Listes et algorithmes de recherches

1 Description des listes et tableaux

Nous avons déjà introduit les types composés lors du cours sur les variables.
Un n-uplet est une collection de valeurs en Python. Aucune de ses valeurs n'est modifiable.
Une liste est une collection de valeurs qui sont modifiables. Elle est notée entre crochets.
Par exemple :
["Pairs",0,2,4,6,8]
est une liste en Python. Comme cet exemple le montre, les listes ne sont pas forcément de types homogènes.

Nous appellerons tableau une liste de nombres (d'entiers ou flottants) ; même si il n'y a pas de vrai consensus sur la différence entre les deux en Python.

2 Constructions d'un tableau

Il y a plusieurs façons de définir une liste en Python :

2.1 Construction explicite

La façon la plus simple de définir une liste ou un tableau est de la définir explicitement. Par exemple :
`L1=[2,4,8,16,32,64,128,1]`
définit bien une liste en Python.

2.2 Construction par concaténation

Une autre manière de construire certaines listes est par concaténation (ajout d'une liste à une autre liste). Cette méthode permet de répéter certains motifs récurrents. Par exemple :
`L2=6*[0,1]`
donne la liste suivante :
`L2=[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 suivant le contexte :

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

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

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

Exercice 1 : Qu'est ce qui est stocké dans les tableaux T1 et T2 suivants :

```
T1=[i**2 for i in range(6)]
```

```
T2=[2*i+1 for i in T1]
```

Exercice 2 : Comment construire en compréhension le tableau T3 contenant `[1,3,7,15,...,65535]` ?

3 Accès aux éléments d'une liste

Les différents éléments d'un tableau ou d'une liste A sont accessibles par A[indice] où l'indice peut prendre les valeurs allant de 0 à N-1 où N est la taille de A.

Comme ces objets sont modifiables, il est possible de modifier directement un élément en utilisant une instruction d'affectation sur cet élément :

Si `A=[0,1,2,0,1,2]`

l'instruction

`A[2]=0`

transformera le tableau A en :

`A=[0,1,0,0,1,2]`

Il est à noter que tenter d'accéder à un élément dont l'indice n'est pas une case valide du tableau renverra une erreur. Ainsi, dans l'exemple précédent l'instruction

`A[6]=0`

renverra :

`IndexError: list assignment index out of range`

Par contre les notations A[-1], A[-2],...,A[-N] désignent la dernière, l'avant dernière case ... du tableau.

Il est possible d'accéder à plusieurs éléments d'un tableau directement. L'expression A[i :j] désigne le tableau constitué des éléments situés entre l'indice i(inclus) et l'indice j(exclu).

Ainsi A[4 :6] est évalué comme [0,1] dans l'exemple précédant.

L'instruction B=A[4 :6] a comme effet de copier le tableau extrait dans un nouveau tableau appelé B. Une modification ultérieure d'éléments de B ne modifiera pas les éléments de A et réciproquement.

Il est également possible d'accéder à un tableau en entier en utilisant la syntaxe A[:].

Ainsi l'instruction B=A[:] crée dans B une copie de A. La modification d'un élément de A ne modifiera pas B et réciproquement. Derrière cet instruction d'apparence simple se cache un processus complexe pour le processeur - qui doit effectuer N opérations de lecture et N opérations d'affectation. Ce processus est donc potentiellement coûteux en temps et en mémoire !

4 Opérations sur les listes

Les listes possèdent plusieurs méthodes très utiles à leur manipulation, et qu'il est nécessaire de connaître pour utiliser ces structures de données efficacement. Pour obtenir une liste de toutes ces fonctions, on pensera à lire la la documentation officielle de Python ou à utiliser help(list) dans la console Python.

Voici un liste de fonctions et méthodes à connaître :

1. `len(liste)`

   Cette fonction renvoie la longueur d'une liste. Noter que les indices des éléments d'une liste L appartiennent à [0,len(L)-1]

   ```
   L=[2,4,6,8]
   print(len(L)) # affiche 4
   ```

2. `list.append(element)`

   Cette fonction sert à ajouter un élément à la fin d'une liste.

   ```
   L = [1, 2, 3, 4, 5]
   L.append(6)
   print(L) # [1, 2, 3, 4, 5, 6]
   ```

3. `list.extend(autre_liste)`
   Cette fonction ressemble beaucoup à list.append ; elle joue le même rôle, si ce n'est qu'elle prend en paramètre une liste et qu'elle va ajouter à la fin de la liste tous les éléments de la liste passée en paramètre.

```
L = [1, 2, 3, 4, 5]
L.extend([6, 7, 8])
print(L) # [1, 2, 3, 4, 5, 6, 7, 8]
```

4. `list.remove(valeur)`
   list.remove retire la première valeur trouvée qui est égale au paramètre. Cela permet de supprimer une valeur dont on ne veut pas dans la liste.

```
L = [1, 2, 3, 1, 2, 3]
L.remove(1)
print(L) # [2, 3, 1, 2, 3]
```

5. `list.reverse()`
   Cette fonction permet de renverser l'ordre des éléments d'une liste, mais sans renvoyer de valeur : en effet, le renversement se fait directement dans la liste sur laquelle on a utilisé la méthode. Si on veut récupérer la valeur, on peut utiliser la fonction reversed(liste) (qui n'est pas une méthode de la classe list).

```
L = [1, 2, 3, 4, 5]
L.reverse()
print(L) # [5, 4, 3, 2, 1]
T = [1, 2, 3, 4]
print(reversed(T)) # [4, 3, 2, 1]
print(T) # [1, 2, 3, 4] ; l'ordre de la liste n'a pas été modifié
```

6. `list.sort()`
   Cette fonction permet de trier une liste dans l'ordre croissant (ou dans l'ordre alphabétique selon le type des données de la liste). Des paramètres additionnels peuvent être donnés à la fonction pour changer le mode de tri et l'ordre dans lequel le tri s'effectue, consultez la documentation Python de la fonction pour cela. De la même façon qu'il existe reversed(list) pour list.reverse(), on a sorted(list) pour list.sort().

```
L = [3, 1, 4, 2, 5]
L.sort()
print(L) # [1, 2, 3, 4, 5]
T = [3, 1, 4, 2, 5]
print(sorted(T)) # [1, 2, 3, 4, 5]
print(T) # [3, 1, 4, 2, 5] ; l'ordre n'a pas été modifié
```

5 Mode de passage des tableaux

L'affectation d'une liste à une autre présente une particularité en Python : la nouvelle liste n'est pas copiée mais les deux noms de variables désignent la même liste. De fait les deux noms de variables sont liés.

Considérons par exemple le code suivant :

```
a=[2,4,6,8]
b=a
b[2]=0
print(b)
print(a)
```

print(b) affichera [2,4,0,8] mais de manière plus surprenante print(a) affichera la même chose !

Tout se passe comme si l'état de la mémoire évoluait de la manière suivante :

a	[2,4,6,8]

a,b	[2,4,6,8]

a,b	[2,4,0,8]

Ainsi ce qu'on a vu pour le mode de passage par valeur dans une fonction n'est plus vrai pour un tableau. Il est donc possible d'échanger deux éléments d'un tableau à l'aide d'une fonction, là où il n'est pas possible de le faire avec deux variables.

6 Parcours de listes

Supposons que l'on souhait additionner tous les éléments d'un tableau d'entiers. Un algorithme simple permettant de le faire consiste à initialiser une variable temporaire à zéro et sommer les valeurs des éléments de ce tableau parcourus à l'aide d'une boucle for :

```
L=[1,3,7,11]
s=0
for i in range(len(L)):
   s=s+L[i]
print(s)
```

Python permet néanmoins de parcourir directement la liste L sans se préoccuper de sa longueur à l'aide de la syntaxe :

```
for i in L
```

Le programme précédant peut ainsi s'écrire également :

```
L=[1,3,7,11]
s=0
for i in L:
   s=s+i
print(s)
```

L'avantage de cette méthode est qu'elle permet de lire et comprendre (et donc corriger) beaucoup plus facilement les programme écrits. En un sens, c'est une bonne pratique "Pythonnesque".

Par contre, cette méthode est qu'elle ne permet pas d'accéder à l'indice i au moment du parcours de la liste.

Si l'on souhaite par exemple évaluer le polynôme

$$P(X) = \sum_{i=0}^{i=n} a_i X^i$$

en un point a, avec les coefficients de P stockés dans une liste L on souhaiterait accéder à la fois à i et a_i. Si la première construction fonctionne toujours, Python propose une solution élégante sous la forme de la syntaxe :

```
for i, ai in enumerate(L):
```

Dans notre exemple :

```
def evaluer(P,a):
   s = 0
   for i, ai in enumerate(P):
      s=s+ai*a**i
   return s
L=[1,3,-2,4] # Représente le polynôme 4X^3 - 2X^2 + 3X + 1
print(evaluer(L,2)) # Affiche P(2)=31
```

7 Recherche par balayage dans un tableau

Une première problématique consiste à savoir si un élément donné est présent dans une liste. À la différence du paragraphe précédant on ne va pas nécessairement parcourir tous les éléments de la liste, car on souhaite interrompre le parcours dès qu'un élément est trouvé.

Deux solutions sont possibles :

1. Utiliser une boucle while :

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

2. Interrompre à l'aide d'un return une boucle for dès qu'un élément est trouvé :

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

Exercice 3 : Implémenter l'algorithme précédant en le modifiant pour qu'il renvoie le premier indice du tableau contenant l'élément recherché ou None si l'élément est absent

Exercice 4 : Modifier le code précédent pour qu'il renvoie une liste composée de tous les indices contenant l'élément recherché ou une liste vide

Ces deux exemples permettent d'illustrer une idée nouvelle concernant la complexité d'un algorithme.

Le premier algorithme ne parcourra la liste que tant que l'élément recherché n'est pas trouvé. Ainsi son temps d'exécution est inconnu : au mieux l'algorithme nécessitera 1 opération, au pire il en nécessitera len(L).

Le second algorithme parcourra nécessairement cette liste len(L) fois. Ainsi les deux algorithmes ont la même complexité au pire mais pas la même complexité dans le meilleur des cas.

8 Recherche dichotomique dans un tableau trié

Lorsque le tableau dans lequel on recherche un élément est trié (dans ce paragraphe nous supposerons qu'il est rangé dans l'ordre croissant ; le cas où il est rangé dans l'ordre décroissant se traite de la même manière), on peut nettement améliorer l'algorithme de recherche en utilisant un procédé de dichotomie.

L'idée de base n'est plus de parcourir tout le tableau mais de comparer l'élément recherché à la valeur médiane du tableau. Si l'élément recherché est inférieure à celle-ci on recommencera notre recherche dans la partie gauche du tableau, si il est plus grand dans la partie droite.

Dans le détail, voici une présentation de cet algorithme en pseudo-code :

Exercice 5 : commenter le pseudo-code suivant en expliquant la démarche utilisée :

```
données : liste L, élément a

g=0

d=len(L)-1

tant que g ≤ d faire :

m=int(g+d/2)

si L[m]=x : renvoyer m ; fin de l'algorithme

sinon si L[m]<x : affecter m+1 à g

sinon : affecter m-1 à d

fin tant que

renvoyer None ; fin de l'algorithme
```

Exercice 6 : Quelle est la complexité au pire de l'algorithme précédant ?