

1 Introduction

Les types simples vus dans le précédent chapitre ne sont pas suffisants si nous avons besoin de garder en mémoire un grand nombre de valeurs comme dans le cas d'un traitement de données statistiques. Il en est de même si l'on souhaite regrouper des valeurs, par exemple afin d'avoir une variable représentant les coordonnées d'un point.

L'objectif est donc de construire un type de variable capable de contenir plusieurs valeurs. Nous pouvons nous inspirer du type **str** et utiliser des indices pour repérer les éléments.

Ceci amène à la construction des p-uplets, type **tuple**, et des listes, type **list**.

Comme pour le type **str**, un objet de type **tuple** n'est pas modifiable par une affectation `t[i]=valeur`.

Un objet de type **list** est lui modifiable par une affectation ce qui autorise de nombreuses méthodes applicables à ces objets mais en contrepartie une grande vigilance sur leur utilisation.

Un troisième type sera présenté ultérieurement, le type **dict** pour les **dictionnaires**. La principale différence avec les listes est qu'un dictionnaire n'est pas ordonné. Un élément n'est pas repéré par un indice entier mais par une "clé".

2 p-uplets

Définition

Un objet de type **tuple**, un p-uplet, est une suite ordonnée d'éléments qui peuvent être chacun de n'importe quel type. On parlera indifféremment de p-uplet ou de **tuple**.

2.1 Création d'un p-uplet

Pour créer un p-uplet non vide, on écrit n valeurs séparées par des virgules. Par exemple :

```
>>> t=1,2,3                                #Un 3-tuple à 3 éléments
>>> s= "porte","fenetre","5", 12           #un 4-tuple
>>> u=()                                    #un tuple à 0 élément
>>> x= 3.14 ,                               # un tuple à un élément (noter la présence de la virgule)
```

Pour écrire un p-uplet qui contient un n-uplet, l'utilisation de parenthèses est nécessaire. Voici un exemple avec un tuple à 2 éléments dont le second est un tuple :

```
t = 3, ("a","b","c")
```

En général, les parenthèses sont obligatoires dès que l'écriture d'un p-uplet est contenue dans une expression plus longue. Dans tous les cas, les parenthèses peuvent améliorer la lisibilité.

2.1.1 Opérations

Nous avons deux opérateurs de concaténation qui s'utilisent comme avec les chaînes de caractères, ce sont les opérateurs `+` et `*`. De nouveaux p-uplets sont créés :

```
>>> t1 = "a", "b"
>>> t2 = "c", "d"
>>> t1 + t2
('a', 'b', 'c', 'd')
>>> 3 * t1
('a', 'b', 'a', 'b', 'a', 'b')
```

2.1.2 Appartenance

Pour tester l'appartenance d'un élément à un tuple, on utilise l'opérateur **in** :

```
>>> t = "a", "b", "c"
>>> "a" in t
True
>>> "d" in t
False
```

2.2 Utilisation des indices

Les indices permettent d'accéder aux différents éléments d'un tuple. Pour accéder à un élément d'indice i d'un tuple t , la syntaxe est $t[i]$.

L'indice i peut prendre les valeurs entières de 0 à $n-1$ où n est la longueur du tuple.

Cette longueur s'obtient en utilisant la fonction **len**. Exemple :

```
>>> t = "a", 1, "b", 2, "c", 3
>>> len(t)
6
>>> t[2]
'b'
```

La notation est celle utilisée avec les suites en mathématiques : $u_0, u_1, u_2, \dots$:

les indices commencent à 0 et par exemple le troisième élément a pour indice 2. Le dernier élément d'un tuple t a pour indice $\text{len}(t)-1$.

On accède ainsi au dernier élément avec $t[\text{len}(t)-1]$ qui peut s'abréger en $t[-1]$

```
>>> t = "a", 1, "b", 2, "c", 3
>>> t[-1]
3
>>> t[-2]
'c'
```

]

Exemple avec des tuples emboîtés (un tuple contenant des tuples)

```
>>> t = ("a","b"), ("c","d")
>>> t[1][0]
'c'
```

Explication : `t[1]` est le tuple ("c","d") et 'c' est l'élément d'indice 0 de ce tuple. Rappelons ce qui a été annoncé plus haut :

Avertissement :

Les éléments d'un tuple ne sont pas modifiables par une affectation de la forme : `t[i]=valeur` qui provoque une erreur et arrête le programme.

Ainsi :

```
>>> t[1]=("d","c")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

2.3 Affectation multiple

Prenons pour exemple l'affectation `a, b, c = 1, 2, 3`. Ceci signifie que le tuple `(a, b, c)` prend pour valeur le tuple `(1, 2, 3)`, autrement dit, les valeurs respectives des variables `a`, `b` et `c` sont 1, 2 et 3.

En particulier, l'instruction **`a, b = b, a` permet d'échanger les valeurs des deux variables `a` et `b`.**

Les valeurs des éléments d'un tuple peuvent ainsi être stockées dans des variables.

```
>>> t = 1, 2, 3
>>> a, b, c = t
>>> b
2
```

Cette syntaxe s'utilise souvent avec une fonction qui renvoie un tuple. Voici un exemple avec une fonction qui calcule et renvoie les longueurs des trois côtés d'un triangle ABC. La fonction prend en paramètres trois p-uplets représentant les coordonnées des trois points. On importe au préalable la fonction racine carrée `sqrt` du **module** `math` :

```
1
2 from math import sqrt
3
4 def longueurs(A,B, C):
5     xA,yA = A
6     xB, yB = B
7     xC,yC = C
8     dAB= sqrt((xB - xA)**2 +(yB - yA)**2)
9     dBC= sqrt((xC - xB)**2 +(yC - yB)**2)
10    dAC= sqrt((xC - xA)**2 +(yC - yA)**2)
11    return dAB,dBC, dAC
```

La fonction étant définie, nous l'utilisons dans l'interpréteur

```
>>> M = (3.4, 7.8)
>>> N = (5, 1.6)
>>> P = (-3.8, 4.3)
>>> dMN, dNP, dMP = longueurs(M, N, P)
>>> dMN
6.403124237432848
```

3 Les listes

3.1 Définition

Définition

Un objet de type **list**, que nous appelons une liste, ressemble à un p-uplet : un ensemble ordonné d'éléments avec des indices pour les repérer. Les éléments d'une liste sont séparés par des virgules et entourés de crochets.

Exemples :

```
>>>liste1 = ["a","b","c"] # une liste à 3 éléments
>>>liste2 = [1] # une liste contenant un seul élément
>>>liste3 = [[1, 2], [3, 4]] # une liste de listes
>>>liste_vide = [] # une liste vide
```

Notons la fonction **len** qui renvoie la longueur d'une liste, le nombre d'éléments de la liste :

```
>>> L=['loup','renard','chèvre']
>>> len(L)
3
```

3.1.1 Création d'une liste

Pour créer une liste d'entiers, nous pouvons utiliser les fonctions **list** et **range**.

```
>>> liste=list(range(2, 10, 3))
[2, 5, 8]
```

Dans l'exemple précédent, on forme la liste d'entiers entre 2 (inclus) et 10 (exclu) et pour passer d'un entier au suivant dans la liste on ajoute 3.

La méthode append() :

Pour ajouter les éléments un par un en fin de liste, nous utilisons une boucle et la méthode **append** :

```
>>> multiples_de_3= [ ]
>>> for i in range(100):
>>> ...multiples_de_3.append(3*i)
```

3.2 Construction par compréhension

L'instruction s'écrit sous la forme : **[expression(i) for i in objet]**. Ce type de construction est très spécifique au langage Python. En voici deux exemples :

```
multiples_de_3= [3 * i for i in range(100)]
multiples_de_6= [2 * n for n in multiples_de_3]
```

Exercice

1. Ecrire une liste qui contient les cubes des entiers présents dans la liste L.
2. Ecrire une liste L qui contient les 10 premiers entiers naturels non nuls.

Si on dispose d'une fonction f et d'une liste d'abscisses :

```
images= [f(x)for x in abscisses]
```

Un exemple avec des listes emboîtées :

```
>>> liste = [[i, j]for i in range(3)]for j in range(2)]
>>> liste
[[[0,0], [1, 0], [2, 0]], [[0, 1], [1, 1], [2, 1]]]
```

3.3 Utilisation

3.3.1 Accès aux éléments

```
>>>liste = ["a","b","c"]
>>>liste[1]
'b'
>>> liste =  [["a","b"], ["c","d"]]
>>> liste[1][0]
'c'
```

3.3.2 Méthodes

Le type d'une variable définit les valeurs qui peuvent être affectées à cette variable ainsi que les opérateurs et les fonctions utilisables. Les fonctions propres à un type donné sont appelées des **méthodes**.

La fonction **len** par exemple, s'applique aux chaînes de caractères, aux p-uplets, aux listes. La méthode **append**, présentée plus haut, est par contre propre aux listes.

Attention à la syntaxe :

on écrit **len(liste)**, mais **liste.append(*element*)**. Le nom de la variable est suivi d'un point puis du nom de la méthode.

Voici quelques autres méthodes sur les listes :

```
>>> liste = ["a","b","c"]
>>> liste.insert(1,"d")# insertion à l'indice 1 de l'élément "d"
>>> liste
['a','d','b','c']
>>> liste.remove("b")
>>> liste
['a','d','c']
>>> x = liste.pop()
>>> x
'c'
>>> liste
['a','d']
>>> liste.reverse()
>>>liste
['d','a']
>>>liste.sort()
>>>liste
['a','d']
```

Toutes ces méthodes modifient la liste initiale contrairement aux opérateurs de concaténation `+` et `*` avec lesquels une nouvelle liste est créée. Ces deux opérateurs s'utilisent comme avec les p-uplets.

Python Tutor: Visualize code in [Python](#), [JavaScript](#), [C](#), [C++](#), and [Java](#)



Notons qu'il est aussi possible de trier une liste sans la modifier avec la fonction **`sorted`** qui crée une nouvelle liste.

```

>>>liste = [5, 2, 7, 4]
>>>tri =sorted(liste)
>>>liste[5,2, 7, 4]
>>>tri[2,4, 5, 7]

```

3.3.3 Copie

Une liste peut donc être modifiée par une méthode. **On peut aussi modifier l'un de ses éléments par affectation**

```

>>> liste = ["a","b","c"]
>>> liste[1] ="d"
>>> liste
["a","d","c"]

```

Ceci oblige à une grande attention en particulier dans la création de copie d'une liste. Observons le code suivant

```

>>> ma_liste = ["a","b","c"]
>>> nouvelle_liste = ma_liste
>>> ma_liste[1] ="d"
>>> nouvelle_liste
["a","d","c"]

```

Dans cet exemple, une même liste a deux noms et `_nouvelle_liste` n'est pas une copie de `ma_liste`. Pour obtenir une copie, il faut créer une nouvelle liste. Par exemple

```

>>> ma_nouvelle_liste =list(ma_liste)

```

Voici un code écrit dans "Python tutor"

Python Tutor: Visualize code in [Python](#), [JavaScript](#), [C](#), [C++](#), and [Java](#)

Explications du code :

```

1
2 ma_liste = [4, 5, 9] # Crée une liste nommée 'ma_liste'
3                       #et stocke les nombres 4, 5, et 9 dans ma_liste.
4
5 nouvelle_liste = ma_liste # Crée une nouvelle liste nommée 'nouvelle_liste'
6                           # et lui assigne la même valeur que 'ma_liste'.
7                           #Ceci signifie que les deux listes pointent
8                           # vers la même adresse mémoire.
9
10 ma_liste[1] = 25 # Modifie le second élément de "ma_liste".
11                 #Comme 'nouvelle_liste' et 'ma_liste' pointent vers la même
12                 #adresse mémoire ceci affectera également 'nouvelle_liste'
13
14 ma_nouvelle_liste = list(ma_liste)
15 # Crée une nouvelle liste nommée 'ma_nouvelle_liste'
16 # en copiant tous les éléments de 'ma_liste'.
17 #Ceci crée une copie de telle sorte que tout changement des valeurs de '
18   ma_liste'
19   #affectera également 'ma_nouvelle_liste'

```

3.3.4 Autres manières de réaliser des copies de liste

Nous pouvons créer une copie d'une liste en insérant un à un les éléments à l'aide de la méthode `append()` :

Python Tutor: Visualize code in [Python](#), [JavaScript](#), [C](#), [C++](#), and [Java](#)

On peut également utiliser la méthode `copy` ou la méthode par tranches.
Mais ceci n'est pas recommandé dans les instructions officielles.