Listes

1 Présentation

Une **liste** est une collection de valeurs qui sont modifiables. Elle est notée entre crochets. Par exemple : `[2,2.,1+i,"deux"]` est une liste de 4 éléments en Python. Comme cet exemple le montre, les listes ne sont pas forcément de types homogènes.

Il existe aussi la liste vide : `[]`

2 Construction d'une liste

Plusieurs façons de définir une liste sont envisageables sous Python :

2.1 Construction explicite

La façon la plus simple de définir une liste est d'utiliser une affectation explicitement. Par exemple :

```
L1=[2,4,8,16,32,64,128]
```

2.2 Construction par concaténation

Cette opération consiste en l'assemblage d'une liste par morceaux :

```
L2=L1+[256,512]
```

Dans la même logique, l'operation * assemble plusieurs fois le même morceau. Par exemple :

```
>>> print(6*[0,1])
[0,1,0,1,0,1,0,1,0,1,0,1]
```

2.3 Construction par compréhension

Enfin il est possible de construire une liste par compréhension. C'est une méthode qui consiste à définir une liste à l'aide d'une expression dépendante d'une variable qui parcourt un itérable.

La syntaxe générale est à adapter suivant le contexte :

```
[e(i) for i in range(n)]
```

ou

```
[e(x) for x in L]
```

Par exemple L3=[i%3 for i in range(9)] construit la liste suivante :
L3=[0,1,2,0,1,2,0,1,2]

Application : Qu'est ce qui est stocké dans les listes L4 et L5 suivantes :

```
L4=[i**2 for i in range(6)]
L5=[2*i+1 for i in L4]
```

3 Accès aux éléments d'une liste

3.1 Principes

Le traitement de données nécessite souvent le travail sur certains éléments d'un liste uniquement. Il est donc important de maîtriser comment accéder à un élément ou groupe d'éléments d'une liste. Les éléments d'une liste de longueur n sont numérotés de 0 **à** $n-1$. On appelle **indice** ce numéro (*index* en anglais). Il est aussi possible d'utiliser un **indice négatif** qui correspond à une numérotation des éléments en partant de la fin ($-n$ pour le premier à -1 pour le dernier).

Soit `L` une liste quelconque.

1. `L[p]` renvoie l'élément d'indice p de `L`
2. `L[p:n]` renvoie une nouvelle liste constituée des éléments de `L` d'indice p inclus à n **exclu**
3. `L[p:n:pas]` renvoie une nouvelle liste constituée des éléments de `L` d'indice p inclus à n exclu, tous les pas
4. `L[:]` renvoie une nouvelle liste constituée de tous les éléments de `L`
5. `L[p:]` renvoie une nouvelle liste constituée de tous les éléments de `L` à partir de l'élément d'indice p inclus jusqu'à la fin
6. `L[:n]` renvoie une nouvelle liste constituée de tous les éléments de `L` depuis le premier jusqu'à l'élément d'indice n exclu
7. `L[::pas]` renvoie une nouvelle liste constituée des éléments de `L`, tous les pas

3.2 Exemples

Soit la liste :

```
L=["A","B","C","D","E","F","G","H","I","J","K"]
```

3.3 Affectation d'un élément de liste

L'affectation d'une liste est possible. L'affectation d'un élément ou groupe d'éléments de liste est possible aussi. La syntaxe est la suivante :

```
L1[3]=14
```

Attention, cette instruction est un raccourcis de l'instruction complète :

```
L1=L1[:3]+[14]+L1[4:]
```

Evidemment, si `L1` n'a pas au moins 4 éléments, cette instruction renvoie une erreur.

3.4 Parcours d'une liste

A partir des syntaxes d'accès à un élément de liste, il est possible de parcourir un après l'autre les élément d'une liste. Deux approches sont possibles :

```python
L=[2,4,8,16,32,64,128]
for i in range(len(L)):
    print(L[i])
```

```python
L=[2,4,8,16,32,64,128]
for x in L:
    print(x)
```

4 Fonctions et méthodes agissant sur les listes

Pour modifier des listes, outre les **opérateurs** `+` et `*` vus plus haut, on peut utiliser des fonctions Python qui prendront comme argument une liste, et des méthodes qui agirons sur un objet de type liste.

4.1 Fonctions

Les fonctions ne modifient pas l'argument.

1. `len(liste)` : renvoie la longueur, c'est à dire le nombre d'éléments, d'une liste
2. `min(liste)` : renvoie le plus petit élément d'une liste (dans la mesure où ils sont comparables)
3. `max(liste)` : renvoie le plus grand élément d'une liste (dans la mesure où ils sont comparables)
4. `x in liste` : renvoie `True` si l'élément `x` est présent dans `liste`, `False` sinon
5. `x not in liste` : renvoie `True` si l'élément `x` n'est pas présent dans `liste`, `False` sinon
6. `sorted(liste)` : renvoie une nouvelle liste des éléments de `liste` triés par ordre croissant
7. `del(liste)` : supprime `liste` de la mémoire

4.2 Méthodes

Les méthodes modifient la liste à laquelle elles sont appliquées.

1. L.`append(x)` : ajoute l'élément `x` à la fin de la liste L
2. L.`extend(T)` : concatène la liste `T` à la fin de la liste L
3. L.`remove(x)` : retire l'élément `x` de la liste L, si `x` est présent
4. L.`pop(i)` : retire le $i^{\text{ème}}$ élément et renvoie sa valeur
5. L.`index(x)` : renvoie l'indice de l'élément `x` si `x` est présent
6. L.`insert(i,x)` : insère l'élément `x` à l'indice `i` dans L. Si cet indice dépasse la taille de la liste, l'élément est ajouté à la fin.
7. L.`count(x)` : renvoie le nombre de fois où l'élément `x` est présent dans la liste L
8. L.`reverse()` : renverse l'ordre des éléments de la liste L
9. L.`sort()` : trie les éléments de la liste L par ordre croissant
10. L.`clear()` : vide la liste L de tous ses éléments
11. L.`copy()` : renvoie une copie de la liste L (voir ci-dessous)

4.3 Copier une liste

Une variable de type liste ne contient pas directement de valeurs. C'est plutôt un lien (un index, une adresse) vers le lieu de mémoire où sont les valeurs. L'affectation d'une liste à une autre lie alors les deux listes, ce qui peut alors poser des problèmes :

```
>>> L1 = [1, 2, 3]
[1, 2, 3]
>>> L2 = L1
[1, 2, 3]
>>> id(L1),id(L2)
(200563912, 200563912)
>>> L1.append(4)
>>> print(L2)
[1, 2, 3, 4]
```

Pour créer une nouvelle liste identique à l'originale mais indépendante, il faut utiliser l'une des deux instructions :

```
L2=L1[:]
L2=L1.copy()
```

5 Recherches dans les listes

5.1 Recherche d'un élément dans une liste

Une première problématique consiste à savoir si un élément donné est présent dans une liste. Il n'est pas nécessaire de parcourir tous les éléments de la liste, car on souhaite interrompre le parcours dès qu'un élément est trouvé. Deux solutions sont possibles, en utilisant

une boucle `for` ou une boucle `while`.

```python
def appartient(L,a):
    i=0
    while i<len(L)and L[i]!=a :
        i=i+1
    return i<len(L) # Ceci est un booléen
```

```python
def appartient(L,a):
    for x in L:
        if x==a:
            return True
    return False # Noter l'indentation
```

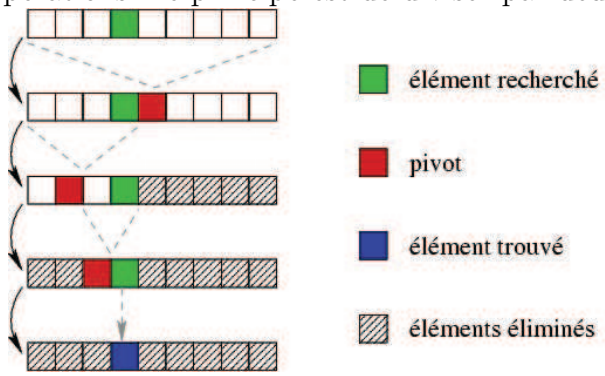
5.2 Recherche d'un maximum

Deuxième problématique : trouver le maximum d'une liste de nombre.

```python
def Le_max(liste):
    maxi=liste[0]
    for i in range(len(liste)) :
        if liste[i]>maxi:
            maxi=liste[i]
    return maxi
```

5.3 Recherche d'un nombre dans une liste triée par dichotomie

Troisième problématique : trouver un nombre prédéfini d'une liste en réduisant le nombre d'opérations. Le principe est de diviser par deux la longueur de la liste à chaque itération.



```python
def is_number_in_list_dicho (nb, liste ):
    g, d = 0, len( liste )-1
    while g <= d:
        m = (g + d) // 2
        if liste [m] == nb:
            return m
        if liste [m] < nb:
            g = m+1
        else:
            d = m-1
    return None
```