Python Types construits

Cours / TD

PARTIE 1 : Types construits

Nous avons manipulé des variables avec des types simples : int, bool, float, str. Ce sont des conteneurs ne contenant qu'une valeur.

Nous avons manipulé aussi, sans vraiment l'étudier, des types construits (ou types abstraits) : liste, tuple. Ces types de données s'appellent des séquences.

Dans ce cours nous allons étudier trois types de séquence : les p-uplets, les tableaux et les dictionnaires. En python, les tableaux s'appellent les listes et les p-uplets s'appellent les tuples.

DÉFINITION 1 : Séquence

Famille indexée d'éléments par les entiers strictement positifs inférieurs ou égaux à un certain entier, ce dernier étant appelé « longueur » de la séquence.

En informatique, une liste est une structure de données représentant une séquence finie d'éléments auxquels on peut accéder efficacement par leur position, ou indice, dans la séquence.

En Python, une séquence est une collection ordonnée d'objets qui permet d'accéder à ses éléments par leur numéro de position dans la séquence. Les listes et les tuples sont des séquences.

1. Les types construits : les p-uplets – tuples en Python

DÉFINITION 2 : p-uplet

Un p-uplet est une séquence **immutable**. C'est à dire une suite indexée de valeurs (de n'importe quel type) que l'on ne peut pas modifier.

1.1. Construire les objets tuples

PROPRIÉTÉ 1 :

Un tuple est une séquence d'éléments entre parenthèses séparés par des virgules.

✔ EXEMPLE 1 :

```
tuple1 = ()          # Tuple vide
tuple2 = (3,)        # Tuple avec un seul element
tuple3 = (1,4,5,6,2,7)       # Tuple de 6 elements numerotes de 0 a 5
tuple4 = (1,'a','voiture')   # Tuple d'elements de differents types
type(tuple1)         # permet d'acceder au type de l'objet
```

.\scripts\part1\exemple1.py

```
*** Console de processus distant Réinitialisée ***
>>> type(tuple4)
<class 'tuple'>
>>> type(tuple3)
<class 'tuple'>
```

Remarque : Le mot tuple étant le nom d'une fonction interne, vous devez éviter de l'utiliser comme nom de variable.

PROPRIÉTÉ 2 : Opérations simples sur les tuples

Différentes opérations de base sont disponibles pour les tuples :

- La **concaténation**. Deux tuples peuvent être collés l'un à l'autre avec la commande **+**
- La **reproduction** de tuples. Un tuple peut être "multiplié" avec la commande *****

✔ EXEMPLE 2 :

```
>>> tuple1 = (1 ,2 ,3)
>>> tuple2 = (5 ,6)
>>> tuple1 + tuple2              # concatenation de deux tuples
(1, 2, 3, 5, 6)
>>>
>>> tuple4 = tuple1 * 3          # "multiplication d'un tuple
>>> tuple4
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>>
>>> print(* tuple4)              # Dispersion d'un tuple
1 2 3 1 2 3 1 2 3
```

.\scripts\part1\exemple2.png

PROPRIÉTÉ 3 : Indices des tuples

Soit **i** un entier et **monTuple** un tuple. On accède à un élément du tuple en écrivant : **monTuple[i]**

```
>>> monTuple = (5,1,3,0,2)
>>> monTuple [0]            # Acces au premier element du tuple
5
>>> monTuple [3]            # Acces au 4eme element du tuple (indice 3)
0
>>> monTuple [-1]           # Acces au dernier element du tuple
2
>>> monTuple [1:4]          # slicing: Affiche de monTuple [1] à monTuple [3]
(1, 3, 0)
>>> monTuple [2:]           # slicing: Affiche de monTuple [2] à la fin
(3, 0, 2)
>>> monTuple [:2]           # slicing: Affiche du debut a monTuple [1]
(5, 1)
```

✍ Exercice 1 : Créer une fonction nommée coorVecteur( ) prenant 4 paramètres xA, yA, xB, yB et renvoyant sous forme de tuple les coordonnées du vecteur $\overrightarrow{AB}$

```
>>> coorVecteur(2,1,5,8)
(3, 7)                          # En effet x_AB=5 -2=3 et y_AB=8 -1=7
```

.\scripts\part1\exo1.py

✍ *Exercice 2 : En utilisant la fonction, coorVecteur( ) créer une fonction multVecteurk( ) créant les coordonnées du vecteur k*

```
>>> multVecteurk(5,2,1,5,8)
(15, 35)                        # car x_AB=3 et y_AB=7
```

.\scripts\part1\exo2.py

1.2. Itérer un tuple

PROPRIÉTÉ 4 : Un tuple est itérable. Cela signifie que l'on peut organiser une itération (boucle for) sur cette structure.

✔ *EXEMPLE 3 :*

```
monTuple = (2,4,5,8,6)          # Initialisation du tuple
n = len(monTuple)               # n est la longueur du tuple, ici n=5
for i in range(n):              # Balayage du tuple de i=0 a i=4
    print(monTuple[i], end=" ") # Affichage des elements du tuple
print()                         # Retour à la ligne

for elt in monTuple:            # Balayage du tuple par des elements
    print(elt, end= " ")        # Affichage des elements du tuple9 >>>
```

```
*** Console de processus distant Réinitialisée ***
2 4 5 8 6
2 4 5 8 6
```

Dans les deux cas, le résultat est le même

Exercice 3 : Créer une fonction `affichageTuple( )` prenant comme paramètre un tuple et affiche sous forme de tuple chaque indice et sa valeur correspondante. On utilisera des boucles pour balayer le tuple.

```
>>> monTuple = (4 ,3 ,6)
>>> affichageTuple(monTuple)
0 4
1 3
2 6
```

.\scripts\part1\exo3A.py

Il est possible de récupérer la longueur d'un tuple avec l'instruction suivante :

```
>>> monTuple = (1,3,4,2,8)
>>> len(monTuple)
5
```

.\scripts\part1\exo3B.png

Dans cet exemple, le tuple a une longueur de 5 avec des indices allant de 0 à 4.

Exercice 4 : Sans utiliser l'instruction `len( )`, créer une fonction `longueurTuple( )` qui renvoie la longueur d'un tuple.

```
>>> monTuple = (1,3,4,2,8)
>>> longueurTuple(monTuple)
5
```

.\scripts\part1\exo4.py

Exercice 5 : Créer une fonction `affichagePair( )` qui affiche uniquement les éléments pairs du tuple.

```
>>> monTuple = (1,3,4,2,8,7,1,6)
>>> affichagePair(monTuple)
4 2 8 6
```

.\scripts\part1\exo5.py

Exercice 6 : Créer une fonction verifElement( ) qui vérifiera si une valeur appartient ou non à un tuple

```
>>> monTuple = (2,4,1,8,7)
>>> verifElement(4,monTuple)
1                          # 4 est bien dans monTuple. On affiche l'indice trouve
>>> verifElement(5,monTuple)
5 False                    # 5 n'appartient pas à monTuple
```

.\scripts\part1\exo6.py

1.3. Opération et méthodes communes aux listes et aux tuples

Opérations ou méthodes	Description
`x in L`	Renvoie True si un élément de L est égale à x, False sinon
`x not in L`	Renvoie True si aucun un élément de L n'est égale à x, False sinon
`len(L)`	Renvoie le nombre d'éléments de L
`L == L1`	Renvoie True si L et L1 sont de même type, ont la même longueur, et ont des éléments égaux deux à deux.
`L[i]`	Renvoie l'élément d'indice i de L. Le premier élément a pour indice 0.
`L[i:j]`	Renvoie une partie de l'indice i à j non inclus
`L.index(x)`	Renvoie l'indice de la première apparition de x dans L
`L.count(x)`	Renvoie le nombre d'apparitions de x dans L
`L+L1`	Renvoie une nouvelle séquence concaténation de L et L1.
`L*n`	Renvoie une nouvelle séquence composée de la concaténation de L avec lui-même n fois.

Exercice 7 : Tester toutes ces opérations et méthodes avec :

```
1 >>> L = (5,4,8,9,1,4)
2 >>> L1 = (4 ,3 ,1)
3 >>> x = 9
4 >>> n = 3
```

.\scripts\part1\exo7.py

✍ *Exercice 8 : Créer une fonction `maxiTuple( )` qui prendra comme paramètre un p-uplet et qui renverra la valeur de l'élément le plus grand.*

```
>>> monTuple = (1,0,-15,8,-6,1,6,4,-2,3,1)
>>> maxiTuple(monTuple)
8
```

.\scripts\part1\exo8.py

✍ *Exercice 9 : Créer une fonction `comptage( )` qui prendra comme paramètre un p-uplet et une valeur et qui comptera le nombre d'éléments de cette valeur dans le p-uplet.*

```
>>> monTuple = (1,0,-15,8,-6,1,6,4,-2,3,1)
>>> comptage(monTuple ,1)
3
```

.\scripts\part1\exo9.py

✍ *Exercice 10 : Créer une fonction `sommePlus( )` qui prendra comme paramètre un p-uplet et qui renverra la somme des valeurs strictement positives.*

```
>>> monTuple = (1,0,-15,8,-6,1,6,4,-2,3,1)
>>> sommePlus(monTuple)
24
```

.\scripts\part1\exo10.py

✍ Exercice 11 : Modifier le programme de l'exercice précédent afin de ne renvoyer que la somme des valeurs paires positives. Nommer cette fonction `sommePlus2( )`

```
>>> monTuple = (1,0,-15,8,-6,1,6,4,-2,3,1)
>>> sommePlus2(monTuple)
18
```

.\scripts\part1\exo11.py

2. Les types construits : les listes

PROPRIÉTÉ 5 :

- Une liste a exactement les mêmes propriétés que les tuples
- Il est possible d'appliquer toutes les opérations et méthodes vues pour les tuples
- Une liste est une séquence mutable. On va donc pouvoir ajouter des propriétés aux listes

DÉFINITION 3 : p-uplet

- Une liste est définie à l'aide de crochets. Exemple : `maListe = [2,4,1,3,8]`
- Il est possible d'accéder aux éléments de la liste maListe[i]

✔ *EXEMPLE 4 :*

```
>>> maListe = []                    # Creation d'une liste vide
>>> maListe = [1,2,4,3,8,7,5]
>>> maListe[2] = 6                  # Réaffectation de l'element d'indice 2 -->
[1,2,6,3,8,7,5]
>>> maListe.append(9)               # Ajout d'un element en fin de liste -->
[1,2,6,3,8,7,5,9]
>>> del maListe[2]                  # Suppression de l'element d'indice 2 -->
[1,2,3,8,7,5,9]
>>> maListe.remove(8)               # Suppression de la premiere occurence de 8 -->
[1,2,3,7,5,9]
>>> maListe.pop()                   # Suppression du dernier element de la liste -->
[1,2,3,7,5]
>>> maListe.insert(2,15)            # Insere la valeur 15 a l indice 2 -->
[1,2,15,3,7,5]
>>> maListe.reverse()               # Modifie la liste en inversant l'ordre -->
[5,7,3,15,2,1]
>>> min(maListe)                    # Recherche la valeur min
1
>>> max(maListe)                    # Recherche la valeur max
15
```

.\scripts\part1\listes.py

Exercice 12 : Dans cet exercice, on prendra la liste `L = [4,2,1,3,4,6]`

1. Insérer la valeur 5 à l'indice 4
2. Supprimer la valeur d'indice 3
3. Supprimer la deuxième valeur 4
4. Insérer la valeur 15 à l'indice 2

.\scripts\part1\listes.py

2.1. Tri d'une liste

Pour trier une liste, plusieurs possibilités :

```
>>> maListe = [1,2,4,3,8,7,5]
>>> maListe.sort()            # maListe est modifiee
[1, 2, 3, 4, 5, 7, 8]

>>> maNouvelleListe = sorted(maListe,reverse = False)   # maListe n'est pas modifiee

>>> maNouvelleListe = sorted(maListe,reverse = True)    # ordre decroissant
```

.\scripts\part1\tri.py

2.2. Le hasard

Pour avoir accès aux fonctions suivantes, il faut importer la bibliothèque random :

✔ EXEMPLE 5 :

```
from random import *

>>> maListe = [1,3,8,0,7,2]
>>> choice(maListe)        # choisit au hasard 1 elemnt de la liste
8
>>> shuffle(maListe)       # melange la liste. Liste modifiee
>>> maListe
[7, 1, 0, 3, 8, 2]
>>> sample(maListe,2)      # Choix de 2 elements au hasard
[8, 1]
```

.\scripts\part1\hasard.py

2.3. Transtypage

Il est possible de changer le type des variables, on parle de transtypage.

```
>>> a = 123
>>> type(a)
<class 'int'>          # a est un entier par defaut
>>> a = str(a)         # conversion en chaine de caractere --> '123'
>>>type(a)
<class 'str'>

>>> a = list(a)        # conversion en liste --> ['1', '2', '3']
>>>type(a)
<class 'list'>

>>> a = tuple(a)       # conversion en tuple --> ('1', '2', '3')
>>>type(a)
<class 'tuple'>
```

.\scripts\part1\transtypage.py

2.4. Génération d'une liste

Il y a plusieurs moyens de générer une liste :

```
L = []                      # Liste vide
n = 10                      # longueur de la liste
for i in range(n):          # Construction par extension
   L.append(2**i)           # ajout de la valeur 2**i à droite

>>> L                       # contenu de L
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

.\scripts\part1*geneliste1*.py

```
M = [2**i for i in range(n)]        # Construction par comprehension

>>> M                               # contenu de M
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

.\scripts\part1*geneliste2*.py

✍ Exercice 13 : Dans cet exercice, vous devez écrire deux scripts Python qui créent la liste des entiers de 0 à 10000 de deux méthodes différentes : par extension et par compréhension.

.\scripts\part1\exo13.py

✍ Exercice 14 : Créer en compréhension une liste qui contienne les nombres pairs inférieurs à 10000.

.\scripts\part1\exo14.py

✍ Exercice 15 : Créer une liste contenant les puissances de 2 jusqu'à 2^{10}

.\scripts\part1\exo15.py

✍ Exercice 16 : Créer une liste par compréhension contenant 30 valeurs aléatoires entre 0 et 20.

.\scripts\part1\exo16.py

✍ Exercice 16Bis : Créer une liste par extension contenant 30 valeurs aléatoires (au maximum) entre 0 et 20 jusqu'à l'obtention d'un 0.

.\scripts\part1\exo16bis.py

3. Les listes de listes (tableau de tableaux) ou matrices

Les éléments d'une liste peuvent être également une liste (les éléments d'un tableau peuvent être également un tableau).
Nb : Les tableaux sous Python sont appelés listes, ils diffèrent des tableaux que l'on trouve dans d'autres langage de programmation par plusieurs aspects (agrandis / rétrécis du côté droit ...). Nous dirons qu'un tableau est une suite finie de valeurs d'un même type (entier, flottant, booléen ...), stockées dans des cases mémoires contigües. On parle de liste en Python, les éléments d'une liste Python peuvent néanmoins avoir des types différents. Un tableau est une liste ordonnée de nombres uniquement (flottants, entiers, complexes voire booléens codés en binaire). Les tableaux sont accessibles à partir de la librairie Numpy qui doit être préalablement chargée.

DÉFINITION 4 : Matrice

On appelle matrice un tableau de tableaux dont chaque tableau à la même longueur.

Chaque élément d'une matrice A est noté $a_{i,j}$ où i est le numéro de ligne et j le numéro de colonne.

En mathématiques on représente une matrice de taille n,m ainsi :

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

En python une matrice est une liste de listes de même longueur.

PROPRIÉTÉ 6 : Pour accéder à un élément organisé en liste de liste, on utilise une notation avec un double crochets. Le premier indice pointe la ligne et le deuxième indice pointe la colonne. Si notre matrice contient n listes de m éléments on peut la voir ainsi :

$$List = \begin{pmatrix} List[0][0] & List[0][1] & \cdots & List[0][n-1] \\ List[1][0] & List[1][1] & \cdots & List[1][n-1] \\ \vdots & \vdots & \ddots & \vdots \\ List[m-1][0] & List[m-1][1] & \cdots & List[m-1][n-1] \end{pmatrix}$$

Pour accéder à l'élément j de la ième liste de L on écrira :

```
>>> L[i][j]
```

✔ EXEMPLE 6 :

```
>>> L = [[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]]
>>> L[1][2]                 # element d'indice 2 de la sous liste d'indice 1
8
>>> L[2]                    # sous liste 2
[11,12,13,14,15]
```

✍ Exercice 17 : A partir de la liste L précédente,

1. Afficher la deuxième ligne
2. Afficher la valeur 14
3. Afficher la longueur de L
4. Afficher la longueur de la première sous liste

3.1. Itérer sur une matrice

Pour itérer sur une matrice il faudra deux compteurs, un premier pour les lignes et un deuxième pour les colonnes.

✔ EXEMPLE 7 :

```
L = [[1 ,2 ,3 ,4 ,5] ,[6 ,7 ,8 ,9 ,10] ,[11 ,12 ,13 ,14,15]]

# Iterer avec les indices
m = len(L)                    # Longueur de la liste L
n = len(L[0])                 # Longueur des sous listes
for i in range(m):            # Balayage des sous listes
   for j in range(n) :        # Balayage des elements des sous listes
      print(L[i][j], end=" ")  # Affichage de l'element
```

.\scripts\part1\exemple7.py

```
*** Console de processus distant Réinitialisée ***
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

```
L1 = [[1 ,2 ,3 ,4 ,5] ,[6 ,7 ,8 ,9 ,10] ,[11 ,12 ,13 ,14,15]]
# Iterer avec les elements
for sousListe in L:                    # Balayage des sous listes
        for element in sousListe:      # Balayage des elements des sous listes
                print(element , end= "␣")          # Affichage de l'element

>>> # Affichage
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

```
*** Console de processus distant Réinitialisée ***
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

✍ Exercice 18 : Écrire un script permettant d'afficher chaque élément de la liste par colonne :

```
A = [[1 ,2 ,3] ,[4 ,5 ,6] ,[7,8,9]]

>>>
 1 4 7 2 5 8 3 6 9
```

.\scripts\part1\exo18.py

✍ Exercice 19 : A partir de la liste suivante, créer une nouvelle liste en échangeant les colonnes.

```
B = [[4 ,5] ,[2 ,9] ,[3 ,7]]

 >>>
[[5 ,4] ,[9 ,2] ,[7 ,3]]
```

.\scripts\part1\exo19.py

✍ Exercice 20 : Calculer la somme des termes d'une matrice

```
C = [[4 ,5] ,[2 ,9] ,[3 ,7]]
>>>
30
```

.\scripts\part1\exo20.py

✍ Exercice 21 : Déterminer la valeur la plus grande de cette matrice sans utiliser de méthodes.

```
D = [[1 ,4] ,[9 ,5] ,[7 ,2]]
>>>
9
```

.\scripts\part1\exo21.py

✍ Exercice 22 : Créer une matrice contenant les parités des éléments présents dans la matrice (True pour pair et False pour impair).

```
E = [[2 ,1 ,3] ,[4 ,8 ,6] ,[7,5,9]]
>>>
[[True ,False ,False][True ,True ,True][False ,False ,False]]
```

.\scripts\part1\exo22.py

✍ Exercice 23 : Créer une fonction fabriquant une matrice contenant la somme des éléments de chaque ligne jusqu'à l'indice courant.

```
F = [[4 ,2 ,1] ,[7 ,9 ,0] ,[3,5,8]]
 >>>
[[4 ,6 ,7] ,[7 ,16 ,16] ,[3 ,8 ,16]]
```

.\scripts\part1\exo23.py

✍ Exercice 24 : Un carré magique d'ordre 3 est une matrice de 3 lignes et trois colonnes dont la somme des lignes, des colonnes et des diagonales fait le même nombre.
Écrire une fonction `verifMagique( )` qui vérifie si un carré est un carré magique.

```
>>> L =   [[2,7,6],
           [9,5,1],
           [4 ,3 ,8]]

>>> verifMagique(L)
True
```

.\scripts\part1\exo24.py

3.2. Créer une matrice

```
# Créer la matrice [[1 ,2 ,3] ,[4 ,5 ,6] ,[7,8 ,9]]
L = []                      # Initialisation d'une liste vide
valeur = 1                  # Initialisation de la premiere valeur
for i in range(3):          # Balayage des sous listes
SL = []                     # Creation d'une sous liste vide
for j in range(3):          # Balayage des elements de la sous liste
SL.append(valeur)           # Ajout d'un elemet a droite de la sous liste
valeur = valeur + 1         # Incrementation de la valeur a ajouter
L.append(SL)                # Ajout de la sous liste a la liste
```

.\scripts\part1\creermatrice.py

```
*** Console de processus distant Réinitialisée ***
>>> L
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>>
```

✍ Exercice 25 : Créer une fonction matriceAlea( ) qui renvoie une matrice à n lignes et m colonnes d'entiers aléatoires entre 0 et 100

```
>>> matriceAlea(4 ,3)       # Lignes: n=4 et Colonnes: m=3

[[55 ,41 ,12] ,[5 ,87 ,14] ,[3 ,24 ,11] ,[56 ,34 ,25]]
```

.\scripts\part1\exo25.py

4. Copie de liste

La copie de liste en Python est un exercice délicat. En effet, si l'on crée une première liste liste1 puis que l'on écrit

```
>>> liste2 = liste1
```

Le contenu de liste1 ne sera pas recopié dans liste2 , les deux objets ne feront qu'un. Cela signifie notamment que si l'on modifie l'une des deux listes, la modification sera répercutée sur l'autre liste.

MÉTHODE 1 : Copier une liste

./scripts/part1/methode1.py

```
1 >>> import copy
2 >>> x = [[1,2],3]
3 >>> y = copy.deepcopy(x)
```

PARTIE 2 : Quelques algorithmes

1. Recherche dans une liste

Pour toute la suite du cours, nous utiliserons une liste L :

```
L = [-2,4,0,7,-3]
```

1.1. Recherche d'un élément dans une liste

```
L = [-2,4,0,7,-3]

# Recherche d'un élément dans une liste par itération
def chercheVal1(L,valeur):
    for i in range(len(L)):
        if valeur == L[i]:
        return True
    return False

assert chercheVal1(L,4) == True
assert chercheVal1(L,5) == False
```

.\scripts\part2\chercheVal1.py

```
L = [-2,4,0,7,-3]

# Recherche du élément dans une liste par itération sur les éléments
def chercheVal2(L,valeur):
    for element in L:
        if valeur == element:
            return True
    return False

assert chercheVal2(L,4) == True
assert chercheVal2(L,5) == False
```

.\scripts\part2\chercheVal2.py

✍ Exercice 1 : En utilisant la méthode d'itération sur les indices, créer une fonction `chercheValMatrice1(M,valeur)` indiquant si un élément est dans une matrice M.

.\scripts\part2\exo1.py

✍ Exercice 2 : En utilisant la méthode d'itération sur les éléments, créer une fonction `chercheValMatrice2(M,valeur)` indiquant si un élément est dans une matrice M.

```
M=[[1 ,2 ,3] ,[4 ,5 ,6] ,[7 ,8,9]]

assert chercheValMatrice1(M ,2) == True
assert chercheValMatrice1(M ,10) == False
```

.\scripts\part2\exo2.py

1.2. Recherche d'un extremum dans une liste

```
L = [-2,4,0,7,-3]

# Recherche du minimum d'une liste par itération
def chercheMin1(L):
        min = L[0]
        for i in range(1,len(L)):
                if min > L[i]:
                        min = L[i]
        return min

print(chercheMin1(L))
```

.\scripts\part2\minListe1.py

```
L = [-2,4,0,7,-3]

# Recherche du minimum d'une liste par itération sur les éléments
def chercheMin2(L):
        min = L[0]
        for element in L:
                if min > element:
                        min = element
        return min

print(chercheMin2(L))
```

.\scripts\part2\minListe2.py

✍ Exercice 3 : De la même manière programmer deux fonctions `chercheMax1()` et `chercheMax2()` qui recherchent les maximums des listes de deux manières différentes.

.*scripts\part2\exo3.py*

✍ Exercice 4 : En utilisant la méthode d'itération sur les indices, créer une fonction `chercheMaxMatrice1(M)` retournant le maximum d'une matrice.

.*scripts\part2\exo4.py*

✍ Exercice 5 : En utilisant la méthode d'itération sur les éléments, créer une fonction `chercheMaxMatrice2(M)` retournant le maximum d'une matrice.

.*scripts\part2\exo5.py*

```
M=[[1 ,2 ,3] ,[4 ,5 ,6] ,[7 ,8,9]]

assert chercheValMatrice1(M ,2) == True
assert chercheValMatrice1(M ,10) == False
```

2. Calclul de sommes

```
L = [-2,4,0,7,-3]

# Calcul de la somme des valeurs d'une liste par itération
def calculSomme1(L):
somme = 0
        for i in range(len(L)):
                somme = somme + L[i]
        return somme

assert calculSomme1(L) == 6
```

.\scripts\part2\sommeVal1.py

```
L = [-2,4,0,7,-3]

# Calcul de la somme des valeurs d'une liste par itération sur les éléments
def calculSomme2(L):
        somme = 0
        for element in L:
                somme = somme + element
        return somme

assert calculSomme2(L) == 6
```

.\scripts\part2\sommeVal2.py

✍ Exercice 6 : En utilisant la méthode d'itération sur les indices, créer une fonction sommeValMatrice1(M) retournant la somme des éléments de cette matrice.

\scripts\part2\exo6.py

✍ Exercice 7 : En utilisant la méthode d'itération sur les éléments, créer une fonction sommeValMatrice2(M) retournant la somme des éléments de cette matrice.

\scripts\part2\exo7.py

```
M=[[1 ,2 ,3] ,[4 ,5 ,6] ,[7 ,8,9]]

assert sommeValMatrice1(M) == 45
```

✍ Exercice 8 : En utilisant la méthode de votre choix, créer une fonction produitVal(L) retournant le produit des éléments d'une liste.

\scripts\part2\exo8.py

```
L = [2,4,6,7,3]

assert produitVal(L) == 1008
```

3. Calcul de moyenne

```
L = [2,4,6,7,3]

# Calcul de la moyenne des valeurs d'une liste par itération
def calculMoyenne1(L):
        somme = 0
        for i in range(len(L)):
                somme = somme + L[i]
        return somme/len(L)

assert calculMoyenne1(L) == 4.4
```

✍ Exercice 9 : En utilisant la méthode de votre choix, créer une fonction calculmoyenneMatrice(M) retournant la moyenne des éléments d'une matrice M.

\scripts\part2\exo9.py

```
M=[[1 ,2 ,3] ,[4 ,5 ,6] ,[7 ,8,9]]

assert calculmoyenneMatrice(M) == 5
```