Les listes

Nous pourrions faire plusieurs pages d'explication sur les listes, mais c'est un objet très intuitif que nous préférons présenter par l'exemple.

Si l'on veut stocker l'ensemble des noms des élèves d'une classe, les températures du mois passé ou encore les plaques d'immatriculation d'un parc de véhicules de location, il nous faut une structure nous permettant de stocker plusieurs informations.

Il faut aussi que cette structure puisse évoluer dans le temps car un nouvel élève peut arriver, il faut alors ajouter son nom dans la liste. Un autre élève peut déménager, et dans ce cas, il faut retirer son nom de la liste. On a pu faire une erreur de saisie et un nom a pu être mal orthographié, il faut alors pouvoir le modifier sans toucher aux autres noms présents. Les listes sont faites pour offrir toutes ces fonctionnalités que nous allons présenter dans la suite.

1. Créer une liste

Nous créons une liste vide en écrivant un symbole crochet ouvrant suivi d'un symbole crochet fermant. Pour créer une liste à partir d'un échantillon de valeurs, nous séparons ces différentes valeurs par des virgules et nous les encadrons par une paire de crochets. Voici un exemple :

```
L = []                  # Liste vide
M = [ 11, 12, 15]       # Liste avec trois éléments
print(M)                # la fonction print accepte les listes !
>> [11, 12, 15]
```

Sous Mac, les symboles crochets ne sont pas présents sur le clavier. Il faut utiliser une combinaison de touches : [Alt] [Maj] (.

Le langage Python étant très souple, il vous permet de stocker des valeurs de tout type dans une même liste. Ainsi, vous pouvez écrire `L = [ "Bonjour",7, 3.14]`. Ce n'est pas parce que c'est possible que c'est une bonne idée. En effet, cette pratique n'est absolument pas encouragée, car c'est le meilleur moyen d'avoir des soucis. De toute façon, si vous créez une liste contenant des types hétérogènes, hormis si vous voulez faire juste un affichage, dans les autres cas, il faudra traiter chaque élément à part pour lui appliquer un traitement propre à son type et cela deviendra vite assez complexe.

Bonne pratique : créez des listes contenant un seul type de données.

2. Lire et modifier une liste

À tout moment, nous pouvons lire ou modifier une valeur de la liste en utilisant les symboles crochets [ ] contenant l'indice de l'élément considéré dans la liste. Attention, le premier élément d'une liste est positionné tout à gauche et correspond à l'élément d'indice 0. De ce fait, le dernier élément d'une liste de n éléments a pour indice n-1. La fonction `clear()` appliquée depuis une liste permet de retirer l'ensemble des éléments.

Retenez ceci : le premier élément d'une liste se situe à l'indice 0.

Voici un exercice d'entraînement, essayez de trouver les résultats :

```
L = [4,15,7]
print(L)
print(L[2])
L[0] += 1
print(L)
L[2] += 9
print(L)
L.clear()
print(L)
```

Voici l'affichage du programme :

```
>> [4, 15, 7]
>> 7
>> [5, 15, 7]
>> [5, 15, 16]
>> []
```

Pour désigner le dernier élément d'une liste, on peut écrire l'indice -1 qui correspond au premier élément de la liste en partant de la droite. L'indice -2 correspond au deuxième élément en partant de la droite, et ainsi de suite :

```
L = [4,14,7]
print(L[-1])
print(L[-2])

>> 7
>> 14
Fusion
```

3. Fusionner des listes

Il est possible de fusionner les éléments de deux listes pour former une nouvelle liste grâce à l'opérateur + appliqué entre deux listes. Voici un exemple :

```
L = [4,15,7]
M = [1,2]
R = L + M
print(L)
print(M)
print(R)
```

Ce qui donne après l'exécution :

```
>> [4,15,7]
>> [1,2]
>> [4, 15, 7, 1, 2]
```

4. Insérer et supprimer un élément

On peut ajouter un élément en fin de liste en utilisant la syntaxe : `L.append(element)`. On peut aussi insérer un élément à une position précise en utilisant la syntaxe : `L.insert(position, element)`. De manière similaire, on peut retirer l'élément en fin de liste en utilisant la commande `L.pop()` et retirer un élément à une position donnée en utilisant : `L.pop(position)`. Voici un exemple :

```
L = [4,15,7]
L.append(8)
print(L)
L.insert(1,5)
print(L)
L.pop(2)
print(L)
L.pop()
print(L)
```

Ce qui donne comme résultat :

```
>> [4, 15, 7, 8]
>> [4, 5, 15, 7, 8]
>> [4, 5, 7, 8]
>> [4, 5, 7]
```

Dans le langage Python, les fonctions qui insèrent/retirent un élément en fin de liste sont optimisées et sont plus rapides que lorsqu'on doit faire une insertion/suppression à l'intérieur de la liste. En effet, une liste est gérée en mémoire comme une pile de livres sur votre bureau. Si vous rajoutez ou si vous retirez un livre en haut de la pile, c'est facile. Si vous devez retirer un livre au milieu de la pile, ou en rajouter un au même endroit, cela devient plus compliqué !

5. Extraire une sous-liste

Comme nous l'avons vu pour les chaînes de caractères, il est possible d'extraire une partie d'une liste L en utilisant la syntaxe : `L[a:b]`. Ainsi, nous retrouvons les mêmes règles que pour les chaînes de caractères, mais appliquées aux listes :

- L'indice 0 indique le premier élément de la liste en partant de la gauche, 1 le deuxième, et ainsi de suite.
- L'indice -1 indique le dernier élément de la liste en partant de la droite, -2 le deuxième, et ainsi de suite.
- L'écriture `L[a:b]` correspond à une sous-liste commençant à l'élément d'indice `a` et allant jusqu'à l'élément d'indice `b` non compris.
- L'écriture `L[a:]` correspond à une sous-liste commençant à l'élément d'indice `a` et allant jusqu'à la fin de la liste.
- L'écriture `L[:b]` correspond à une sous-liste commençant au premier élément situé à gauche et allant jusqu'à l'élément d'indice `b` non compris.

Voici quelques exemples :

```
L = [ 0, 10, 20, 30, 40, 50, 60, 70, 80 ]
print(L[0:2])
print(L[2:-2])
print(L[3:])
print(L[:3])
```

```
>> [0, 10]
>> [20, 30, 40, 50, 60]
>> [30, 40, 50, 60, 70, 80]
>> [0, 10, 20]
```

6. Tester la présence et le nombre d'occurrences d'un élément

Nous pouvons déterminer si un élément `x` appartient à une liste `L` en utilisant la condition `x in L`. Nous pouvons aussi déterminer l'indice de cet élément dans la liste avec la commande : `L.index(x)`.

```
L = [4,15,7]
print(4 in L)
print(L.index(4))
```

Ce qui donne :

```
>> True
>> 0
```

La fonction `count()` appliquée depuis une liste permet de compter le nombre d'occurrences d'un élément dans une liste. Ainsi, nous avons :

```
L=[0, 1, 1, 2, 2, 2, 7]
print(L.count(1))
print(L.count(2))
print(L.count(10))
>> 2
>> 3
>> 0
```

7. Les fonctions natives appliquées aux listes

Nous présentons les fonctions natives pouvant s'appliquer aux listes :

- `len()` permet de connaître le nombre d'éléments d'une liste en lui donnant en argument la liste en question. La valeur retournée est un nombre entier et vaut 0 pour une liste vide.
- `min()` et `max()` acceptent aussi une liste en argument et retournent la valeur minimale/maximale des éléments de

la liste.

Voici un exemple :

```
L = [ 7, -4 , 6]
print(len(L))
print(min(L))
print(max(L))
```

Ce qui donne :

```
>> 3
>> -4
>> 7
```

8. Parcourir une liste avec une boucle for

Il est souvent nécessaire de parcourir les éléments d'une liste, par exemple pour les afficher. Le langage Python possède une syntaxe très simple et pratique permettant de traiter les éléments d'une liste un à un en utilisant une boucle `for` : `for element in liste` :.

Nous n'utilisons plus un nombre entier comme indice de parcours, mais directement les éléments de la liste. Ainsi, à chaque itération, la variable de boucle désigne successivement chaque élément de la liste. Voici un exemple :

```
L = [4,6,9]
for v in L :
 print(v+10)

>> 14
>> 16
>> 19
```

Nous remarquons que le code précédent est équivalent à celui-ci :

```
L = [4,6,9]
for i in range(len(L)) :
 print(L[i]+10)
```

Ce code utilisant une boucle `for` avec indice est équivalent à la version précédente. Cependant, dans ce cas où l'indice n'a pas de rôle dans le programme, nous préférons l'écriture avec la boucle `for` sur les éléments de la liste car elle est plus compacte, plus simple et surtout plus lisible.

9. Copier une liste

Sous une apparence simple se cachent beaucoup de difficultés derrière cette action. En effet, en copiant des listes,

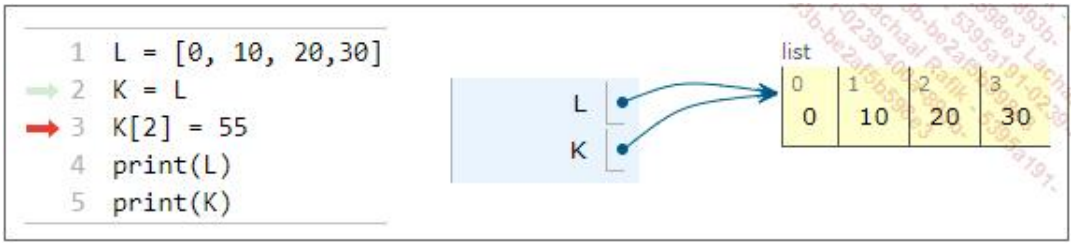
ou en dupliquant des listes, vous entrez dans une zone compliquée de la programmation qui nécessite de nombreuses explications pour arriver à maîtriser la situation. Nous présenterons ces questions dans le chapitre Les fonctions. Nous allons pour l'instant vous présenter quelques cas simples vous permettant de connaître l'existence de ces configurations problématiques et de les traiter. Voici le premier cas :

```
L = [0, 10, 20,30]
K = L
K[2] = 55
print(L)
print(K)
```

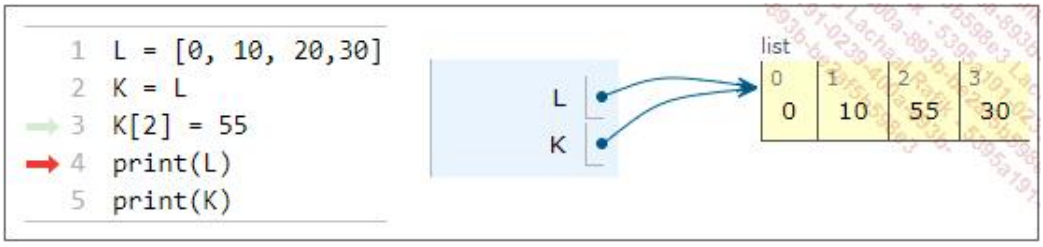
Quel va être le résultat de cet affichage ? Si vous ne connaissez pas exactement les règles qui se cachent sous ce mécanisme, n'essayez pas de deviner, c'est impossible. Normalement, sans trop savoir, on répondrait L = [0,10,20,30] et K = [0,10,55,30]. Or, voici le résultat de l'affichage :

```
[0, 10, 55, 30]
[0, 10, 55, 30]
```

Que s'est-il passé ? En fait, l'écriture K=L est trompeuse, car elle peut faire croire qu'une deuxième liste est créée et qu'elle contient une copie à l'identique des éléments de L. Non. Tout ce qu'a fait l'interpréteur Python à cet instant, c'est de créer une nouvelle variable K et de l'associer à la même liste que celle associée à la variable L. Examinons le schéma de la mémoire en utilisant Python Tutor. Ici, les flèches en bleu désignent des associations :



Les variables K et L désignent la même liste. Ainsi, en écrivant L[2] ou K[2], on accède exactement au même élément. Et lorsque l'on écrit L[2] = 55 ou K[2] = 55, on applique exactement la même modification à l'élément de la liste :



Si nous voulons cependant créer une nouvelle liste identique à la première mais totalement indépendante, nous devons utiliser une syntaxe spéciale : K = L[:] ou K = L.copy()

Voici le résultat :

```
L = [0, 10, 20,30]
```

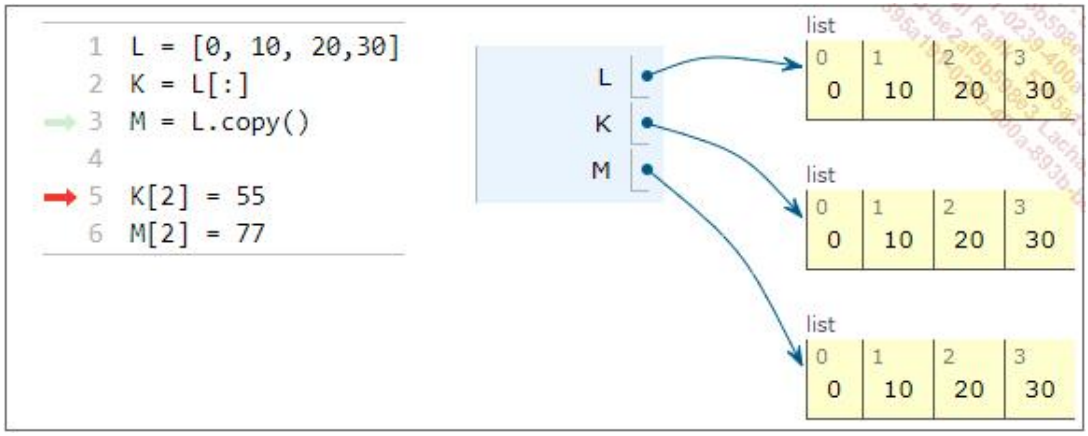
```
K = L[:]
M = L.copy()

K[2] = 55
M[2] = 77

print(L)
print(K)
print(M)

>> [0, 10, 20, 30]
>> [0, 10, 55, 30]
>> [0, 10, 77, 30]
```

Voici l'état de la mémoire après appel de la fonction `copy()` et de la syntaxe `[:]` :



Maintenant, les variables `K` et `M` sont associées à des listes contenant les mêmes éléments que la liste `L` mais dans des listes indépendantes. Mission réussie !