()
```

```
>>> (executing lines 1 to 10 of "exemple copie listes.py")
1 id(x) : 61655816
2 id(x) : 61655240
3 id(x) : 61655240
4 id(x) : 61655240
4 id(x) : 61655240
```

En résumé : pour rajouter un élément à une liste, il est préférable d'utiliser la méthode append ou bien l'assignation augmentée.

6) Copie de liste

Le problème consiste à recopier les éléments d'une liste en ayant des identités différentes afin que les deux listes puissent évoluer ensuite indépendamment l'une de l'autre. Plusieurs méthodes existent :

Script :

```
L0 = [4,5]
L1 = L0[:]
print('id(L0) :',id(L0))
print('id(L1) :',id(L1)) # renvoie une valeur différente de id(L0)
>>> (executing lines 1 to 4 of "exemple copie listes.py")
id(L0) : 55807560
id(L1) : 61654472
```

Script :

```
L0 = [4,5]
L1 = [ x for x in L0]
print('id(L0) :',id(L0))
print('id(L1) :',id(L1)) # renvoie une valeur différente de id(L0)
>>> (executing lines 1 to 4 of "exemple copie listes.py")
id(L0) : 55844424
id(L1) : 55807560
```

```
L0 = [4,5]
L1 = L0.copy()
print('id(L0) :',id(L0))
print('id(L1) :',id(L1)) # renvoie une valeur différente de id(L0)
>>> (executing lines 1 to 4 of "exemple copie listes.py")
id(L0) : 61515464
id(L1) : 55844424
```

En conclusion, 3 manières d'effectuer une copie de liste.

7) Copie de liste de listes

Reste à valider que les méthodes de copie de liste fonctionnent également pour les listes de listes.

Script :

```
L0 = [[2,3], 4, 5]
L1 = L0[:]
print('id(L0) :',id(L0))
print('id(L1) :',id(L1)) # renvoie une valeur différente de id(L0)
L0[0][0] = 1
print('L1 =',L1)
>>> (executing lines 1 to 6 of "exemple copie listes.py")
id(L0) : 55807112
id(L1) : 61515464
L1 = [[1, 3], 4, 5]
```

On constate que L0 et L1 ont bien des identités différentes, mais une modification d'un terme de la sous liste de L0 est répercuté sur L1. L0 et L1 ne sont pas des copies indépendantes. Il est en est de même pour les deux autres méthodes :

- L1 = [x for x in L0]
- L1 = L0.copy()

Pour remédier au problème, on peut si il existe un seul niveau de sous liste et que chaque élément de la liste initiale est une liste, procéder ainsi.
Script :

```
L0 = [[2,3], [4, 5]]
L1 = [ x[:] for x in L0]
print('id(L0) :',id(L0))
print('id(L1) :',id(L1)) # renvoie une valeur différente de id(L0)
L0[0][0] = 1
print('L1 =',L1)
>>> (executing lines 1 to 6 of "exemple copie listes.py")
id(L0) : 61515336
id(L1) : 55807112
L1 = [[2, 3], [4, 5]]
```

Une modification d'un terme d'une sous liste de L0 ne se répercute pas sur la sous liste de L1.

Module copy :

La méthode fortement conseillée consiste à utiliser les fonctions copy et deepcopy du module copy.
Script :

```
import copy
L0 = [4, 5]
print('id(L0) :',id(L0))
L1 = [L0, 2]
L2 = copy.copy(L1)
print('copy id(L1) :',id(L1))
print('copy id(L2) :',id(L2))
print('copy id(L1[0]) :',id(L1[0]))
print('copy id(L2[0]) :',id(L2[0]))
L0[0] = 0
print('L2 =',L2)
>>> (executing lines 1 to 11 of "exemple copie listes.py")
id(L0) : 55494984
copy id(L1) : 61569224
copy id(L2) : 61655048
copy id(L1[0]) : 55494984
copy id(L2[0]) : 55494984
L2 = [[0, 5], 2]
```

On constate que la fonction copy.copy() crée bien un copie de la liste L1 avec une identité différente. Mais une modification d'un terme de la sous liste L0 est répercutée sur L2. On pourra donc utiliser copy.copy() uniquement pour copier des listes mais pas pour des copies de listes de listes.

Fonction deepcopy du module copy :

Il existe une version récursive de copy.copy() qui est copy.deepcopy() qui opère de la même manière que copy.copy() pour chaque niveau de sous liste. On parle alors de copie profonde.
Script :

```
import copy
L0 = [4, 5]
```

```
print('id(L0) :',id(L0))
L1 = [L0, 2]
L2 = copy.deepcopy(L1)
print('deepcopy id(L1) :',id(L1))
print('deepcopy id(L2) :',id(L2))
print('deepcopy id(L1[0]) :',id(L1[0]))
print('deepcopy id(L2[0]) :',id(L2[0]))
L0[0] = 0
print('L2 =',L2)
>>> (executing lines 1 to 11 of "exemple copie listes.py")
id(L0) : 55844424
deepcopy id(L1) : 61513928
deepcopy id(L2) : 61569224
deepcopy id(L1[0]) : 55844424
deepcopy id(L2[0]) : 61654344
L2 = [[4, 5], 2]
```

On constate que les deux listes L1 et L2 sont bien indépendantes après copie de L2 dans L1, et ceci grâce à l'usage de deepcopy du module copy.

Fonction Python avec une liste comme paramètre

1) Cas d'une variable non mutable

Script :

```
def test1(a):
        a +=1
        print('a =',a)

b = 1
test1(b)
print('b = ', b)
```

```
>>> (executing lines 1 to 7 of "exemple copie listes.py")
a = 2
b = 1
```

La fonction test1 reçoit comme valeur pour son unique paramètre la valeur 1 stockée dans b. On incrémente a, b reste inchangé. Tout est normal pourrait on dire.

2) Cas d'une variable mutable

Script :

```
def test2(a):
        a +=[1]
        print('a =',a)

b = [0]
test2(b)
print('b = ', b)
```

```
>>> (executing lines 1 to 7 of "exemple copie listes.py")
a = [0, 1]
b = [0, 1]
```

La fonction test2 reçoit comme « valeur » pour son unique paramètre la liste b. On rajoute un élément à a et on constate que la liste b a été également modifiée. L'explication de cette différence de comportement s'explique par le caractère mutable qui conduit non pas à transmettre à la fonction test2 la valeur de la liste b mais son identité, et donc a et b pointent sur la même adresse mémoire.

3) Modification en place

On souhaite écrire une fonction, qui modifie en place une liste (par exemple pour la trier, dont on simulera le comportement ici par un simple échange de deux termes). Cela signifie que l'on exploite le caractère mutable et la fonction en général ne renvoie rien.

Script :

```
def tri(LL):
        LL[0],LL[1] = LL[1], LL[0]

L = [3, 2, 4]
```

```
tri(L) # on appelle la fonction tri qui modifie la liste L
print('L = ', L)
>>> (executing lines 1 to 6 of "exemple copie listes.py")
L = [2, 3, 4]
```

L'appel à la fonction tri a permis de modifier la liste L sans utiliser de return. On parle alors de modification en place.

4) Sans modification en place

On souhaite écrire une fonction, qui à partir d'une liste renvoie la liste triée (dont on simulera le comportement ici de nouveau par un simple échange de deux termes). Cela signifie que l'on ne souhaite pas modifier la liste initiale, et donc on en effectue une copie avant de la trier puis de la renvoyer.

Script :

```
def tri2(LL):
        L_tri =LL[:] # copie ou copy.copy(LL) ou LL.copy()
        L_tri[0], L_tri[1] = L_tri[1], L_tri[0]
        return L_tri

L = [3, 2 , 4]
L_trie = tri2(L) # on appelle la fonction qui renvoie la liste L triée
# sans modifier la liste L

print('L = ', L)
print('L_trie = ', L_trie)

>>> (executing lines 1 to 11 of "exemple copie listes.py")
L = [3, 2, 4]
L_trie = [2, 3, 4]
```

Ne pas confondre les deux comportements attendus de la fonction et la manière de faire appel à la fonction.

Les méthodes qui opèrent sur les listes

Comme pour les chaînes de caractères, les listes possèdent de nombreuses **méthodes** qui leur sont propres et qui peuvent se révéler très pratiques. On rappelle qu'une méthode est une fonction qui agit sur l'objet auquel elle est attachée par un point.

1) Les méthodes usuelles sur les listes

Script :

```
L1 =[1,2,3]
print('Aff1 : ',L1)
L1.append(4) # rajoute l'élément 4 sans modifier l'identité de L1
print('Aff2 : ',L1)
L1.insert(0,3) #insère à l'indice 0 la valeur 3
print('Aff3 : ',L1)
L1.remove(3) # supprime un élément à partir de sa valeur (le premier rencontré)
```

```
print(‘Aff4 : ‘,L1)
L1.remove(3) # supprime un élément à partir de sa valeur (le premier rencontré)
print(‘Aff5 : ‘,L1)
>>> (executing lines 1 to 10 of "exemple copie listes.py")
Aff1 : [1, 2, 3]
Aff2 : [1, 2, 3, 4]
Aff3 : [3, 1, 2, 3, 4]
Aff4 : [1, 2, 3, 4]
Aff5 : [1, 2, 4]
```

2) Les méthodes pour trier

Script :

```
L2 =[3, 1, 2]
print(‘Aff1 : ‘,L2)
L2.sort() # trie en place la liste
print(‘Aff2 : ‘,L2)
L2.sort(reverse=True) # tri en place la liste, ordre inversé
print(‘Aff1 : ‘,L2)
L3 = sorted(L2) # trie L2 sans la modifier et renvoie le résultat
print('L2 = ',L2)
print('L3 = ',L3)

>>> (executing lines 1 to 9 of "exemple copie listes.py")
Aff1 : [3, 1, 2]
Aff2 : [1, 2, 3]
Aff3 : [3, 2, 1]
L2 = [3, 2, 1]
L3 = [1, 2, 3]
```

3) Autres méthodes

Script :

```
L3 = [4, 5, 6, 5, 4, 7, 5]
L3.reverse() # inverse L3
print(‘Aff1 : ‘,L3)
nb5 = L3.count(5) # compte le nombre d'occurrence de l'élément de valeur 5
print(‘Aff2 : ‘,nb5)

>>> (executing lines 1 to 5 of "exemple copie listes.py")
Aff1 : [5, 7, 4, 5, 6, 5, 4]
Aff2 : 3
```

4) Comptage

```
L4 = [4, 5, 6]
print(max(L3)) # le maximum
print(min(L3)) # le minimum
print(sum(L3)) # la somme des termes
print(sum(L3)/len(L)) # pour déterminer la moyenne
```

5) Test d'appartenance

Script :

```
L5 =[7, 8, 9, 11]
print(9 in L5) # test d'appartenance
print(not 0 in L5) # permet de vérifier qu'un élément n'est pas dans la liste
```

```
>>> (executing lines 1 to 3 of "exemple copie listes.py")
True    # 9 appartient à la liste L5
True    # 0 n'appartient pas à la liste L5
```

6) Pour aller plus loin

Il existe d'autres méthodes :

- L.pop() # pour supprimer un élément
- L.insert() # pour insérer un élément

dir(ma_liste) # ma_liste étant une liste renvoie toutes les méthodes disponibles sur les listes.

Il est important de savoir créer et manipuler les listes, rechercher un élément, le supprimer, copier une liste, …

Les éléments à connaître

- Savoir créer une liste (en place, liste vide et méthode append, compréhension) :

```
L1 = [1,2,3] # création en place
L2 = [] # liste vide
L2.append(4) # on rajoute un élément
L3 = [x**3 for x in range(3)] # par compréhension
```

- Parcourir les éléments d'une liste (par indice ou valeur) :

```
L1=[x**3 for x in range(3)]
for k in range(len(L1)): # parcourt par indice k=0 puis 1 puis 2 (3 valeurs)
        print(L1[k])

for Lk in L1: # parcourt par valeur
        print(Lk)

print(L1[-1]) # indiçage négatif
```

- Copier une liste

```
L1=[x**3 for x in range(3)]
L2 =L1[:]
L3 = [x for x in L2]
print(id(L1),id(L2),id(L3))

import copy
L4 =[[4,5],[6,7]] # liste de listes
L5 = copy.deepcopy(L4)
L4[1][0]=0
print(L4)
print(L5)
```

- Passage d'argument (modification en place)

```
L1 = [1,2,3]
def modifie(L):
    L.append(4)

modifie(L1)
print(L1)
```

- Les méthodes qui opèrent sur les listes (dont tri et appartenance)

```
L1 = [1,2,3]
print(not 0 in L1)
```

Exercices complémentaires

1) Fonction Ecart1

Ecrire une fonction qui à partir d'une liste d'entiers non triés, renvoie les deux éléments les plus proches.
Par exemple :
L1 = [1, 5 , 13, 6]
Ecart1(L1) renvoie (5,6)

2) Fonction Ecart2

Ecrire une fonction qui à partir d'une liste d'entiers triés, renvoie les deux éléments les plus proches.
Par exemple :
L2 = [1, 5 , 6, 13]
Ecart2(L2) renvoie (5,6)