Informatique en CPGE (2018-2019)
Les listes

1 Rappels sur les listes

Les objets de type **list**, que nous appelons des listes, sont étudiés en première année. Ils sont très souvent utilisés et il est bon de faire quelques rappels.

1.1 Définition

Une liste est un ensemble ordonné d'éléments éventuellement hétérogènes dont les valeurs peuvent être modifiées.

Les éléments d'une liste sont séparés par des virgules et entourés de crochets.

1.2 Création d'une liste

```
liste1=[]  # une liste vide
# ou liste1=list()
liste2=['a']  # une liste contenant un unique élément
liste3=['a','bonjour',17]
liste4=['d','e']
liste3[1]='b'  # modification d'un élément
print(liste3) # affiche ['a','b',17]
liste5=liste3+liste4
print(liste5) # affiche ['a','b',17,'d','e']
liste6=3*liste4 # soit ['d','e','d','e','d','e']
```

La fonction **list()** permet de convertir certains objets en listes. Nous pouvons aussi l'utiliser avec la fonction **range** pour initialiser une liste d'entiers.

```
liste=list('bonjour') # ['b','o','n','j','o','u','r']
liste1=list(range(4)) # [0,1,2,3]
liste2=list(range(1,4)) # [1,2,3]
liste3=list(range(2,14,3)) # [2,5,8,11]
```

Construction par compréhension

```
liste=[2*x-1 for x in range(1,5)] # construit [1,3,5,7]
```

Nous pouvons construire une autre liste en itérant sur les éléments d'une liste.

```
liste2=[2*x for x in liste] # construit [2,6,10,14]
```

Copie d'une liste
Pour créer une nouvelle liste, copie d'une liste existante, le code est le suivant :

```
liste1=[0,2,4,6,8]
liste2=list(liste1)
# ou liste2=liste1[:]
```

Attention : ce code peut poser problème si les éléments de la liste sont eux-mêmes des listes.

```
liste1=[[0,1],[2,3],[4,5]]
liste2=list(liste1)
liste2[1][0]=8 # modifie aussi liste1
print(liste1) # affiche [[0,1],[8,3],[4,5]]
```

Pour éviter cela, nous devons plutôt écrire :

```
liste1=[[0,1],[2,3],[4,5]]
liste2=[list(x) for x in liste1]
```

Le module **copy** propose la fonction **deepcopy** pour effectuer une vraie copie en profondeur.

```
from copy import deepcopy
liste1=[[0,1],[2,3],[4,5]]
liste2=deepcopy(liste1)
```

Insertion et extraction
La méthode **append** permet d'ajouter un élément à la fin d'une liste.

```
liste=['a','b']
liste.append('c')  # (liste=['a','b','c'])
```

La méthode **insert** permet d'insérer un objet dans une liste.

```
liste=['a','b','d','e']
liste.insert(2,'c') # l'élément 'c' est inséré à l'index 2
print(liste) # affiche ['a','b','c','d','e'],
# 'd' et 'e' sont décalés vers la droite
```

La syntaxe pour extraire une sous-liste est la suivante :

```
liste=['a','b','c','d','e','f']
liste2=liste[1:4] # liste2 est la liste ['b','c','d']
# liste[-1] est le dernier élément soit 'f'
```

Suppression d'un élément

Pour supprimer et récupérer un élément d'une liste, nous utilisons la méthode **pop** qui supprime l'élément dont l'indice est passé en paramètre et le renvoie.

```
liste=['a','b','c','d']
x=liste.pop(2)  # l'élément d'indice 2 est affecté dans x
#  et il est supprimé de la liste
print(liste) # affiche ['a','b','d']
print(x) # affiche 'c'
```

La valeur par défaut du paramètre de la fonction **pop**, (l'indice), est -1. Donc à l'exécution de l'instruction `liste.pop()`, l'élément en fin de liste est supprimé et renvoyé.

La méthode **remove** permet de supprimer un élément de valeur donnée.

```
liste=['a','b','c','d','c','e']
liste.remove('c')  # l'élément 'c' est supprimé
# seul le premier rencontré !
print(liste) # affiche ['a','b','d','c','e'],
# 'd', 'c' et 'e' sont décalés vers la gauche
```

2 Compléments sur les listes

Le type **list** est un objet central, présent dans de nombreux programmes. Il est important de bien comprendre sa manipulation.

2.1 Implémentation en mémoire

Nous allons schématiser ce qui se passe lorsque nous créons une liste.

De manière simple, l'instruction `L=[...]` crée un objet de type **list**, sous la forme d'un nom (ou identificateur) et d'une adresse en mémoire. A cette adresse, nous trouvons le nombre d'éléments de la liste et les adresses de ces éléments. Si ces éléments sont eux-mêmes des listes, alors à leurs adresses respectives, nous trouvons le nombre d'éléments composant chaque sous-liste et les adresses de ces éléments. Et ainsi de suite.

Voici un exemple avec `L1=[[257,258],[259,260]]`.

Nom	adresse
L1	51967800

adresse	nombre d'éléments	adresses des éléments	
51967800 :	2	51740424	51738384
51740424 :	2	51954224	51953888
51738384 :	2	51953904	51956848

L'adresse 51953888 par exemple est l'adresse du nombre 258.

2.2 Copie d'une liste

Nous allons voir différentes façons de copier une liste.

Commençons par écrire l'instruction `L2=L1`. Cette instruction associe le nom L2 avec l'adresse en mémoire de L1.

Nom	adresse
L2	51967800

L1 et L2 ont simplement une adresse commune en mémoire et donc si nous modifions avec une instruction ce qui se trouve à cette adresse, ou aux adresses suivantes, alors les deux listes L1 et L2 sont modifiées. Ainsi, les instructions `L1[0]=300` et `L2[1][1]=270` modifient aussi bien L1 que L2.

Attention : les instructions `L1.append(301)` et `L1+=[301]` modifient les deux listes puisque c'est encore le contenu de l'adresse commune qui est modifié.

Par contre l'instruction `L1=L1+[301]` ne modifie pas L2. Une instruction du type `L1=...` modifie l'adresse de L1 qui devient alors distincte de celle de L2.

Reprenons la liste initiale L1 et écrivons l'instruction `L2=L1[:]`. Ceci revient à copier les adresses des éléments de L1. Donc L2 a une adresse distincte de L1. Mais les éléments de L2 ont les mêmes adresses que les éléments de L1.

Autrement dit, L1 et L2 ont des adresses différentes mais nous trouvons à ces deux adresses les mêmes informations.

Nom	adresse
L2	51615424

adresse	nombre d'éléments	adresses des éléments	
51615424 :	2	51740424	51738384
51740424 :	2	51954224	51953888
51738384 :	2	51953904	51956848

L'instruction `L2[1][1]=270` modifie le contenu de l'adresse 51738384 qui est commune aux deux listes. Donc la modification est aussi effective pour L1.

Par contre, l'instruction `L2[0]=300` modifie l'adresse du premier élément de L2, c'est-à-dire le contenu de l'adresse 51615424 propre à L2. Donc L1 n'est pas modifiée. Et il en est de même avec les instructions `L2.append(301)` et `L2+=[301]` qui ne modifient pas L1.

Si nous utilisons la méthode **copy** avec l'instruction `L2=L1.copy()`, alors le comportement est le même qu'avec l'instruction `L2=L1[:]`.

Le module `copy` présent dans la bibliothèque standard propose une fonction **copy** et une fonction **deepcopy**. La fonction **copy** a les mêmes caractéristiques que la méthode **copy** utilisée précédemment. Nous pouvons parler, comme avec l'instruction `L2=L1[:]`, de "copie superficielle".

Par contre, avec la fonction **deepcopy**, la copie est effectuée, comme son nom l'indique, "en profondeur". Donc L1 et L2 ont chacune une adresse différente où nous trouvons les adresses de leurs éléments. Si ces éléments sont des listes, alors les éléments de L1 et les éléments de L2 ont des adresses différentes, et ainsi de suite. Donc une modification sur une des deux listes n'a aucune incidence sur l'autre liste.

Note : la différence entre une copie superficielle et une copie en profondeur n'existe que pour les objets composés comme des listes de listes.

2.3 Ajout d'un élément ou d'une liste

Ajout d'un élément

Avec l'instruction `L.append(x)`, l'adresse de L n'est pas modifiée, c'est son contenu qui l'est. Il en est de même avec l'instruction `L+=[x]`. Dans les deux cas, l'ajout de l'élément se fait à peu près en temps constant.

Par contre, avec l'instruction `L=L+[x]`, l'adresse de L est modifiée, c'est une nouvelle liste qui est créée. La complexité en est grandement affectée et pour des listes "longues", cette instruction est à éviter. La complexité est ici linéaire en la longueur de la liste.

Notons que l'instruction `a+=x` est un simple raccourci d'écriture de `a=a+x` si a est un nombre par exemple. La vitesse d'exécution reste inchangée contrairement à ce qui se passe avec d'autres langages de programmation. Par contre, l'instruction `L+=[x]` est un véritable raccourci de `L=L+[x]` qui permet une nette accélération de l'exécution.

Ajout de plusieurs éléments

Si L et x sont des objets de type **list**, alors les instructions `L.extend(x)` et `L+=x` ont une complexité linéaire en la longueur de x. Attention, ce sont les adresses des éléments de x qui sont copiées. Donc, si x est une liste de listes, une instruction comme `x[0][0]=...` modifie x mais modifie aussi L.

Par contre l'instruction `L=L+x` n'est pas adaptée si L a une taille importante, car sa complexité est linéaire en la longueur de `L+x`.

Note : si x est un objet itérable, les instructions `L.extend(x)`, et `L+=x` s'exécutent sans problème (l'objet x est converti automatiquement en une liste). Par contre, il faut écrire la conversion avec l'instruction `L=L+list(x)`.

2.4 Effets de bord

Considérons le code qui suit.

```
def f(u):
    u=u+[1]
    return u

def g(u):
    u+=[1]
    return u
```

Ces deux fonctions se ressemblent. Pourtant, si L = [0], l'appel `f(L)` renvoie la liste [0, 1] et L n'est pas modifiée, alors que l'appel `g(L)` renvoie aussi la liste [0, 1], mais L est modifiée (L = [0 ,1]).

Explication : en ce qui concerne la fonction **f**, l'instruction `u=u+[1]` crée une variable locale **u** à une adresse différente de celle passée en paramètre qui est alors utilisée pour évaluer la valeur de cette variable locale. Pour la fonction **g**, l'instruction `u+=[1]` ne crée pas de nouvelle variable et la modification a lieu sur la variable dont l'adresse est passée en paramètre, donc sur L.

Nous disons qu'une fonction a un effet de bord si elle modifie autre chose que la valeur renvoyée.