Réflexe :
Importer la librairie nécessaire au calcul scientifique.

Entrée[2]:
```
1 import numpy as np
```

TP Python 13 : Introduction aux matrices

Correction

Partie I - Notion d'*array*

Nous avons vu au premier semestre que les **listes** en Python sont de très bons outils pour stocker de l'information. Jusqu'ici, nous les avons surtout utilisées pour stocker des nombres de même types (*float* ou *int*. Pourtant, il ne faut pas oublier qu'une liste peut **contenir des objets de nature différentes**! Les listes sont donc un outil puissant et **versatile** en Python pour stocker des données *hétérogènes*.

Toutefois, nous sommes parfois amenées à ne traiter que des données du même type (des dates d'anniversaire, des mots clefs rentrées dans un moteur de recherche, des températures...). Dans ce cas, certains outils Python peuvent se révéler plus adéquats en terme de **gain de temps et d'économie d'espace mémoire**. C'est le cas par exemple lorsque l'on souhaite stocker des données *homogènes*. Pour ce faire, nous allons utiliser un objet Python appelé *array*.

Le terme *array* signifique "tableau de données". En programmation, un *array* est donc une structure destinée à stocker des données. On peut se représenter un *array* comme une grille, dont chaque cellule contient un élément de donnée, chaque élément étant de même **nature** : un *float*, un *int*, une chaîne de caractère... Une grille à n lignes et p colonnes ($n, p \in \mathbb{N}^*$) ne contenant que des réels n'étant rien d'autre qu'une matrice, on dispose donc d'un outil Python pour le calcul matriciel!

I.1) Propriétés des *arrays*

Voici quelques propriétés fondamentales à bien connaître pour distinguer listes et *arrays*:

Array/Matrice	**Liste**
Tous les éléments sont obligatoirement du même **type.**	Les éléments peuvent être de type différents. *Exemple:* `L = [1,"fleurs",3.56,"motdepasse"]`

Une fois créée, une matrice (un *array*) ne peut changer de taille.	Une liste peut toujours changer de taille, avec la commande `append` ou la commande `remove` .
Un *array* peut avoir n lignes et p colonnes, avec $n, p \in \mathbb{N}^*$.	Une liste consiste en une unique ligne de données.

I.2) Définition d'une matrice

Commençons par se familiariser avec la défintion d'une matrice. En Python, on fixe une matrice ligne à ligne.

Par exemple, la commande

```
a = np.array([1,2,3,4,5])
print(a)
```

permet de définir le vecteur ligne $a = (1, 2, 3, 4, 5) \in \mathbb{R}^3$, puis l'affiche.
La commande

```
B = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(B)
```

permet de définir la matrice $B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \in \mathcal{M}_3(\mathbb{R})$, puis l'affiche. **Attention, il y a double crochets!**

Exercice 1

1. Définir le vecteur $x = (1, 0, 1, 0, 1, 1) \in \mathbb{R}^6$ à l'aide de la commande `np.array` .
2. Pouvez-vous **ajoutez** la coordonnée -1 au vecteur x afin de le transformer en vecteur $y = (1, 0, 1, 0, 1, 1, -1)$ avec la commande `x.append(-1)` ?
3. Définir la matrice $A = \begin{pmatrix} 5 & 2 & 6 & 4 \\ 0 & 1 & 0 & 1 \\ 11 & 22 & 33 & 44 \end{pmatrix}$ à l'aide de la commande `np.array` .

Entrée[16]:
```
x = np.array([1,0,1,0,1,1])
print(x)
A = np.array([[5,2,6,4],[0,1,0,1],[11,22,33,44]])
print(A)
```

```
[1 0 1 0 1 1]
[[ 5  2  6  4]
 [ 0  1  0  1]
 [11 22 33 44]]
```

On dispose de quelques commandes pour définir des matrices un peu spéciales :

1. `np.ones(n)` crée une matrice ligne de longueur n remplie 1.
2. `np.ones((n,p))` crée une matrice de taille $n \times p$ remplie de 1.

3. `np.zeros(n)` crée une matrice ligne de longueur n remplie 0.
4. `np.zeros((n,p))` crée une matrice de taille $n \times p$ remplie de 0.
5. `np.eye(m,n)` crée la matrice de taille $n \times p$ ne contenant que des 0 et dont la "diagonale" (on parle de diagonale principale) ne contient que des 1.
   Attention! La commande `np.eye` fait figure d'exception : la taille de la matrice n'y est pas indiquée entre parenthèses, contrairement aux commandes précédentes.

Exercice 2

1. Définir la matrice $A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$ et l'afficher.

2. Définir la matrice $B = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$ et l'afficher.

3. Définir la matrice $C = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ et l'afficher.

4. Définir la matrice $D = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$ et l'afficher.

5. Définir la matrice $E = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$ et l'afficher.

6. Définir la matrice $F = I_8$ et l'afficher.

Entrée[11]:

```
A = np.ones((3,4))
print("A=",A,"\n")
B = np.ones(4)
print("B=",B,"\n")
C = np.zeros(3)
print("C=",C,"\n")
D = np.zeros((5,2))
print("D=",D,"\n")
E = np.eye(5,3)
print("E=",E,"\n")
F = np.eye(8)
print("F=",F,"\n")

```

```
A= [[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]

B= [1. 1. 1. 1.]

C= [0. 0. 0.]

D= [[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]

E= [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]
 [0. 0. 0.]
 [0. 0. 0.]]

F= [[1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1.]]
```

Partie II - Opérations sur les matrices

II.1) Obtenir des informations sur les coefficients

On dispose aussi de plusieurs commandes nous permettant d'obtenir des informations intéressantes sur les matrices. Pour une matrice M fixée :

1. `transpose(M)` renvoie la matrice transposée M^t.
2. `np.sum(M)` renvoie la somme des coefficients de M.
3. `np.min(M)` renvoie le coefficient minimal de M.
4. `np.max(M)` renvoie le coefficient maximal de M.
5. `np.mean(M)` renvoie la moyenne des coefficients de M.
6. `np.cumsum(M)` renvoie un vecteur ligne dont chaque coefficient vaut la somme cumulative des coefficients précédents.

7. `np.median(M)` renvoie la médiane des coefficients de M.
8. `np.var(M)` renvoie la variance des coefficients de M.

Exercice 3

Tester chacune de ces commandes sur les matrices suivantes, après les avoir définies :

$$A = \begin{pmatrix} 5 & 2 & 6 & 4 \\ 0 & 1 & 0 & 1 \\ 11 & 22 & 33 & 44 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad C = \begin{pmatrix} \cos\left(\frac{pi}{6}\right) & -\sin\left(\frac{\pi}{6}\right) \\ \sin\left(\frac{\pi}{6}\right) & \cos\left(\frac{\pi}{6}\right) \end{pmatrix}.$$

Entrée[24]:

```
A = np.array([[5,2,6,4],[0,1,0,1],[11,22,33,44]])
B = np.array([[0,0,1],[0,1,0],[1,0,0]])
C = np.array([[np.cos(np.pi/6), -np.sin(np.pi/6)],[np.sin(np.pi/6), np.cos
# On print pour vérifier :
print(A, '\n')
#print(B)
#print(C)

# On teste les transposées
print('Affichons les transposées :')
print(np.transpose(A), '\n')
#print(np.transpose(B))
#print(np.transpose(C))

#etc etc
print(np.cumsum(B))
print(np.cumsum(C),'\n')

print('Le min de A est',np.min(A),'et le max de A est',np.max(A))
```

```
[[ 5  2  6  4]
 [ 0  1  0  1]
 [11 22 33 44]]

Affichons les transposées :
[[ 5  0 11]
 [ 2  1 22]
 [ 6  0 33]
 [ 4  1 44]]

[0 0 1 1 2 2 3 3 3]
[0.8660254  0.3660254  0.8660254  1.73205081]

Le min de A est 0 et le max de A est 44
```

II.2) Obtenir la taille

La commande `np.shape(M)` renvoie la taille de l'*array* M : en premier, le nombre de ligne, et en second, le nombre de colonne. Ces deux informations sont elles-même stockées dans une liste à deux éléments.

Exercice 4
Avez-vous bien compris ce que faisait la commande `np.shape(M)` ?

1. Compléter la fonction `affiche_taille` qui, étant donnée une matrice M, **affiche** la nombre de lignes de la matrice M puis **affiche** le nombre de colonnes de la matrice M.
2. Tester cette fonction sur une matrice de votre choix. **Remarque :** il va de soit que pour vérifier l'efficacité de la fonction, il vaut mieux tester sur une matrice de taille $n \times p$ avec $n \neq p$...

Entrée[27]:
```
def affiche_taille(M):
    taille = np.shape(M)
    print("La matrice a ", taille[0] ,"lignes \n")
    print("La matrice a ",taille[1],"colonnes \n")

    return

affiche_taille(np.array([[1,2],[5,6],[9,88]]))
```

```
La matrice M a  3 lignes

La matrice a  2 colonnes
```

→ Remarque ←

II.2) Produit matriciel

Enfin, on peut effectuer simplement un produit matriciel à l'aide de la commande `np.dot(A,B)` où A et B sont deux matrices compatibles pour le produit $A \times B$.

Exercice 5

1. Déclarez les matrices I_3, $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 3 & 2 & 1 \end{pmatrix}$ et $B = \begin{pmatrix} -1 & 2 \\ 3 & -4 \\ 0 & 0 \end{pmatrix}$.
2. Effectuer le produit $I_3 \times A$ et $A \times I_3$.
3. Effectuer le produit $A \times B$.
4. Effectuer le produit $B \times A$: que renvoie le message d'erreur?

Entrée[35]:

```
I = np.eye(3)
A = np.array([[1,2,3],[4,5,6],[3,2,1]])
B = np.array([[-1,2],[3,-4],[0,0]])

M = np.dot(I,A)
N = np.dot(A,I)
K = np.dot(A,B)
print(M)
print(N)
print(K)
L = np.dot(B,A)
```

```
[[1. 2. 3.]
 [4. 5. 6.]
 [3. 2. 1.]]
[[1. 2. 3.]
 [4. 5. 6.]
 [3. 2. 1.]]
[[  5  -6]
 [ 11 -12]
 [  3  -2]]

Traceback (most recent call last):
  File "<basthon-input-35-f3236abf41fc>", line 11, in <module>
    L = np.dot(B,A)
        ^^^^^^^^^^^
  File "<__array_function__ internals>", line 200, in dot
ValueError: shapes (3,2) and (3,3) not aligned: 2 (dim 1) != 3 (dim 0)
```

Partie III - *Slicing*

En calcul matriciel, on s'intéresse souvent à un coefficient en particulier (*par exemple un pivot*), ou à une ligne/colonne donnée d'une matrice M.

Attention, *arrays start at zero*!

- Pour obtenir un seul coefficient, par exemple celui en 3ème ligne et 2ème colonne, on dispose de deux manières de faire : `M[2][1]` ou `M[2,1]` .
- On peut aussi récupérer une ligne entière. Par exemple, la commande `M[k,:]` **désigne** la $k+1$-ème ligne d'une matrice M donnée.
- De même, on peut récupérer une colonne entière. Par exemple, la commande `M[:,k]` **désigne** la $k+1$-ème colonne d'une matrice M donnée.

Exercice 6

Compléter la fonction `table_multiplication` suivante qui prends en argument un entier $n \in \mathbb{N}^*$, construit la matrice de taille $10 \times n$ suivante :

$$M_n = \begin{pmatrix} 1 & 2 & 3 & \dots & n \\ 2 & 4 & 6 & \dots & 2n \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 10 & 20 & 30 & \dots & 10n \end{pmatrix}.$$

Entrée[15]:

```
def table_multiplication(n):

    M = np...((10,n))

    for i in range(...):
        for k in range(...):
            M[i,k] = ...*...
    print(M)
    return

table_multiplication(3)
```

```
  File "<basthon-input-15-afacf07e8b8e>", line 3
    M = np...((10,n))
          ^^^
SyntaxError: invalid syntax
```

Entrée[16]:

```
def table_multiplication(n):

    M = np.zeros((10,n))

    for i in range(10):
        for k in range(n):
            M[i,k] = (i+1)*(k+1)
    print(M)
    return

table_multiplication(3)
```

```
[[ 1.  2.  3.]
 [ 2.  4.  6.]
 [ 3.  6.  9.]
 [ 4.  8. 12.]
 [ 5. 10. 15.]
 [ 6. 12. 18.]
 [ 7. 14. 21.]
 [ 8. 16. 24.]
 [ 9. 18. 27.]
 [10. 20. 30.]]
```

Exercice 7

1. Écrire une fonction `ligne3_colonne2` qui prend en argument une matrice M et qui, par dans l'ordre :
   A. vérifie la taille de la matrice : si M a moins de trois lignes ou moins de deux colonnes, la fonction s'arrête,
   B. affiche la troisième ligne de la matrice M,
   C. renvoie la deuxième colonne de la matrice M.
2. Tester la fonction sur les matrice $A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$, $B = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ et $C = \begin{pmatrix} -1 & 0 & 0 & 1 \\ -2 & 0 & 1 & 4 \\ 1 & 1 & -1 & 2 \\ -3 & -2 & -6 & -1 \end{pmatrix}$.

Entrée[14]:

```
def ligne3_colonne2(M):
    if np.shape(M)[0]<3 or np.shape(M)[1]<2 :
        print("erreur taille")
        return

    print("La troisième ligne de la matrice est :",M[2,:])
    return M[:,1]

ligne3_colonne2(np.array([[1,2,3,4],[5,6,7,8]]))
ligne3_colonne2(np.eye(2))
ligne3_colonne2(np.array([[-1,0,0,1],[-2,0,1,4],[1,1,-1,2],[-3,-2,-6,-1]]))

```

```
erreur taille
erreur taille
La troisième ligne de la matrice est : [ 1  1 -1  2]
```

Sortie[14]:

```
array([ 0,  0,  1, -2])
```

Exercice 8

Écrire une fonction `strange_mult` qui prend en argument deux matrices A et B de même taille et renvoie la matrice C dont le coefficient en position (i, j) est le produit des coefficients en position (i, j) de A et B.

Remarque : la taille $n \times p$ n'est pas un argument de la fonction.

Entrée[22]:

```
def strange_mult(A,B):
    if np.shape(A) != np.shape(B):
        print("erreur taille")
        return
    n = np.shape(A)[0]
    p = np.shape(A)[1]

    C = np.zeros((n,p))
    for i in range(n):
        for j in range(p):
            C[i,j] = A[i,j]*B[i,j]
    return C

A = 2*np.ones((3,3))
B = np.array([[1,2,3],[4,5,6],[7,8,9]])
strange_mult(A,B)
```

Sortie[22]:

```
array([[ 2.,  4.,  6.],
       [ 8., 10., 12.],
       [14., 16., 18.]])
```