

Chapitre 10 : Types construits

[version complète]

Les compréhensions de listes fournissent un moyen de construire des listes de manière très concise. Une application classique est la construction de nouvelles listes où chaque élément est le résultat d'une opération appliquée à chaque élément d'une autre séquence ; ou de créer une sous-séquence des éléments satisfaisant une condition spécifique.

I. 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).

Nous allons étudier plus particulièrement 2 types de séquences : les tuples et les listes (il en existe d'autres que nous n'évoquerons pas ici).

1. 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)
autre_tuple = 'hello', # <-- notez la virgule
```

Dans le code ci-dessus, la variable "mon_tuple" référence un tuple, 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

Pour accéder à l'élément d'indice i dans un tuple il suffit d'utiliser la "notation entre crochets" :

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

Quelle est la valeur référencée par la variable a ?

La variable mon_tuple référence le tuple (5, 8, 6, 9), la variable a référence l'entier 6 car cet entier 6 est bien le troisième élément du tuple, il possède donc l'indice 2.

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".

Complétez le code ci-dessous (en remplaçant les ..) afin qu'après l'exécution de ce programme la variable a référence l'entier 8.

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

a = mon_tuple[..]
```

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

Quel est le résultat attendu après l'exécution de ce programme ?

Testons ce programme :

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

print(mon_tuple[2] + " " + mon_tuple[0] + " " + mon_tuple[1] + "!")
```

Grâce au tuple, une fonction peut renvoyer plusieurs valeurs :

```
def add(a, b):

    c = a + b

    return (a, b, c)

mon_tuple = add(5, 8)

print(f"{mon_tuple[0]} + {mon_tuple[1]} = {mon_tuple[2]}")
```

Il faut bien comprendre dans l'exemple ci-dessus que la variable mon_tuple référence un tuple (puisque la fonction "add" renvoie un tuple), d'où la "notation entre crochets" utilisée avec mon_tuple (mon_tuple[1]...)

Après avoir exécuté le programme ci-dessous, saisissez mon_tuple dans la console.

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

Il est possible d'assigner à des variables les valeurs contenues dans un tuple :

```
a, b, c, d = (5, 8, 6, 9)
```

Quelle est la valeur référencée par la variable a ? La variable b ? La variable c ? La variable d ?

2. LES LISTES EN PYTHON

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

Pour créer une liste, 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.

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

Quelle est la valeur référencée par la variable `ma_variable` après l'exécution du programme ci-dessous ?

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

NB : il est possible de saisir directement `mon_tab[2]` dans la console sans passer par l'intermédiaire de la variable `ma_variable`.

Il est possible de modifier une liste à l'aide de la "notation entre crochets".

Quel est le contenu de la liste référencée par la variable `mon_tab` après l'exécution du programme ci-dessous ?

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

L'instruction "`del`" permet de supprimer un élément d'une liste en utilisant son index.

Quel est le contenu de la liste référencée par la variable `mon_tab` après l'exécution du programme ci-dessous ? (utilisez la console pour vérifier votre réponse).

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

La fonction "len" permet de connaître le nombre d'éléments présents dans une séquence (liste et tuple).

Quelle est la valeur référencée par la variable nb_len après l'exécution du programme ci-dessous ? (utilisez la console pour vérifier votre réponse).

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

nb_ele = len(mon_tab)
```

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

3. LA BOUCLE "FOR" : PARCOURIR LES ELEMENTS D'UNE LISTE

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

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

for element in mon_tab:

    print(element)
```

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

element est une variable qui va :

- au premier tour de boucle, référencer le premier élément de la liste (l'entier 5)
- au deuxième tour de boucle, référencer le deuxième élément de la liste (l'entier 8)
- au troisième tour de boucle, référencer le troisième élément de la liste (l'entier 6)
- au quatrième tour de boucle, référencer le quatrième élément de la liste (l'entier 9)

Une chose importante à bien comprendre : le choix du nom de la variable qui va référencer les éléments de la liste les uns après les autres (*element*) 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 item in mon_tab:

    print(item)
```

Dans la boucle `for... in` il est possible d'utiliser la fonction prédéfinie `range` à la place d'une liste d'entiers :

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

Comme vous pouvez le constater, "`range(0,5)`" est, au niveau de la boucle "`for..in`", équivalent à la liste `[0,1,2,3,4]`, le code ci-dessous donnerait le même résultat que le programme vu précédemment.

```
mon_tab = [0, 1, 2, 3, 4]  
  
for element in mon_tab:  
    print(element)
```

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

Il est possible d'utiliser la méthode "`range`" pour "remplir" une liste.

Quel est le contenu de la liste référencée par la variable `mon_tab` après l'exécution du programme ci-dessous ?

```
mon_tab = []  
  
for element in range(0, 5):  
    mon_tab.append(element)
```

4. CREER UNE LISTE PAR COMPREHENSION

Nous avons vu qu'il était possible de "remplir" une liste en renseignant les éléments de la liste les uns après les autres :

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

ou encore à l'aide de la méthode "`append`".

Il est aussi possible d'obtenir exactement le même résultat en une seule ligne grâce à la compréhension de liste.

Quel est le contenu de la liste référencée par la variable `mon_tab` après l'exécution du programme ci-dessous ?

```
mon_tab = [p for p in range(0, 5)]
```

Les compréhensions de liste permettent de rajouter une condition (`if`).

Quel est le contenu de la liste référencée par la variable `mon_tab` après l'exécution du programme ci-dessous ?

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

autre possibilité, utiliser des composants "arithmétiques".

Quel est le contenu de la liste référencée par la variable `mon_tab` après l'exécution du programme ci-dessous ?

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

*Rappel : $p**2$ permet d'obtenir la valeur de p élevée au carrée*

Comme vous pouvez le remarquer, nous obtenons une liste (`mon_tab`) qui contient tous les éléments de la liste `l` élevés au carré à condition que ces éléments de `l` soient inférieurs à 10. Comme vous pouvez le constater, la compréhension de liste permet d'obtenir des combinaisons relativement complexes.

5. TRAVAILLER SUR DES "LISTES DE LISTES"

Chaque élément d'une liste peut être une liste, on parle de liste de listes.

Voici un exemple de liste de listes :

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

Le premier élément de la liste ci-dessus est bien une liste (`[1, 3, 4]`), le deuxième élément est aussi une liste (`[5, 6, 8]`)...

Il est souvent plus pratique de présenter ces "listes de listes" 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 "listes de listes". Pour cela nous allons considérer notre liste de listes 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 vue que la première ligne et la première colonne ont pour indice 0).

Quelle est la valeur référencée par la variable `a` après l'exécution du programme ci-dessous ?

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

Comme vous pouvez le constater, la variable `a` référence bien l'entier situé à la 2e ligne (indice 1) et à la 3e colonne (indice 2), c'est-à-dire 8.

Quel est le contenu de la liste référencée par la variable `mm` après l'exécution du programme ci-dessous ?

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

Comme vous pouvez le constater, la modification de la liste référencée par la variable `m` entraîne la modification de la liste référencée par la variable `mm` (alors que nous n'avons pas directement modifié la liste référencée par `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, nb_ligne = 3, 4  
  
for i in range(0, nb_ligne):  
    for j in range(0, nb_colonne):  
        a = m[i][j]  
  
        print(a)
```

2. LES DICTIONNAIRES

Comme les listes, les dictionnaires permettent de "stocker" des données. Chaque élément d'un dictionnaire est composé de 2 parties, on parle de paires "clé : valeur". Voici un exemple de dictionnaire :

```
mon_dico = {"nom": "Gagarine", "prenom": "Youri ", "date de  
naissance": "09/03/1934"}
```

Comme vous pouvez le constater, nous utilisons des accolades {} pour définir le début et la fin du dictionnaire (alors que nous utilisons des crochets [] pour les listes et les parenthèses pour les tuples). Dans le dictionnaire ci-dessus, "nom", "prenom" et "date de naissance" sont des clés et "Gagarine", "Youri" et "09/03/1934" sont des valeurs. La clé "nom" est associée à la valeur "Gagarine", la clé "prenom" est associée à la valeur "Youri" et la clé "date de naissance" est associée à la valeur "09/03/1934". Les clés sont des chaînes de caractères ou des nombres. Les valeurs peuvent être des chaînes de caractères, des nombres, des booléens...

Pour créer un dictionnaire, il est aussi possible de procéder comme suit :

```
mon_dico = {}  
  
mon_dico["nom"] = "Gagarine"  
  
mon_dico["prenom"] = "Youri"  
  
mon_dico["date de naissance"] = "09/03/1934"
```

La variable "mon_dico" référence un dictionnaire. Il est possible d'afficher le contenu du dictionnaire référencé par la variable "mon_dico" en saisissant "mon_dico" dans la console :

```
mon_dico = {"nom": " Gagarine ", "prenom": " Youri ", /  
            "date de naissance": "09/03/1934"}
```

Il est possible d'afficher la valeur associée à une clé :

```
mon_dico = {"nom": " Gagarine ", "prenom": " Youri ", /  
            "date de naissance": "09/03/1934"}  
  
print(f'Bonjour je suis {mon_dico["prenom"]} {mon_dico["nom"]}, /  
je suis né le {mon_dico["date de naissance"]}') 
```

Quel est le résultat attendu après l'exécution de ce programme ? Vérifiez votre réponse.

Il est facile d'ajouter un élément à un dictionnaire (les dictionnaires sont mutables) :

```
mon_dico = {"nom": " Gagarine ", "prenom": " Youri ", /  
            "date de naissance": "09/03/1934"}  
  
print(f'Bonjour je suis {mon_dico["prenom"]} {mon_dico["nom"]}, /  
je suis né le {mon_dico["date de naissance"]}')  
  
mon_dico['lieu naissance'] = "Bonneville"  
  
print (f'à {mon_dico["lieu naissance"]}') 
```

Quel est le résultat attendu après l'exécution de ce programme ? Vérifiez votre réponse.

L'instruction "del" permet de supprimer une paire "clé/valeur". Quel est le contenu du dictionnaire référencé par la variable "mes_fruits" après l'exécution du programme ci-dessous ? Vérifiez votre réponse à l'aide de la console.

```
mes_fruits = {"poire": 3, "pomme": 4, "orange": 2}  
  
del mes_fruits["pomme"]
```

Il est possible de modifier une valeur :

```
mes_fruits = {"poire": 3, "pomme": 4, "orange": 2}  
  
mes_fruits["pomme"] = mes_fruits["pomme"] - 1
```

Quel est le contenu du dictionnaire référencé par la variable "mes_fruits" après l'exécution du programme ci-dessus ? Vérifiez votre réponse à l'aide de la console.

Il est possible de parcourir un dictionnaire à l'aide d'une boucle for. Ce parcours peut se faire selon les clés ou les valeurs. Commençons par parcourir les clés à l'aide de la méthode "keys" :

```
mes_fruits = {"poire": 3, "pomme": 4, "orange": 2}

print("liste des fruits :")

for fruit in mes_fruits.keys():
    print(fruit)
```

La méthode values() permet de parcourir le dictionnaire selon les valeurs :

```
mes_fruits = {"poire": 3, "pomme": 4, "orange": 2}

for qte in mes_fruits.values():
    print(qte)
```

Enfin, il est possible de parcourir un dictionnaire à la fois sur les clés et les valeurs en utilisant la méthode item() :

```
mes_fruits = {"poire": 3, "pomme": 4, "orange": 2}

print ("Stock de fruits :")

for fruit, qte in mes_fruits.items():
    print (f"{fruit} : {qte}")
```

Vous avez sans doute remarqué l'utilisation de deux variables ("fruit" et "qte") au niveau du "for...in".