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 tableaux.

I. 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]
```

Le nom *mon_tuple* est associé au tuple (5, 8, 6, 9), le nom *a* est associé 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".

- Un tuple ne contient pas forcément des nombres entiers, il peut aussi contenir des nombres décimaux, des chaînes de caractères, des booléens...
- Grâce au tuple, une fonction peut renvoyer plusieurs valeurs :
- La console permet d'afficher les éléments présents dans un tuple simplement en saisissant le nom associé à un tuple dans cette console.
- Il est possible d'associer à des noms les valeurs contenues dans un tuple, par exemple :
  ```
  a, b, c, d = (5, 8, 6, 9)
  ```
- Il est possible de tester directement l'appartenance d'un élément ou d'accéder à celui-ci grâce à la commande in.
  ```
  a=(3,89,-4)
  3 in a
  ```
- On peut créer un tuple en concaténant d'autres tuples grâce l'opérateur +.

II. 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)
- 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
  ```
- Comme vous pouvez le constater avec l'exemple ci-dessus, 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)
  ```
- 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]
  ```
- La fonction "*len*" permet de connaître le nombre d'éléments présents dans une séquence (tableau et tuple)
  ```
  mon_tab = [5, 8, 6, 9]
  nb_ele = len(mon_tab)
  ```

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.

1. la boucle "for" : parcourir les éléments d'un tableau

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

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

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.
element va être associé :

au premier tour de boucle, au premier élément du tableau (l'entier 5)
au deuxième tour de boucle, au deuxième élément du tableau (l'entier 8)
au troisième tour de boucle, au troisième élément élément du tableau (l'entier 6)
au quatrième tour de boucle, au quatrième élément de le tableau (l'entier 9)

Une chose importante à bien comprendre : le choix du nom de la variable qui va associé aux éléments du tableau 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 toto in mon_tab:
    print (toto)
```

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

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

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.

Il est possible d'utiliser la méthode "range" pour "remplir" un tableau :

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

2. 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 = [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 tableau :

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

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]
```

autre possibilité, utiliser des composants "arithmétiques" :

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

3. 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)

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

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

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)
```

III. Les chaînes de caractères

Les chaînes de caractères ont été introduites comme un type de base. Elles ont cependant certaines caractéristiques des tableaux :

- on peut obtenir leur longueur grâce à *len*
- on peut accéder aux caractères de la chaîne avec la notation indexée
- on peut énumérer les caractères d'une chaîne avec une boucle *for*

cependant, les chaînes sont immutables ; on ne peut pas modifier leur contenu

IV. 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": "Durand", "prenom": "Christophe", "date de naissance": "29/02/1981"}
```

- 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 tableaux et les parenthèses pour les tuples).
- Dans le dictionnaire ci-dessus, "nom", "prenom" et "date de naissance" sont des clés et "Durand", "Christophe" et "29/02/1981" sont des valeurs.
- La clé "nom" est associée à la valeur "Durand", la clé "prenom" est associée à la valeur "Christophe" et la clé "date de naissance" est associée à la valeur "29/02/1981".
- Les clés sont des éléments non modifiables et doivent être uniques. Elles ont pour types des objets immutables comme les entiers, les chaînes de caractères, les tuples.
- Les valeurs associées au clés sont elles modifiables et peuvent être de n’importe quel type.
- Un dictionnaire est non ordonné.
- Un dictionnaire est modifiable, on peut modifier ses valeurs, ajouter ou supprimer des éléments après sa création.

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

```
mon_dico = {}
mon_dico["nom"] = "Durand"
mon_dico["prenom"] = "Christophe"
mon_dico["date de naissance"] = "29/02/1981"
```

Le "*mon_dico*" est associé à un dictionnaire. Il est possible d'afficher le contenu du dictionnaire associé au nom *mon_dico* en saisissant *mon_dico* dans la console.

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

```
mon_dico = {"nom": "Durand", "prenom": "Christophe", "date de naissance": "29/02/1981"}
print(f'Bonjour je suis {mon_dico["prenom"]} {mon_dico["nom"]}, je suis né le {mon_dico["date de naissance"]}')
```

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

```
mon_dico = {"nom": "Durand", "prenom": "Christophe", "date de naissance": "29/02/1981"}
```

```
mon_dico['lieu naissance'] = "Bonneville"
```

L'instruction "del" permet du supprimer une paire "clé/valeur"

```
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
```

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)
```

À noter que le ".keys()" n'est pas obligatoire pour parcourir les clés :

```
mes_fruits = {"poire": 3, "pomme": 4, "orange": 2}
for d in mes_fruits:
     print (d)
```

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 items().

```
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"