Informatique

7

Listes

Cours

CC BY NC SA

CC BY NC SA

Listes

1.I. Description

Une liste est un ensemble d'objets mis les uns à côté des autres dans un ordre précis. Elle s'écrit à l'aide de crochets ouverts et fermés encadrant différentes données séparées par des virgules.

1.II. Créer une liste

Appelons dans la suite L la liste créée.

1.II.1 Liste vide

Créer une liste vide permet ensuite de lui ajouter des termes, lors d'itérations dans des boucles par exemple. Pour la créer, il suffit d'écrire :

```
L = []
```

1.II.2 Liste d'objets divers

Il est possible de créer une liste avec des objets de types différents, prenons l'exemple suivant :

```
L = ['essai',1,True,None,2,'fin']
```

1.II.3 Liste d'objets répétés

Pour créer une liste contenant n fois le même objet, il suffit d'écrire :

```
>>> L = 2*['a']

>>> L
['a', 'a']

>>> L = 10*[0]

>>> L
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

>>> L = 2*['a',1]

>>> L
['a', 1, 'a', 1]
```

1.II.4 Listes de lettres

Pour créer une liste de lettres, il suffit d'écrire :

$$L = ["a","b","c","d","e","f","g","h"]$$

Toutefois, dans ce cas très particulier, je vous recommande d'utiliser des chaines de caractères puisqu'il existe a peu près les mêmes méthodes sur ces deux objets (`L[i]`, `len(L)`). Il est donc plus simple d'écrire :

$$L = "abcdefgh"$$

Ces deux objets sont bien différents :

```
>>> L1 = ["a","b","c","d","e","f","g","h"]

>>> L2 = "abcdefg"

>>> type(L1)
<class 'list'>

>>> type(L2)
<class 'str'>
```


CC BY NC SA

1.II.5 Ajouter/retirer un élément

Lorsqu'une liste existe, il est possible d'ajouter ou enlever un élément très facilement avec les fonctions pop et append :

```
>>> L
[1, 2, 3]

>>> L.append(10)

>>> L
[1, 2, 3, 10]

>>> L.pop()
10

>>> L
[1, 2, 3]

>>> L.pop()
3

>>> L
[1, 2]
```

$L.append(x)$ ajoute l'élément x dans la liste

$L.pop()$ retire le dernier élément (à droite) et retourne sa valeur, qui peut par exemple alors être utilisée comme nouvelle variable :

```
>>> L
[1, 2]

>>> x = L.pop()

>>> x
2

>>> L
[1]
```

On peut aussi ajouter deux listes en faisant : `L+=[1]`. Attention, même si écrire `L+=L+[1]` fonctionne, c'est une grosse erreur car cela coûte en temps. Si la liste L possède n valeur, la complexité de cette écriture est en $O(n)$ alors qu'append ou `L+=[1]` est en $O(1)$.

Attention L.pop() renvoie un message d'erreur si la liste est vide puisqu'il n'y a plus d'objets à enlever.

Il est possible d'enlever un élément en début de liste en écrivant $L.pop(0)$

En réalité, on peut enlever n'importe quel objet en écrivant $L.pop(i)$ où i décrit sa position en partant de 0 pour le premier élément.

ATTENTION : écrire L.append(Objet) modifie L en mémoire, mais ne renvoie rien dans la console : « on va à la ligne ». En effet :

```
>>> L.append(1)==None
True
```

Exécuter L.append(Objet) renvoie en réalité un objet appelé « NONE ». Lorsque l'on écrit :

~~L = L.append(Objet)~~

, cela revient à écrire L = NONE et L est donc d'abord modifiée (ajout de l'objet), puis supprimée !!!!!

```
>>> L=[]

>>> L = L.append(5)

>>> L

>>> print(L)
None
```


CC BY NC SA

1.III. Les méthodes de l'objet de type liste

Pour connaitre toutes les possibilités (méthodes) associées aux listes, il suffit d'écrire :

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delite
m__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__
getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__'
, '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem
__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clea
r', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove'
, 'reverse', 'sort']
```

En programmation objet, le __ pour double underscore, se dit « dunder ». Les méthodes dunder sont des « magic methods » associées à des opérations transparentes pour l'utilisateur, par exemple :

- __add__ définit ce que doit faire l'ordinateur quand on somme 2 listes avec l'opérateur « + »
- __repr__ définit ce que doit afficher l'ordinateur quand on tape le nom d'une liste existante et entrer

Les méthodes sans dunder sont les méthodes que l'on peut utiliser sur les listes, comme append etc.

Nous reparlerons de cela à propos de la programmation objet.

1.IV. Taille

Pour obtenir la taille d'une liste, on utilise la commande $len(L)$

```
>>> L = [1,2,3]

>>> len(L)
3
```

On obtient le nombre d'éléments contenus dans L.

On peut donc accéder au dernier élément ainsi :

$$x = L[len(L) - 1]$$

On va le voir juste après, on peut aussi l'obtenir en écrivant :

$$x = L[-1]$$

1.V. Indices/séparateurs

Au départ, vous aurez du mal à utiliser les listes pour une simple raison. Les indices vont de 0 à n-1 pour n termes. Il faudra vous habituer à travailler avec des indices partant de 0.

1.V.1 Indices : récupérer un objet

Prenons l'exemple d'une liste de lettres : $L = ["a","b","c","d","e","f","g","h"]$

Pour accéder à l'un des objets de la liste L, il suffit d'écrire $L[i]$ où i est l'indice de l'objet recherché

Le tableau ci-dessous résume les indices positifs et négatifs associés aux éléments de L :

Indice positif	0	1	2	3	4	5	6	7
Objet	$"a"$	$"b"$	$"c"$	$"d"$	$"e"$	$"f"$	$"g"$	$"h"$
Indice négatif	-8	-7	-6	-5	-4	-3	-2	-1

Ainsi :

```
>>> L = ["a","b","c","d","e","f","g","h"]
>>> L[0]
'a'
>>> L[-8]
'a'
```

On pourra accéder simplement au dernier terme d'une liste en écrivant :

$$L[-1] \text{ ce qui est identique à } L[len(L) - 1]$$

Il est aussi possible d'accéder à une portion de la liste en créant une nouvelle liste extraite de la première.

Remarque : On peut créer une liste de taille donnée ne comportant pas pour autant de valeurs à l'aide de ***None***. ***None*** est un objet qui « n'est rien ». Ainsi, on peut appeler un terme d'une liste sans pour autant recevoir de message d'erreur si le terme n'existe pas. Avec None, python retourne « None », ou « rien ».

Exemple :

```
>>> L=[None,None,None]
>>> L[1]
```

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

1.V.2 Séparateurs et slices : récupérer une portion de liste

Si on veut récupérer les 3 premiers termes, c'est-à-dire de l'indice 0 à l'indice 2 inclus, il faut écrire :

```
>>> L[0:3]
['a', 'b', 'c']
```

Cela peut paraître perturbant, car on aurait voulu logiquement écrire $L[0:2]$. Toutefois, on peut retenir que l'on demande tous les termes entre les « séparateurs » (mot personnel définissant toute séparation d'un objet d'autre chose) précisés. Le tableau ci-dessous présente les numéros des séparateurs en indices positifs et négatifs entre chaque objet :

	0	1	2	3	4	5	6	7	8
Objet	"a"	"b"	"c"	"d"	"e"	"f"	"g"	"h"	
	-8	-7	-6	-5	-4	-3	-2	-1	Rien !

Ainsi, on a :

```
>>> L[2:4]
['c', 'd']

>>> L[-6:-4]
['c', 'd']
```

Attention, prendre les indices dans le mauvais sens conduit à un résultat vide :

```
>>> L[4:2]
[]

>>> L[-4:-6]
[]
```

Il est possible de demander tous les termes à partir d'une position ainsi :

```
>>> L[-6:]
['c', 'd', 'e', 'f', 'g', 'h']

>>> L[:3]
['a', 'b', 'c']

>>> L[:-5]
['a', 'b', 'c']

>>> L[3:]
['d', 'e', 'f', 'g', 'h']
```

On remarquera que la liste sera toujours prise dans le sens gauche -> droite lorsque l'on utilise le : seul ($L[:-3]$)

Attention : Comme le séparateur 0 en négatif n'existe pas, il est nécessaire d'écrire L[-1 :] pour récupérer le dernier terme. Ou plus simplement, L[-1]…

```
>>> L[-1:0]
[]
```

Enfin, et cela sera utile lorsque nous étudierons la complexité d'algorithmes de tri par exemple, écrire L[:i] pour créer une sous liste de L jusqu'au séparateur i, est de complexité $O(i)$.

1.VI. Récupération des plusieurs valeurs

Il faut absolument que vous reteniez que l'on peut facilement récupérer plusieurs valeurs dans une liste pour les affecter à des variables. Exemples : si `mot = [1,0,0,0,0,1,0]`

HORRIBLE	BIEN

`d1 = mot[0]`
`d2 = mot[1]`
`p1 = mot[2]`
`d3 = mot[3]`
`p2 = mot[4]`
`p3 = mot[5]`
`p4 = mot[6]`
	`d1,d2,p1,d3,p2,p3,p4 = mot`

Si on ne veut récupérer qu'une partie de liste : `d2,p1,d3 = mot[1:4]`

1.VII. Opérations

Vous pourrez utiliser les quelques opérations suivantes :

$L.remove(objet)$	Retire la première occurrence de l'objet en question de la liste en partant de la gauche
$L.reverse()$	Inverse le sens des termes de la liste
$L.count(objet)$	Retourne le nombre de fois où l'objet apparaît
$L.index(objet)$	Retourne l'index de la première occurrence de l'objet
$L.extend(LL)$ $L + LL$	Ajoute la liste LL à la liste L
$L.insert(i,x)$	Ajoute x au niveau du séparateur numéro i
$del\ L[i]$	Supprime le terme d'indice i de L, soit le $(i+1)°$ terme

1.VIII. Echange de deux valeurs

Les deux codes suivants sont équivalents :

```
L = [1,2]
a = L[0]
L[0] = L[1]
L[1] = a
print(L)
```

```
L = [1,2]
L[0],L[1]=L[1],L[0]
print(L)
```

On retiendra donc :

```
L[i],L[j]=L[j],L[i]
```


cc BY NC SA

1.IX. Fonction range

1.IX.1 Pour créer une liste « par compréhension »

La fonction $range(n)$ est très intéressante car elle génère une boucle invisible permettant de créer des listes assez complexes.

```
>>> L=[i for i in range(10)]
```

On écrit par exemple :

```
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Concrètement, cela veut dire : « créer la liste L contenant les différentes valeurs de i pour i dans la plage contenue entre les **séparateurs** 0 et n ». Elle contient n termes pour i allant de 0 à n-1

On peut comme cela créer une liste de nombres paires par exemple :

```
>>> L=[2*i for i in range(10)]

>>> L
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Ou encore n fois le même élément :

```
>>> L = [1 for i in range(10)]

>>> L
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Ou enfin, quelque chose qui revient souvent, créer une liste des termes entre 0 et a avec n intervalles (pas si trivial):

```
>>> a = 10

>>> n = 8

>>> L = [i/n for i in range(n+1)]

>>> L
[0.0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.0]
```

Remarques :

- On peut prendre les termes

du séparateur i au séparateur j : $range(i,j)$	entre deux séparateurs en précisant un pas : $range(i,j,p)$

`>>> L = [i for i in range(2,5)]`
 	
`>>> L`
`[2, 3, 4]`
	`>>> L = [i for i in range(10,20,2)]`
	
	`>>> L`
	`[10, 12, 14, 16, 18]`

- On peut aussi directement créer une liste avec range, auquel cas pour afficher les termes, il faut utiliser la commande list(L) : (mais L n'est pas vraiment une liste, le append ne fonctionne pas par exemple)

```
>>> L = range(0,10)

>>> L
range(0, 10)

>>> list(L)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> L = list(range(10))

>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Attention, quelle que soit la forme utilisée, les arguments de range doivent être des nombres de type int (entiers).

Il est possible de parcourir les nombres en sens inverse : range(10,-1,-1) va de 10 à 0

Enfin : $range(0,n) = range(n)$ donc évitez de mettre 0.

1.IX.2 Pour parcourir une liste

Il est possible de parcourir les éléments d'une liste L en écrivant $for\ t\ in\ L$, par exemple :

```
L = [i for i in range(0,11,2)]
LL = [2*t for t in L]
```

```
>>> L
[0, 2, 4, 6, 8, 10]

>>> LL
[0, 4, 8, 12, 16, 20]
```

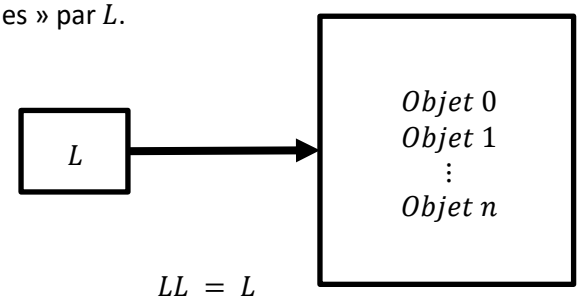
Attention dans les boucles for à l'utilisation de cette commande for i in L à ne pas modifier L ! (cf paragraphe sur la boucle for)

1.X. Copie de listes

Il est très important de faire attention à la copie de listes.

En effet, une liste L est en fait un ensemble de données en mémoire. Taper L consiste à aller chercher en mémoire les données « pointées » par L.

Lorsque l'on écrit :



$$LL = L$$

On croit créer une nouvelle liste LL, copie de L.

Essayons donc de le faire, et modifions une valeur de la nouvelle liste :

```
>>> L = [1,2,3,4,5]

>>> LL = L

>>> LL
[1, 2, 3, 4, 5]
```

Maintenant que nous disposons de deux listes, modifions la nouvelle liste LL :

```
>>> LL[1]=0

>>> LL
[1, 0, 3, 4, 5]
```

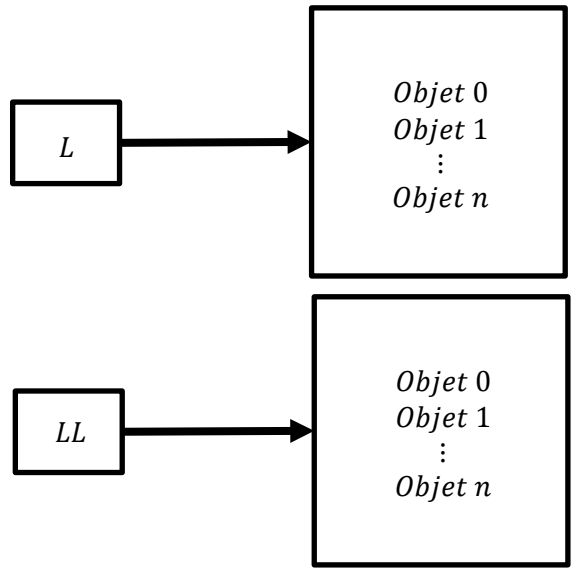
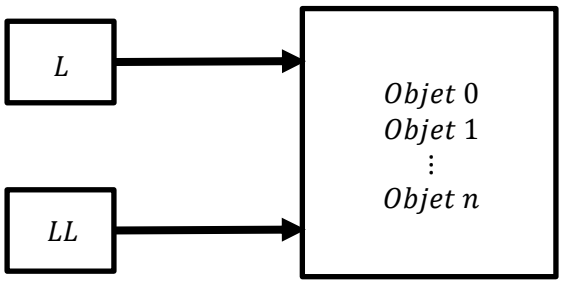
Et maintenant, regardons ce que vaut L :

```
>>> L
[1, 0, 3, 4, 5]
```

OUPS

Il faut faire très attention à cela !

Lorsque l'on écrit $LL = L$:

On espère	On obtient
	

Remarque : ce n'est pas le cas des variables qui elles sont indépendantes.

Il existe des solutions simples, en faisant une itération sur chaque terme et en l'affectant dans une nouvelle liste vide par exemple.

Sinon, on peut écrire :

```
>>> L = [1,2,3,4,5]
>>> LL=L.copy()
>>> LL[1]=0
>>> LL
[1, 0, 3, 4, 5]
>>> L
[1, 2, 3, 4, 5]
```

```
>>> L = [1,2,3,4,5]
>>> LL = L[:]
>>> LL[1]=0
>>> LL
[1, 0, 3, 4, 5]
>>> L
[1, 2, 3, 4, 5]
```

Remarque : il n'y a pas de problème si l'on crée une liste qui n'est qu'un morceau d'une autre liste (comme ci-dessus exemple de droite) :

```
L = [i for i in range(10)]
LL = L[2:4]
LL[0] = 100
print(L)
```

```
>>> (executing file "<tmp 1>")
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

1.XI. Listes de listes

1.XI.1 Création

Il est possible de créer des listes de listes de listes…, pour cela rien de plus simple :

- Créer des listes L1, L2,…,Ln, et écrire : `L=[L1,L2,…,Ln]`
- Ou le faire directement : `L = [[1,2],[3,4,5],…,[6,7,8,9]]`

1.XI.2 Taille

On peut récupérer le nombre de listes dans une liste L de listes Li en écrivant tout simplement :

```
len(L)
```

On peut récupérer la taille d'une sous liste Li de L en écrivant par exemple :

```
len(L[i])
```

1.XI.3 Récupération de valeurs

Soit une liste L de listes Li. On peut récupérer :

- L'une des listes Li de L en écrivant L[i], où i est l'indice de la liste à récupérer
- Un terme d'une des sous listes en écrivant L[i][j] où j est l'indice du terme dans la liste Li d'indice i dans L
  Attention : Ne pas écrire L[i,j], cela ne fonctionne pas avec Python et les listes. Cela fonctionnera avec les array, abordés plus tard dans le cours.

```
>>> L = [[1,2,3],[4,5],[6,7,8,9]]

>>> L[1][1]
5
```

- Une ligne complète en écrivant simplement : `Ligne = L[i]`
- On ne peut pas simplement récupérer une colonne ! Il faudra par exemple écrire :

```
[L[i][0] for i in range(len(L))]
```

Tout ceci s'étend à toute liste de listes de listes…….

1.XII. Modification d'un objet dans une liste

On peut stocker ce que l'on veut dans une liste. Par exemple, stockons une liste dans une liste :

```
L = []
A = [1,1]
L.append(A)
```

Lorsque l'on modifie ensuite A, on modifie l'objet A.

```
A += [1]
print(L)
```

```
[[1, 1, 1]]
```

Mais attention, si l'on écrit A = A + au lieu de A+=, il se passe autre chose :

```
A = A + [1]
print(L)
```

```
[[1, 1]]
```

Dans ce cas, un nouveau A est créé, qui pointe vers une autre adresse mémoire que l'ancien, présent dans L. La valeur dans L ne sont donc pas modifiées.

Conclusion :

Lorsque l'on stocke un objet dans une liste, la MODIFICATION de cet objet s'opère dans la liste, qui en réalité pointe vers cet objet !

Voici des exemples que vous pourriez bien rencontrer et faire des erreurs :

```
>>> L = [[]]*5

>>> L[0].append(1)

>>> L
[[1], [1], [1], [1], [1]]
```

```
>>> Ligne = [0,0,0,0]

>>> Table = [Ligne for _ in range(3)]

>>> Ligne[0] = 1

>>> Table
[[1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0]]
```

```
>>> Ligne = [0,0,0,0]

>>> Table = [Ligne for _ in range(3)]

>>> Table[0][0] = 1

>>> Table
[[1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0]]
```

En ajoutant une liste n fois, c'est n fois la même liste qui est ajoutée. La modification d'un terme de cette liste modifie donc toutes les listes…

Voici les solutions en ajoutant de nouvelles listes à chaque fois :

```
>>> L = [[] for _ in range(5)]

>>> L[0].append(1)

>>> L
[[1], [], [], [], []]
```

```
>>> Table = [[0,0,0,0] for _ in range(3)]

>>> Table[0][0] = 1

>>> Table
[[1, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```


CC BY NC SA

Remarque : je n'ai pas la liste des types d'objets pour lesquels ce comportement est rencontré. Si A est un entier ou un flottant par exemple, pas de soucis (cf exemple ci-dessous). Mais si A est une liste ou un array, on rencontre ce comportement.

```
>>> A = 1

>>> L = [A]

>>> A+=1

>>> L
[1]
```

1.XIII. Comparaison de deux listes

Le test d'égalité de deux listes fonctionne :

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

>>> A==[1,2]
True
```