Cours

1) Les séquences en Python

Il est possible de "stocker" plusieurs grandeurs dans une même structure, ce type de structure est appelé une séquence. De façon plus précise, nous définirons une séquence comme un ensemble fini et ordonné d'éléments indicés de 0 à n-1 (si cette séquence comporte n éléments). Rassurez-vous, nous reviendrons ci-dessous sur cette définition. Nous allons étudier plus particulièrement 2 types de séquences : les tuples et les tableaux (il en existe d'autres que nous n'évoquerons pas ici).

a) Les tuples en Python

Comme déjà dit ci-dessus, un tuple est une séquence. Voici un exemple très simple :

```
mon_tuple = (5, 8, 6, 9)
```

Dans le code ci-dessus, le nom mon_tuple est associé à un tuple (l'association entre un nom et un tuple est aussi une variable), ce tuple est constitué des entiers 5, 8, 6 et 9. Comme indiqué dans la définition, chaque élément du tuple est indicé (il possède un indice):

le premier élément du tuple (l'entier 5) possède l'indice 0 le deuxième élément du tuple (l'entier 8) possède l'indice 1 le troisième élément du tuple (l'entier 6) possède l'indice 2 le quatrième élément du tuple (l'entier 9) possède l'indice 3 Comment accéder à l'élément d'indice i dans un tuple ?

Simplement en utilisant la "notation entre crochets" :

```
mon_tuple = (5, 8, 6, 9)
a = mon_tuple[2]
```

Dans le programme ci-dessus, la variable *a* a pour valeur 6.

ATTENTION : dans les séquences les indices commencent toujours à 0 (le premier élément de la séquence a pour indice 0), oublier cette particularité est une source d'erreur "classique".

Un tuple ne contient pas forcément des nombres entiers, il peut aussi contenir des nombres décimaux, des chaînes de caractères, des booléens...

Dans le programme ci-dessous :

```
mon_tuple = ("le", "monde", "bonjour")
```

```
msg = mon_tuple[2] + " " + mon_tuple[0] + " " + mon_tuple[1] + "!"
```

la variable msg a pour valeur : "bonjour le monde!"

Grâce au tuple, une fonction peut renvoyer plusieurs valeurs :

Intéressons-nous au programme suivant :

```
def add(a, b):
    c = a + b
    return (a, b, c)
val = add(5, 8)
```

Après exécution du programme ci-dessus, la variable *val* a pour valeur le tuple *(5, 8, 13)* car notre fonction *add* renvoie bien un tuple (*return (a, b, c)*)

Il est possible d'associer à des noms les valeurs contenues dans un tuple. Dans l'exemple ci-dessous :

```
a, b, c = (5, 8, 6)
```

la variable *a* a pour valeur 5, *b* a pour valeur 8 et *c* a pour valeur 6.

b) Les tableaux en Python

ATTENTION : Dans la suite nous allons employer le terme "tableau". Pour parler de ces "tableaux" les concepteurs de Python ont choisi d'utiliser le terme de "list" ("liste" en français). Pour éviter toute confusion, notamment par rapport à des notions qui seront abordées en terminale, le choix a été fait d'employer "tableau" à la place de "liste" (dans la documentation vous rencontrerez le terme "list", cela ne devra pas vous perturber)

Il n'est pas possible de modifier un tuple après sa création (on parle d'objet "immutable"), si vous essayez de modifier un tuple existant, l'interpréteur Python vous renverra une erreur. Les tableaux sont,comme les tuples, des séquences, mais à la différence des tuples, ils sont modifiables (on parle d'objets "mutables").

Pour créer un tableau, il existe différentes méthodes : une de ces méthodes ressemble beaucoup à la création d'un tuple :

```
mon_tab = [5, 8, 6, 9]
```

Notez la présence des crochets à la place des parenthèses.

Un tableau est une séquence, il est donc possible de "récupérer" un élément d'un tableau à l'aide de son indice (de la même manière que pour un tuple)

Dans le cas ci-dessous :

```
mon_tab = [5, 8, 6, 9]
val = mon_tab[1]
```

la variable *val* a pour valeur 8 (index 0 : 5, index 1 : 8...)

Il est possible de modifier un tableau à l'aide de la "notation entre crochets" :

```
mon_tab = [5, 8, 6, 9]
mon_tab[2] = 15
```

Après l'exécution du programme ci-dessus, la tableau *mon_tab* est constitué des valeurs suivantes : [5, 8, **15**, 9]. L'élément d'indice 2 (le nombre entier 6) a bien été remplacé par le nombre entier 15

Il est aussi possible d'ajouter un élément en fin de tableau à l'aide de la méthode "append" :

```
mon_tab = [5, 8, 6, 9]
mon_tab.append(15)
```

Après l'exécution du programme ci-dessus, la tableau *mon_tab* est constitué des valeurs suivantes : [5, 8, 6, 9, **15**]. La valeur 15 a bien été ajoutée au tableau en dernière position.

L'instruction "del" permet de supprimer un élément d'un tableau en utilisant son index :

```
mon_tab = [5, 8, 6, 9]
del mon_tab[1]
```

À la suite de l'exécution du programme ci-dessus le tableau *mon_tab* contient les valeurs [5, 6, 9] : l'élément situé à l'index 1 (c'est à dire 8) a bien été supprimé.

La fonction "len" renvoie le nombre d'éléments présents dans une séquence (tableau et tuple)

```
mon_tab = [5, 8, 6, 9]
a = len(mon_tab)
```

Après exécution du programme ci-dessus, la variable *a* a pour valeur 4 (le tableau [5, 8, 6, 9] est bien constitué de 4 éléments)

Après avoir vu les tableaux, on pourrait s'interroger sur l'intérêt d'utiliser un tuple puisque le tableau permet plus de choses ! La réponse est simple : les opérations sur les tuples sont plus "rapides". Quand vous savez que votre tableau ne sera pas modifié, il est préférable d'utiliser un tuple à la place d'un tableau.

2) Parcourir une séquence à l'aide de la boucle *for*

La boucle for... in permet de parcourir chacun des éléments d'une séquence (tableau ou tuple) :

Prenons l'exemple suivant :

```
mon_tab = [5, 8, 6, 9]
for ele in mon_tab:
    print(ele)
```

L'exécution du programme ci-dessus permettra d'afficher toutes les valeurs contenues dans le tableau *mon_tab* :

```
5
8
6
9
```

Quelques explications : comme son nom l'indique, la boucle "for" est une boucle ! Nous "sortirons" de la boucle une fois que tous les éléments du tableau mon_tab auront été parcourus :

- au premier tour de boucle, la variable *ele* sera égale 5
- au deuxième tour de boucle, la variable *ele* sera égale 8
- au troisième tour de boucle, la variable *ele* sera égale 6
- au quatrième et dernier tour de boucle, la variable *ele* sera égale 9

Une chose importante à bien comprendre : le choix du nom de la variable qui va être associé aux éléments du tableau les uns après les autres (*ele*) est totalement arbitraire, il est possible de choisir un autre nom sans aucun problème, le code suivant aurait donné exactement le même résultat :

```
mon_tab = [5, 8, 6, 9]
for toto in mon_tab:
    print(toto)
```

Dans la boucle for... in il est possible d'utiliser la fonction native *range* à la place d'un tableau d'entiers :

```
for ele in range(0, 5):
    print (ele)
```

aura exactement le même effet que le programme :

```
for ele in [0, 1, 2, 3, 4]:
    print (ele)
```

Comme vous pouvez le constater, *range(0,5)* est, au niveau de la boucle "for..in", équivalent au tableau [0,1,2,3,4]

ATTENTION : si vous avez dans un programme "range(a,b)", a est la borne inférieure et b a borne supérieure. Vous ne devez surtout pas perdre de vu que la borne inférieure est incluse, mais que la borne supérieure est exclue.

3) Créer un tableau par compréhension

Nous avons vu qu'il était possible de "remplir" un tableau en renseignant les éléments du tableau les uns après les autres :

```
mon_tab = [0, 1, 2, 3]
```

Il est aussi possible d'obtenir exactement le même résultat que ci-dessus en une seule ligne grâce à la compréhension de tableau :

```
mon_tab = [p for p in range(0, 4)]
```

nous avons une boucle for entre crochets. p va successivement prendre les valeurs 0, 1, 2, 3. Ces différentes valeurs de p vont permettre de remplir le tableau *mon_tab*.

Les compréhensions de tableau permettent de rajouter une condition (if) :

```
l = [1, 7, 9, 15, 5, 20, 10, 8]
mon_tab = [p for p in l if p > 10]
```

ci-dessus nous utilisons le tableau *l* pour créer le tableau *mon_tab* : on parcourt le tableau *l* grâce à la boucle *for p in l* mais on "garde" uniquement les valeurs supérieures à 10 (grâce au *if p > 10*). Après l'exécution du programme ci-dessus, le tableau *mon_tab* est constitué des éléments suivants : [15, 20]

Il y a aussi la possibilité d'ajouter des opérations arithmétiques :

```
l = [1, 7, 9, 15, 5, 20, 10, 8]
mon_tab = [p**2 for p in l if p < 10]
```

On utilise le tableau *l* pour créer le tableau *mon_tab* en mettant au carré tous les éléments du tableau *l* à condition qu'ils soient strictement inférieurs à 10. On obtient donc le tableau *mon_tab* suivant : [1, 49, 81, 25, 64]

4) Travailler sur des "tableaux de tableaux"

Chaque élément d'un tableau peut être un tableau, on parle de tableau de tableau.

Voici un exemple de tableau de tableau :

```
m = [[1, 3, 4], [5 ,6 ,8], [2, 1, 3], [7, 8, 15]]
```

Le premier élément du tableau ci-dessus est bien un tableau ([1, 3, 4]), le deuxième élément est aussi un tableau ([5, 6, 8])...

Il est souvent plus pratique de présenter ces "tableaux de tableaux" comme suit :

```
m = [[1, 3, 4],
     [5, 6, 8],
     [2, 1, 3],
     [7, 8, 15]]
```

Nous obtenons ainsi quelque chose qui ressemble beaucoup à un "objet mathématique" très utilisé : une matrice

Il est évidemment possible d'utiliser les indices de position avec ces "tableaux de tableaux". Pour cela nous allons considérer notre tableau de tableaux comme une matrice, c'est à dire en utilisant les notions de "ligne" et de "colonne". Dans la matrice ci-dessus :

En ce qui concerne les lignes :

- 1, 3, 4 constituent la première ligne
- 5, 6, 8 constituent la deuxième ligne
- 2, 1, 3 constituent la troisième ligne
- 7, 8, 15 constituent la quatrième ligne

En ce qui concerne les colonnes :

- 1, 5, 2, 7 constituent la première colonne
- 3, 6, 1, 8 constituent la deuxième colonne
- 4, 8, 3, 15 constituent la troisième colonne

Pour cibler un élément particulier de la matrice, on utilise la notation avec "doubles crochets" : m[ligne][colonne] (sans perdre de vu que la première ligne et la première colonne ont pour indice 0)

Si nous prenons cet exemple :

```
m = [[1, 3, 4],
     [5, 6, 8],
     [2, 1, 3],
     [7, 8, 15]]
a = m[1][2]
```

la variable *a* aura pour valeur 8.

Explications : *m[1]* correspond au tableau *[5, 6, 8]* (2e élément). Dans ce tableau *[5, 6, 8]*, à l'index 2 (*m[1][2]*), on trouve bien la valeur 8.

Si maintenant nous considérons l'exemple suivant :

```
m = [1, 2, 3]
mm = [m, m, m]
m[0] = 100
```

Après l'exécution de ce programme le tableau *mm* est constitué des éléments suivants : [[100, 2, 3], [100, 2, 3], [100, 2, 3]]

Comme vous pouvez le constater, la modification du tableau associé au nom m entraîne la modification du tableau associé au nom mm (alors que nous n'avons pas directement modifié le tableau associé au nom mm). Il faut donc être très prudent lors de ce genre de manipulation afin d'éviter des modifications non désirées.

Il est possible de parcourir l'ensemble des éléments d'une matrice à l'aide d'une "double boucle for" :

```
m = [[1, 3, 4],
     [5, 6, 8],
     [2, 1, 3],
     [7, 8, 15]]
nb_colonne = 3
nb_ligne = 4
for i in range(0, nb_ligne):
    for j in range(0, nb_colonne):
        a = m[i][j]
        print(a)
```

L'exécution de ce programme donnera le résultat suivant :

```
1
3
4
5
6
8
2
1
3
7
8
15
```

Nous avons bien parcouru l'ensemble des éléments du tableau *m*.

Cette double boucle *for* est une structure complexe, mais pourtant assez répandue. N'hésitez pas à consacrer quelques minutes à son analyse.