Chapitre 3 :
Types séquentiels 1 :
listes et tuples

I Types séquentiels

1.1 Introduction

Cette année, on étudiera quatre types séquentiels :

- les listes,
- les tuples,
- les chaînes de caractères,
- les dictionnaires.

Les types séquentiels sont des types de données avancés. En effet, contrairement aux données numériques (float, int, bool ...), ils sont composés de plusieurs données.

1.2 Les listes

Les listes sont des successions d'objets séparés par des virgules et délimitées par des crochets [et]. Elles peuvent contenir des objets de types quelconques.
Voici quelques exemples de listes :

```
>>> l1=[  ] #Liste vide
>>> l2=[1,2,3]
>>> l3=[1,2.05,True]
>>> l4=[2,[4],[4,5]]
```

Le type des listes est : list.

Les listes peuvent être définies en "compréhension", c'est-à-dire avec des éléments donnés par une expression quand un indice parcourt un ensemble.

Pour cela, on utilisera la commande range. On rappelle que :

- range(a,b) signifie de a inclus à b exclus,
- range(a,b,p) signifie de a inclus à b exclus avec un pas p,
- range(b) signifie de 0 inclus à b exclus.

```
>>> l=[i**2 for i in range(1,10)]
>>> l
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On peut également utiliser plusieurs indices :

```
>>> l=[100*i+j for i in range(1,5) for j in range(1,4)]
>>> l
[101, 102, 103, 201, 202, 203, 301, 302, 303, 401, 402, 403]
```

➪ **Exemple 1 :**

1. Quelle est la liste définie par la commande suivante ?

```
>>> l=[100*i+j for i in range(1,5) for j in range(1,i+1)]
```

.....

2. Ecrire une commande permettant de définir la liste :

$$[0,2,4,6,\ldots,100].$$

.....

3. Ecrire une commande permettant de définir la liste :

$$[[], [1], [1,2], [1,2,3], \ldots, [1,2,3,\ldots,50]].$$

.....

1.3 Les tuples

Les tuples sont des successions d'objets séparés par des virgules et délimités par des parenthèses (et). Ils peuvent contenir des objets de types quelconques.
Voici quelques exemples de tuples :

```
>>> t1=(  )
>>> t2=(1,2,3)
>>> t3=(1,2.05,True)
>>> t4=(2,(1,2),[4,5])
```

Le type des tuples est : tuple.
De même que pour les listes, les tuples peuvent être définis en "compréhension".

II Opérations basiques

2.1 Longueur

La commande len(...) donne la longueur d'un objet de type séquentiel :

```
>>> l=[0,8,[1,2]]
>>> len(l)
3
>>> t=(1,[5,6],())
>>> len(t)
3
```

2.2 Extraction

L'extraction permet d'accéder aux informations stockées dans une variable de type séquentiel.

Soit s une liste ou un tuple.
La commande s[k] permet d'accéder à l'élément situé en $k^{\text{ième}}$ position dans la séquence.

La numérotation des éléments de listes et de tuples commence à 0.

On peut utiliser des valeurs négatives de k pour compter à partir de la fin, on commence alors la numérotation à -1.

n^o 0	n^o1	n^o2	n^o3	...	n^o(-3)	n^o(-2)	n^o(-1)

La commande s[i :j] permet de construire la séquence constituée des éléments de l d'indice variant de i inclus à j exclu.
La commande s[i :j :k] permet de construire la séquence constituée des éléments de l d'indice variant de i inclus à j exclu avec un pas k.

```
>>> l=[1,0.2,[2,4],8,7]
>>> t=(1,0.5,1/3)
>>> l[1]
0.2
>>> t[-1]
0.3333333333333333
>>> l[0 :len(l) :2]
[1,[2,4],7]
>>> l[2]
[2,4]
>>> l[2][1]
4
```

2.3 Test d'appartenance

L'opérateur `in` permet de tester l'appartenance à une séquence. Son résultat est un booléen. Sa négation est `not in`.

```
>>> l=[1,0.2,[2,4],8,7]
>>> t=(1,0.5,1/3)
>>> 1 in l
True
>>> 2 in l
False
>>> 2 in l[2]
True
>>> 1 in t
True
>>> 1 not in t
False
```

2.4 Concaténation

On considère `s1` et `s2` deux listes ou deux tuples, c'est-à-dire deux séquences **de même nature**. La commande `s1+s2` désigne la concaténation de `s1` et `s2`, c'est-à-dire la séquence obtenue en mettant bout à bout les deux séquences.
Pour concaténer n copies d'une séquence `s`, on utilise indifféremment `n*s` ou `s*n`.

Les symboles + et * ne représentent pas l'addition et la multiplication lorsqu'ils sont appliqués à des séquences.

```
>>> l1=[1,2,4]
>>> l2=[3,5]
>>> l1+l2
[1, 2, 4, 3, 5]
>>> l1*3
[1, 2, 4, 1, 2, 4, 1, 2, 4]
```

2.5 Conversion de type

On peut transformer une séquence en :

- tuple avec la commande `tuple`
- liste avec la commande `list`

Cela peut être intéressant lors de l'utilisation de `range(n,m)` qui donne tous les entiers compris entre n inclus et m exclu. Cependant, `range` crée une séquence de type particulier : les éléments ne sont pas explicitement représentés en mémoire mais révélés en itérant la séquence.

```
>>> r=range(5)
>>> type(r)
<class 'range'>
>>> list(r)
[0, 1, 2, 3, 4]
```

2.6 Itérables

Les listes et les tuples sont itérables : on peut parcourir leurs valeurs, à l'aide d'un for par exemple.

```
1 def fonction1(l) : # l est une liste ou un tuple de nombres
2     n=len(l)
3     s=0
4     for x in l :
5         s+=x
6     return s/n
```

fonction1(l) renvoie la moyenne des éléments de l.

III Données mutables et données immuables

3.1 Définition

Les séquences Python sont mutables ou immuables :

- immuables : leur valeur ne peut pas changer (à moins de les redéfinir entièrement)
- mutables : leur valeur peut changer.

Les listes sont mutables, les tuples sont immuables.

```
>>> l=[1,2,3] ;t=(1,2,3) ;
>>> l[1]=9
>>> l
[1, 9, 3]
>>> l[3]=6
Traceback (most recent call last) :
File "<console>", line 1, in <module>
IndexError : list assignment index out of range
>>> t[1]=9
Traceback (most recent call last) :
File "<stdin>", line 1, in <module>
TypeError : 'tuple' object does not support item assignment
```

3.2 Copie

Dans cette partie, on ne considère que des données mutables, c'est-à-dire des listes.

On souhaite copier des données afin de conserver l'original et faire des modifications sur la copie.

La commande = permet de faire une copie mais les deux copies sont liées. Ainsi, il sera impossible de ne modifier qu'une seule version, les deux versions seront toujours identiques.

```
>>> l1=[1,2,3]
>>> l2=l1
>>> l1[1]=9
>>> l1
[1, 9, 3]
>>> l2
[1, 9, 3]
```

Afin de remédier à ce problème, on peut utiliser la méthode copy qui faut partie du module copy.

```
>>> import copy
>>> l1=[1,2,3]
>>> l2=copy.copy(l1)
>>> l1[1]=9
>>> l1
[1, 9, 3]
>>> l2
[1, 2, 3]
```

Cependant, la copie obtenue est dite "superficielle". Cela signifie que si l'élément modifié est mutable, cette modification sera également reflétée dans la copie. Dans ce cas, il faudra utiliser une copie profonde.

```
>>> l1=[[1,2],[3,4]]
>>> l2=copy.copy(l1)
>>> l3=copy.deepcopy(l1)
>>> l1[1][0]=5
>>> l1
[[1, 2], [5, 4]]
>>> l2
[[1, 2], [5, 4]]
>>> l3
[[1, 2], [3, 4]]
```

IV Commandes particulières aux listes

4.1 Modification de listes

On suppose ici qu'on a défini une liste l.

- l.append(x) rajoute un élément x à la fin de la liste l,
- l.pop() renvoie et supprime le dernier élément de la liste l.

```
>>> l=[1,2,3,4,5]
>>> l.append(6)
>>> l
[1, 2, 3, 4, 5, 6]
>>> l.pop()
6
>>> l
[1, 2, 3, 4, 5]
```

4.2 Construction de listes

On peut utiliser deux méthodes pour ajouter un élément x à la fin d'une liste l :

- l=l+[x]
- l.append(x)

Ces deux méthodes donnent le même résultat mais n'ont pas la même coût.

Dans l'exemple suivant, on construit une liste composée de n zéros.

```
def test1(n) :
    l=[]
    t=time.time()
    for i in range(n) :
        l=l+[0]
    return time.time()-t

def test2(n) :
    l=[]
    t=time.time()
    for i in range(n) :
        l.append(0)
    return time.time()-t

>>> test1(10**4)
0.1401820182800293
>>> test1(10**5)
15.323781490325928
>>>  test2(10**5)
0.0
>>> test2(10**6)
0.07752299308776855
>>> test2(10**7)
0.7805414199829102
```

On voit ainsi que la concaténation a un coût linéaire, alors que la méthode .append a un coût constant.

Il faut utiliser la méthode .append pour ajouter un élément à une liste plutôt que la concaténation.

Cette différence vient du fait que, lors de la concaténation, une copie de liste est effectuée. On peut voir que la liste est copiée avec le test suivant :

```
>>> l=[1,2,3,4]
>>> t=l
>>> l.append(5)
>>> l
 [1,2,3,4,5]
>>> t
 [1,2,3,4,5]
>>> l=l+[6]
>>> l
 [1,2,3,4,5,6]
>>> t
   [1,2,3,4,5]
```

V Algorithmes classiques

5.1 Recherche d'un élément dans une liste d'entiers

On considère une liste l d'**entiers** et un **entier** x. La fonction suivante renvoie True si x est un élément de l et False sinon.

```
1 def recherche(l,x) :
2     n=len(l)
3     v=False
4     for i in range(n) : #i décrit tous les indices de la liste
5         if l[i]==x :
6             v=True # x est dans la liste
7     return v # si x n'est pas dans la liste, v garde sa valeur initiale : False
```

On obtient :

```
>>> l=[0,5,12,3,4,5]
>>> recherche(l,5)
True
>>> recherche(l,2)
False
>>> l=[2/i+1/j for i in range(1,11) for j in range(1,11)]
>>> recherche(l,0.3)
False
```

Cependant pour $i = 10$ et $j = 10$, on a $\frac{2}{i} + \frac{1}{j} = 0.3$, donc la réponse devrait être True.

Dans la fonction recherche on a utilisé un test d'égalité. Cette fonction n'est donc applicable que pour les listes d'**entiers**.

La fonction recherche permet de programmer le test d'appartenance in. Si on compare la fonction écrite et la fonction préprogrammée, la complexité est la même.

```
import time

def recherche(l,x) :
    tps=time.time()
    n=len(l)
    v=False
    for i in range(n) :
        if l[i]==x :
            v=True
    print("temps d'execution :",time.time()-tps)
    return v

def recherche2(l,x) :
    tps=time.time()
    v=x in l
    print("temps d'execution :",time.time()-tps)
    return v
```

```
>>> l=[0]*(10**7)
>>> recherche(l,1)
temps d'execution : 0.4670403003692627
False
>>> recherche2(l,1)
temps d'execution : 0.07816195487976074
False
>>> l=[0]*(10**8)
>>> recherche(l,1)
temps d'execution : 4.199077367782593
False
>>> recherche2(l,1)
temps d'execution : 0.8030221462249756
False
```

On remarque que les deux complexités sont linéaires.

5.2 Recherche du maximum

On considère une liste l de nombres. La fonction suivante renvoie le maximum des éléments de l.

```
1 def max(l) :
2     n=len(l)
3     m=l[0] # initialisation du maximum
4     for i in range(n) :
5         if m<l[i] : #m est le maximum de l[0],...,l[i]
6             m=l[i]
7     return m
```