Chapitre 7 : Les types construits

1. Les listes (ou tableaux)

Une **liste** est un objet de type « `list` » dans lequel sont stocké un ensemble ordonné d'éléments accessibles avec des indices (ou index). Les éléments d'une liste sont séparés par des virgules et le tout encadré par deux 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
```

1.1. Création d'une liste

Une liste est déclarée par une série de valeurs (ne pas oublier les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des **virgules**, et le tout encadré par des **crochets**.

- **Liste vide**

Exemple :

```
>>> liste_vide = []  # une liste vide
```

- **Par duplication**

On peut créer une liste constituée d'un même élément ou d'une même séquence. Pour cela, on utilise l'opérateur Python * qui signifie duplication et non multiplication pour les variables de type liste.

Exemples :

```
>>> liste_de_0 = [0] * 10        # création par duplication de 0
>>> liste_de_0
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

>>> liste_de_123 = [1,2,3] * 3   # création par duplication d'un ensemble de 3 valeurs
>>> liste_de_123
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

- **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.

Exemples :

```
# création d'une liste d'entiers pairs de 0 à 15
>>> liste_entiers = [i for i in range(0,16,1) if (i % 2 == 0)]
>>> liste_entiers
[0, 2, 4, 6, 8, 10, 12, 14]

# création d'une liste des 10 premiers multiples de 3
>>> multiples_de_3 = [3 * i for i in range(10)]
>>> multiples_de_3
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]

# création d'une liste de listes
>>> 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]]]

# création d'une liste de dix 0
>>> liste = [0 for _ in range(10)]
>>> liste
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- **Liste de listes**

Il est tout à fait possible de construire une liste à partir de listes.

Exemples :

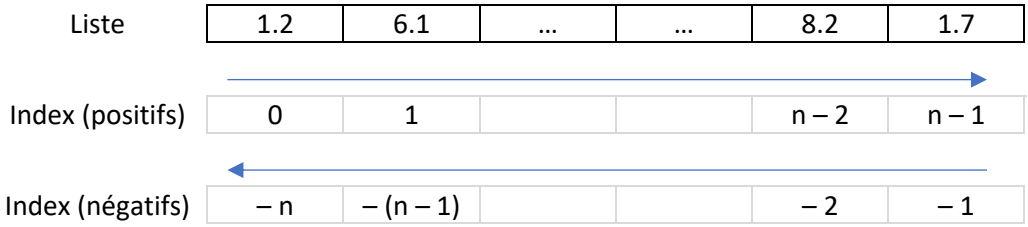
```
>>> seconde = ['Seconde 5', 34, 'Seconde 9', 24, 'Seconde 10', 22]
>>> premiere = ['Première NSI', 19]
>>> classe = [seconde, premiere]  # création de la nouvelle liste
>>> classe
[['Seconde 5', 34, 'Seconde 9', 24, 'Seconde 10', 22], ['Première NSI', 19]]
```

1.2. Accéder aux éléments d'une liste

Est associé à chaque item d'une liste un index qui peut prendre :

- Une valeur **positive** si l'on parcourt la liste de la **gauche vers la droite** : le premier élément commence à l'index 0.
- Une valeur **négative** si l'on parcourt la liste de **droite vers la gauche** : le premier élément commence à -1.

Exemple : Liste = [1.2, 6.1, …, 8.2, 1.7]



On accède à l'élément d'une liste grâce à la valeur de son index :

```
>>> Liste = [1.2, 6.1, 5.2, 3.8, 8.2, 1.7]
>>> Liste[0]
1.2
>>> Liste[-1]
1.7
>>> liste = [["a", "b"], ["c", "d"]]
>>> liste[1][0]
'c'
```

1.3. Opérations sur les listes

Quelques opérations sur les listes à connaitre :

Opérateur	Opération effectuée
len(L)	Donner le nombre d'élément dans la liste L
L.append(e)	Ajouter l'élément « e » à la fin de la liste L
del L[i]	Supprimer l'item d'index i de la liste L
L.remove("e")	Retire l'élément « e » de la liste L
L.reverse()	Inverse les éléments de la liste L (le dernier élément sera le premier, etc.)
L.sort()	Trie la liste L dans l'ordre croissant ou alphabétique
L.insert(index, value)	Permet d'ajouter un nouvel élément (value) à n'importe quel endroit (index) de la liste, pas seulement à la fin.
L1 + L2	Concaténation (mettre bout à bout) de 2 listes (L1 et L2)

Remarque :

```
# création d'une liste de dix listes vides
>>> liste_vide = [[]]*10
>>> liste_vide
[[], [], [], [], [], [], [], [], [], []]
>>> liste_vide[0].append(0)
>>> liste_vide
[[0], [0], [0], [0], [0], [0], [0], [0], [0], [0]]

# création d'une liste de 10 listes vides, par compréhension
>>> liste_vide_2 = [ [] for _ in range(10)]
>>> liste_vide_2
[[], [], [], [], [], [], [], [], [], []]
```

```
>>> liste_vide_2[0].append(0)
[[0], [], [], [], [], [], [], [], [], []]
```

→ Dans la première création de liste « `liste_vide` » (ligne 2), chaque élément pointe vers le même objet « liste » d'où la nouvelle liste (ligne 7), contrairement à la seconde méthode de création de liste « `liste_vide` » (ligne 10).

1.4. Modifier les éléments d'une liste

- Par addition de deux listes

```
>>> liste1 = [1, 2, 3, 4]
>>> liste2 = [5, 6, 7, 8, 9, 10]
>>> liste3 = liste1 + liste2
>>> liste3
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Par multiplication par un entier

```
>>> ['a'] * 5
['a', 'a', 'a', 'a', 'a']
```

- Par ajout d'éléments

```
#à la fin
>>> liste = ['a', 'b']
>>> liste.append('c')
>>> liste
['a', 'b', 'c']

#par insertion
>>> liste.insert(1, 'd')    #on insère la lettre 'd' à l'indice 1 : entre 'a' et 'b'
>>> liste
['a', 'd', 'b', 'c']
```

- Par remplacement d'éléments

```
>>> liste1 = [1, 2, 3, 4]
>>> liste1[1] = 'd'
>>> liste1
[1, 'd', 3, 4]
```

- Par suppression d'éléments

```
>>> liste1 = [1, 2, 3, 4]
>>> del liste1[1]
>>> liste1
[1, 3, 4]

>>> liste1.remove(4)
>>> liste1
[1, 3]

>>> liste2 = ['a', 'b', 'c']
>>> liste2.remove('c')
>>> liste2
['a', 'b']
```

1.5. Parcourir et récupérer les éléments d'une liste

La première solution consiste à parcourir les indices de la liste au moyen d'une boucle `for`, dont l'indice va de 0 à la longueur totale de la liste, pour accéder à chacune de ses valeurs.

```
maList = [10, 1, 8, 3, 5]
somme = 0

for i in range(len(maList)):
    somme += maList[i]        # calcul de la somme des éléments de la liste

print(somme)
```

La deuxième solution consiste à récupérer chaque élément de la liste grâce à l'opérateur `in`.

```
maList = [10, 1, 8, 3, 5]
somme = 0

for value in maList:
    somme += value        # calcul de la somme des éléments de la liste

print(somme)
```

Il est la possibilité de sélectionner une partie d'une liste en utilisant un indiçage construit sur le modèle `[m:n+1]` pour récupérer tous les éléments, du émième au énième (de l'élément `m` inclus à l'élément `n+1` exclu). On dit alors qu'on récupère une **tranche** de la liste.

```
>>> maList = [10, 1, 8, 3, 5]
>>> maList[0:2]
[10, 1]
>>> maList[0:]
[10, 1, 8, 3, 5]
>>> maList[:]
[10, 1, 8, 3, 5]
>>> maList[1:]
[1, 8, 3, 5]
>>> maList[1:-1]
[1, 8, 3]

>>> maList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> maList[1:6:2]   #le dernier chiffre indique le pas
[1, 3, 5]
>>> maList[::2]     #le dernier chiffre indique le pas
[0, 2, 4, 6, 8]
```

Cette opération s'appelle le **slicing** (saucissonage) :

```
liste[debut:fin:pas]
liste[debut:fin]
liste[debut:]
liste[:fin]
liste[::pas]
liste[debut::pas]
liste[:fin:pas]
```

1.6. Trier une liste

- Tri sans copie

```
>>> liste1 = [5, 2, 3, 1, 4]
>>> liste1.sort()
>>> liste1
[1, 2, 3, 4, 5]
```

- Tri avec copie

```
>>> liste2 = sorted(liste1)
[1, 2, 3, 4, 5]
>>> sorted(liste1)
[1, 2, 3, 4, 5]
```

- Tri avec une clé

Un usage fréquent est de faire un tri sur des objets complexes en utilisant les indices des objets en tant que clé : les fonctions de tri peuvent admettre un deuxième argument key qui spécifie une fonction à un seul argument fournissant une *clef* de comparaison de la liste à trier.

```
>>> sorted([(1, 'b'), (3, 'a'), (2, 'c')], key = lambda x: x[1])
[(3, 'a'), (1, 'b'), (2, 'c')]
```

→ Tri d'une liste de tuples (voir $2) à partir de leur élément d'indice 1 : key = **lambda** x: x[1].

Par défaut, les tuples sont triés selon la valeur de leur 1er élément :

```
>>> sorted([(1, 'b'), (3, 'a'), (2, 'c')])
[(1, 'b'), (2, 'c'), (3, 'a')]
```

1.7. Inverser une liste

```
>>> liste = ['quatre', 'deux', 6, 5, 3, 1]
>>> liste.reverse()
>>> liste
[1, 3, 5, 6, 'deux', 'quatre']
```

2. Les tuples

On vient de voir qu'il était possible de « stocker » plusieurs grandeurs dans une même structure, ce type de structure est appelé une **séquence**, c'est-à-dire un ensemble fini et ordonné d'éléments dont les indices vont de 0 à n – 1 (si cette séquence comporte n éléments). On accède à ces éléments par le biais de leur **indice**.

Un objet de type `tuple`, ou un `p-uplet`, est une séquence dont les éléments peuvent être chacun de n'importe quel type. **On parlera indifféremment de p-uplet ou de tuple**.

ATTENTION : une fois le tuple défini, ces valeurs sont immuables, ses éléments ne sont pas modifiables par une affectation de la forme t[i]= valeur.

2.1. Création d'un p-uplet

Exemple	Commentaire
`t = ("a", "b", "c", 3)`	Création d'un tuple à 4 éléments
`t = "a",`	Création d'un tuple à 1 élément (attention à la virgule : elle est indispensable)
`t = ()`	Création d'un tuple à 0 élément (ici, pas de virgule, mais des parenthèses)
`t = 3, ("a", "b", c")`	Création d'un tuple à 2 éléments dont le second est un tuple (entre parenthèses)
`t = 4, 'b', False`	Création d'un tuple dont l'un des éléments est un booléen
`t = (1.0, False, (1, 2))`	Création d'un tuple avec la combinaison d'un float, d'un booléen et d'un tuple

→ Une chaine de caractères est un tuple constitué de chacune de ses lettres.

2.2. Opérations sur les p-uplet

Il existe deux opérateurs de concaténation, qui s'utilisent comme avec les chaînes de caractères, ce sont les opérateurs « + » et « * » qui créent des nouveaux p-uplets.

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

2.3. Vérifier l'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
```

2.4. Accès à un élément

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 (obtenue avec la fonction `len()`) du tuple.

- 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 **len(t) – 1**. On accède ainsi au dernier élément avec **t[len(t) – 1]** qui peut s'abréger en **t[–1]**.

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

Cas des tuples emboités (un tuple contenant des tuples) :

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

→ `t[1]` est le tuple `("c", "d")` et `'c'` est l'élément d'indice **0** de ce tuple.

2.5. Affectation multiple

Il est possible de créer un tuple par affectation multiple :

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

Exemple d'utilisation :

```
def multiples(n):
    return (2 * n, 3 * n, 4 * n)     #renvoie le double, le triple et le quadruple de n

a,b,c = multiples(5)
```

→ La fonction `multiples(n)` renvoie un tuple : les variables `a`, `b` et `c` contiendront chaque valeur du tuple renvoyé.

3. Les dictionnaires

Les éléments d'une liste sont repérés par des indices 0, 1, 2, ... Une différence essentielle avec un dictionnaire, objet de type « `dict` », est que les indices sont remplacés par des objets du type « `str` », « `float` », « `tuple` » (si les n_uplets ne contiennent que des entiers, des flottants ou des p-uplets). On les appelle des clés et à chaque `clé` correspond une `valeur`.

⇒ Les éléments d'un dictionnaire sont des couples « clé-valeur ». Ces clés ne sont pas ordonnées et les valeurs peuvent être de n'importe quel type.

Exemple : `monDico = {'cle1' : 1.56, 'a' : "NSI", 5 : 5}`

3.1. Création d'une liste

Une liste est déclarée par une série de valeurs (ne pas oublier les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des **virgules**, et le tout encadré par des **crochets**.

- **Dictionnaire vide**

Un dictionnaire est créé avec des accolades, les différents couples étant séparés par des virgules. La clé et la valeur correspondante d'un élément sont séparées par deux points.

Exemple :

```
>>> monDico = {}  # un dictionnaire vide
>>> type(monDico)
<class 'dict'>
```

- **Dictionnaire contenant déjà des éléments**

```
>>> monDico = {'cle1' : 1.56, 'a' : "NSI", 5 : 5}
>>> monDico
{'cle1': 1.56, 'a': 'NSI', 5: 5}
>>> type(monDico)
<class 'dict'>
```

3.2. Accéder aux éléments d'un dictionnaire

La syntaxe est proche de celle utilisée pour obtenir des éléments d'une liste :

```
>>> monDico = {'cle1' : 1.56, 'a' : "NSI", 5 : 5}
>>> monDico['a']
'NSI'
```

→ On accède aux valeurs en spécifiant leur clé (ici `'a'`).

3.3. Opérations sur les dictionnaires

- **Récupérer une valeur**

```
>>> monDico = {' cle1' : 1.56, 'a' : "SNT", 5 : 5}
>>> monDico['a']
'NSI'
>>> monDico.get('a')
'NSI'
>>> monDico.get('test', "Clé inconnue")  #on peut spécifier le message à afficher si la clé n'existe pas
'Clé inconnue'
```

- **Modifier la valeur d'un élément**

```
>>> monDico = {'cle1' : 1.56, 'a' : "SNT", 5 : 5}
>>> monDico['a'] = "NSI"  #on indique la clé dont on veut modifier la valeur
>>> monDico
{'cle1': 1.56, 'a': 'NSI', 5: 5}
```

- **Tester la présence d'une clé**

```
>>> monDico = {'Classe1' : 1.56, 'a' : "SNT", 5 : 5}
>>> 5 in monDico
True
>>> "SNT" in monDico
False
>>> "a" in monDico
True
```

- **Ajouter des éléments**

```
>>> monDico = {'cle1' : 1.56, 'a' : "SNT", 5 : 5}
>>> monDico['Année'] = 2022
>>> monDico
{'cle1': 1.56, 'a': 'NSI', 5: 5, 'Année': 2022}
```

→ La clé « `Année` » est automatiquement créée.

- **Taille d'un dictionnaire**

```
>>> monDico = {'cle1' : 1.56, 'a' : "SNT", 5 : 5}
>>> len(monDico)
3
```

- **Supprimer une entrée**

```
>>> monDico = {'cle1': 1.56, 'a': 'NSI', 5: 5, 'Année': 2022}
>>> del monDico['Année']
>>> monDico
{'cle1': 1.56, 'a': 'NSI', 5: 5}
```

- **Fusion de deux dictionnaires avec la méthode « `.update()` »**

```
>>> monDicoEleve = {'Rémi' : "1E3", 'Alice' : "1E4", 'Emilie' : "1E2"}
>>> monDicoNSI = {'Mathéo' : "1E4", 'Axek' : "1E5"}
>>> monDicoNSI.update(monDicoEleve)
>>> monDicoNSI
{'Mathéo': '1E4', 'Axek': '1E5', 'Rémi': '1E3', 'Alice': '1E4', 'Emilie': '1E2'}
```

- **Obtenir l'ensemble des clés avec la méthode « `.key()` »**

```
>>> monDico = {'cle1' : 1.56, 'a' : "SNT", 5 : 5}
>>> monDico.keys()
dict_keys(['cle1', 'a', 5])
```

- **Obtenir l'ensemble des valeurs avec la méthode « `.values()` »**

```
>>> monDico = {'cle1' : 1.56, 'a' : "SNT", 5 : 5}
>>> monDico.values()
dict_values([1.56, 'NSI', 5])
```

- **Obtenir l'ensemble des éléments, sous la forme de tuples (clé, valeur), avec la méthode « `.items()` »**

```
>>> monDico = {'cle1' : 1.56, 'a' : "SNT", 5 : 5}
>>> monDico.items()
dict_items([('cle1', 1.56), ('a', 'NSI'), (5, 5)])
```

- **Accéder à toutes les clés ou à toutes les valeurs ou à tout le contenu**

```
#accès à toutes les clés
>>> for cle in monDico.keys():
        print(cle)
cle1
a
5

#accès à toutes les clés
>>> for val in monDico.values():
        print(val)
1.56
NSI
5

#accès à tout le contenu du dictionnaire
>>> for item in monDico.items():
        print(item)
('cle1', 1.56)
('a', 'NSI')
(5, 5)
```

Programme :

Contenus	Capacités attendues	Commentaires
p-uplets. p-uplets nommés	Écrire une fonction renvoyant un p-uplet de valeurs.	
Tableau indexé, tableau donné en compréhension	Lire et modifier les éléments d'un tableau grâce à leurs index. Construire un tableau par compréhension. Utiliser des tableaux de tableaux pour représenter des matrices : notation a [i] [j]. Itérer sur les éléments d'un tableau.	Seuls les tableaux dont les éléments sont du même type sont présentés. Aucune connaissance des tranches (*slices*) n'est exigible. L'aspect dynamique des tableaux de Python n'est pas évoqué. Python identifie listes et tableaux. Il n'est pas fait référence aux tableaux de la bibliothèque NumPy.
Dictionnaires par clés et valeurs	Construire une entrée de dictionnaire. Itérer sur les éléments d'un dictionnaire.	Il est possible de présenter les données EXIF d'une image sous la forme d'un enregistrement. En Python, les p-uplets nommés sont implémentés par des dictionnaires. Utiliser les méthodes *keys*(), *values* () et *items* ().

Sitographie :

- https://www.jcbmathsandco.fr/nsi-numerique-et-sciences-informatiques-representation-des-donnees-types-construits/
- https://pixees.fr/informatiquelycee/n_site/nsi_prem_pythonSequence.html
- https://www.monlyceenumerique.fr/nsi_premiere/donnees_construites_dc/dc1_tuple_et_liste.php
- https://courspython.com/tuple.html
- https://www.silanus.fr/nsi/premiere/listes/liste.html
- https://info.blaisepascal.fr/nsi-les-listes