listes-tuples

September 8, 2019

#
Listes, p-uplets et p-uplets nommés.

Nous avons déjà rencontrés différents types de base en Python : en citer 4 (au moins) !

On peut travailler avec des éléments plus élaborés, par exemples des séquences.

Une séquence est un ensemble fini et ordonné d'éléments. Nous aborderons deux types de séquences : **les listes** et **les tuples**.

0.1 I Les listes

[5]:
```
L = []
```

Nous venons de créer ici la liste L qui est vide

[6]:
```
L = [0]*10
print(L)
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

La liste L contient exactement dix fois le nombre 0.

[7]:
```
L = [i for i in range(5)]
print(L)
```

```
[0, 1, 2, 3, 4]
```

La liste L vient ici d'être construite **par compréhension**.
Une liste est une séquence : elle est ordonnée. **Son premier indice a pour rang 0** !!
On accède ainsi aux éléments de la liste par leur rang. Prenons par exemple la liste suivante L = [1,5,-2,3,'toto',2/3].

[11]:
```
L = [1,5,-2,3,'toto',2/3]
```

Que renvoie les commandes :
L[2] ?
L[:2] ?
L[2:] ?
L[2:4] ?
type(L[4]) ?

[12]:
```
#tester cela ici et prendre des notes !
```

Les listes sont mutables : on peut changer un élément de valeur !
On peut par exemple écrire L[2] = -87, et l'élément de rang 2 de la liste L prend alors la valeur 87, la preuve :

[13]:
```
L[2] = -87
print(L)
```

```
[1, 5, -87, 3, 'toto', 0.6666666666666666]
```

On peut **ajouter** un élément à une liste à l'aide de la méthode `append`.
Il vient de positionner au dernier rang de la liste.
Un exemple :

[14]:
```
L.append(14)
print(L)
```

```
[1, 5, -87, 3, 'toto', 0.6666666666666666, 14]
```

On peut **supprimer** un élément d'une liste à l'aide de la méthode `del`.

[15]:
```
del(L[2])
print(L)
```

```
[1, 5, 3, 'toto', 0.6666666666666666, 14]
```

La méthode `remove` permet aussi d'enlever les éléments d'une liste.
La commande L.remove(5) enlèverra **tous** les 5 de la liste L. Ici il n'y en a qu'un !

[16]:
```
L.remove(5)
print(L)
```

```
[1, 3, 'toto', 0.6666666666666666, 14]
```

On peut connaître la longueur de la liste avec le méthode `len` :

[17]:
```
print(len(L))
```

```
5
```

Parcourir une liste
On peut parcourir tous les éléments d'un liste :

[18]:
```
L1 = [1,2,3,4,5]
for valeur in L1:
    print(valeur)
```

```
1
2
3
4
5
```

[ ]:
```
On peut aussi afficher autre chose à partir des éléments de la liste :
```

[20]:
```
for valeur in L1 :
    print(valeur**2)
```

```
1
4
9
16
25
```

On peut aussi tester la présence d'un élément dans une liste :

[21]:
```
def teslaoupas(v):
    if v in L1:
        return "j'y suis"
    else:
        return "pas là"
teslaoupas(5)
```

[21]:
```
"j'y suis"
```

[22]:
```
teslaoupas(-3)
```

[22]:
```
'pas là'
```

[23]:
```
#version plus courte, la fonction est booléenne.
def teslaoupasbis(v):
    return v in L1
teslaoupasbis(8)
```

[23]:
```
False
```

Copie d'une liste (dans une autre)
On considère toujours la liste L1.

[26]:
```
 m = L1
print(m)
```

```
[1, 2, 3, 4, 5]
```

[28]:
```
L1[2] = 8
print(m)
```

```
[1, 2, 8, 4, 5]
```

La modification de la liste L1 entraine donc celle de m.
Peut-on éviter cela ? Oui bien sûr !

[30]:
```
n = L1[:] # on met ici L1[:] au lieu de L1
print(n)
```

```
[1, 2, 8, 4, 5]
```

[34]:
```
# Je modifie L1 et affiche n
L1.append(2019)
print(n)
print(L1)
print(m)
```

```
[1, 2, 8, 4, 5]
[1, 2, 8, 4, 5, 2019, 2019, 2019, 2019]
[1, 2, 8, 4, 5, 2019, 2019, 2019, 2019]
```

Remarque: on aurait pu écrire n = list(L1) au lieu de n = L1[:]

Création d'une liste en compréhension

la compréhension de liste est une expression qui permet de construire une liste à partir de tout autre objet itérable (liste, chaîne de caractère,...). Le résultat obtenu est toujours une liste !

[35]:
```
prenom = 'albert'
L2 = [lettre for lettre in prenom]
print(L2)
```

```
['a', 'l', 'b', 'e', 'r', 't']
```

[36]:
```
L3 = [p**2 for p in range(15)]
print(L3)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

On peut même utiliser des instructions conditionnelles dans la création d'une liste par compréhension !

[37]:
```
L4 = [p**2 for p in range(15) if p%3 == 0]
print(L4)
```

```
[0, 9, 36, 81, 144]
```

[39]:
```
L5 = [i for i in range(16)]
L6 = [n**2 if n%2 == 0 else n*3 for n in L5]
print(L5,L6)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] [0, 3, 4, 9, 16, 15, 36,
21, 64, 27, 100, 33, 144, 39, 196, 45]
```

0.1.1 Exercices

Soit L = [1,2,5,7,9], créer une liste à partir de L, qui contient les cubes des éléments de L.

Soit L = [A,b,C,d]. Ecrire un programme qui remplace chaque lettre de la liste L par son code ASCII.

Créer une liste qui contient les 100 premiers nombres entiers non multiples de 2,3 ou 5.

Créer une liste par compréhension qui contient les cubes des entiers non multiples de 5 plus petits que 101.

0.2 II Les tuples

Si les tuples ont des points communs avec les suites, ce sont des séquences (ensemble fini d'éléments ordonnés), ils diffèrent dans la mesure où se sont des objets **non mutables**.

[42]:
```
t = (1,2,3,4,5)
print(type(t))
print(len(t))
print(t[1])
```

```
<class 'tuple'>
5
2
```

Illustration du fait qu'un tuple soit non mutable :

[43]:
```
t[1]=14
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-43-b64842b18d35> in <module>
----> 1 t[1]=14


TypeError: 'tuple' object does not support item assignment
```

Les propriétés des tuples sont assez identiques à celles des listes. On peut cependant réaliser l'opération suivante (que je ne connaissais pas...)

[44]:
```
t = (7,)+t[1:]
print(t)
```

```
(7, 2, 3, 4, 5)
```

Ne pas oublier la virgule dans (7,)...

[45]:
```
t = (7)+t[1:]
```

```
␣
↪---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)

<ipython-input-45-76cdbf53e88f> in <module>
----> 1 t = (7)+t[1:]


TypeError: unsupported operand type(s) for +: 'int' and 'tuple'
```

Remarque, on peut avoir une présentation plus soignée :

0.3 III Les tableaux à deux dimensions

Après avoir jouer avec des tableaux à une dimension, nous allons maintenant nous pencher (mais pas trop quand même) sur les tableaux à deux dimensions !

On va définir ces tableaux à l'aide d'une liste de listes !

Création des tableaux
Supposons que nous voulions créer un tableau de 5 lignes et 5 colonnes contenant de 0 !
On peut très bien faire : tab = [[0,0,0,0,0],[0,0,0,0,0],. . . .] bref c'est pénible surtout si l'on veut plus grand. Alors on peut écrire :

[46]:
```
tab = [[0]*5]*5
print(tab)
```

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0,
0, 0]]
```

En détail : [0]*5 crée [0,0,0,0,0] et on répète 5 fois l'opération qui est enfin stockée dans la liste tab !

0.3.1 Exercices

Créer un tableau dont le première ligne contient les entiers de 0 à 9, la seconde ligne ligne les entiers de 10 à 19, . . . , la dernière ligne (dixième donc) les entiers de 90 à 99 :
le faire à l'aide de boucle for
en utilisant des listes par compréhension
Recherche d'un élément dns un tableau !
On considère le tableau suivant :

[48]:
```
tab = [[0,1,2,3,4],[5,6,7,8,9],[10,11,12,13]]
print(tab)
```

```
[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13]]
```

Que valents les éléments tab[2][1] et tab[1][2] du tableau ?

[50]:
```
#tab[2][1]
#tab[1][2]
```

L'élément tab[i][j] correspond au j-ième élément de la i-ième liste (On compte à partir de 0 !!!!)-

Trier un tableau (ou une liste)

Avant de travailler sur certains algorithmes de tri, la méthode `sort` permet de trier de manière efficace en Python !

[52]:
```
Liste = [-3,4,0,8,9,2]
Liste.sort()
print(Liste)
```

```
[-3, 0, 2, 4, 8, 9]
```

[55]:
```
Listebis =['bac','cab','acb','abc','cba','bca']
Listebis.sort()
print(Listebis)
```

```
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
```

[56]:
```
Listebis.sort(reverse=True) # dans l'ordre décroissant !
print(Listebis)
```

```
['cba', 'cab', 'bca', 'bac', 'acb', 'abc']
```

Pour trier un tableau ayant plusieurs types de données, on doit alors choisir un critère de classement !

[58]:
```
T = [['Anna',17],['benjamin',12],['Sarah',15],['Kevin',9]]
tri = sorted(T, key = lambda note : note[1])
print(tri)
```

```
[['Kevin', 9], ['benjamin', 12], ['Sarah', 15], ['Anna', 17]]
```

Expliquons : on peut classer par ordre alphabétique, ou par notes croissantesn voir décroissantes !
On trie donc suivant l'élément 0 (le nom) ou l'élément 1 la note.
On utilise la méthode `sorted`. La liste triée s'appelle tri

0.3.2 Exercices

Afficher le tableau T classé par ordre alphabétique inversé.

a) créer une liste de cinq listes contenant chacune trois éléments :
un mot en français, sa traduction en deux autres langues.

b) Afficher le tableau avec un classement par ordre alphabétique en français, puis dans ur

c) Afficher le tableau tel que le premier élément de chaque liste soit un mot étranger.

[ ]: