

	<i>Université de Corse - Pasquale PAOLI</i>	
	<b>Diplôme : Licence SPI 3<sup>ème</sup> année</b>	<b>2023-2024</b>
	<b>UE : Ateliers de programmation</b>	
	<b>Atelier 5 Numpy : Matrices, graphes et récursivité</b> Enseignants : Paul-Antoine BISGAMBIGLIA, Marie-Laure NIVET, Evelyne VITTORI	

## Partie 1 – Récursivité

Question 1 : Calcul de la somme d'une liste de manière récursive

Écrivez une fonction récursive *somme\_recursive* qui prend en entrée une liste de nombres et retourne la somme de tous les éléments de la liste en utilisant la récursivité.

```
# Ma fct somme

def somme_recursive(liste:list)->integer:

# Test de la fonction
liste1 = [1, 2, 3, 4, 5]
resultat1 = somme_recursive(liste1)
print("La somme de la liste est :", resultat1)

liste2 = []
resultat2 = somme_recursive(liste1)
print("La somme de la liste est :", resultat2)
```

Question 2 : Calcul de la factorielle d'un nombre de manière récursive

Écrivez une fonction récursive *factorielle\_recursive* qui prend en entrée un nombre entier et retourne sa factorielle en utilisant la récursivité. La factorielle d'un nombre n est le produit de tous les entiers positifs inférieurs ou égaux à n.

```
# Ma fct factorielle

def factorielle_recursive(nombre:integer): ->integer:

# Test de la fonction
nombre = 5
resultat = factorielle_recursive(nombre)
print("Le factoriel de", nombre, "est :", resultat)
```

Question 3 : Définir une fonction récursive *longueur* qui prend en paramètre une liste L et retourne un entier représentant le nombre d'éléments de la liste L.

Question 4 : Définir une fonction récursive *minimum* qui prend en paramètre une liste d'entiers L et retourne un entier représentant l'élément minimum de la liste L.

Question 5 : Définir une fonction récursive *listPairs* qui prend en paramètre une liste L et retourne une liste contenant les éléments pairs présents dans la liste L.

Question 6 : Définir une fonction récursive *concat\_list* qui prend en paramètre une liste LL composée éventuellement d'une ou plusieurs sous-listes et retourne une liste plate constituée de tous les éléments simples de la liste LL et de ses sous-listes.

```
# concat

def concat_list(LL: list) -> list

# Test de la fonction
concat_list([[0,1],[2,3],[4],[6,7]])
# [0,1,2,3,4,6,7]
```

Question 7 : Ecrire une fonction `include` qui prend en paramètre deux listes et vérifie que les éléments de la première liste sont tous présents dans la deuxième. Attention ils doivent être présents dans l'ordre mais non nécessairement consécutivement.

Exemple :

```
>include([], [4,3])      >include([], [])      >include([1,2,6], [1,2,3,5,6])
True                    True                    True
>include([1, 2,3], [1,2]) >include([1..10], [1..10])    >include([1,2], [])
False                   True                    False
```

## Partie 2 – Matrices

### Exercice 1: Opérations de base

Dans cet exercice, vous représenterez les matrices par un array numpy à deux dimensions. NumPy est une librairie python pour manipuler des tableaux de nombres. Il est utilisé pour manipuler des matrices.

**Préalable, installation et utilisation** de numpy :

```
pip install numpy
```

```
import numpy

arr = numpy.array([1, 2, 3, 4, 5])

print(arr)

# ou
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```

Lire la doc : <https://www.w3schools.com/python/numpy/default.asp>

**Test de code :**

```
import numpy as np

# exemple 1

arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in arr:
    print(x)

# exemple 2

arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in arr:
    for y in x:
        print(y)
```

La fonction *searchsorted()* permet de chercher un élément dans un tableau et de renvoyer sa place (indice), tester le code ci-dessous :

```
#Exemple d'usage de la fct where

import numpy as np

arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 7)

print(x)
```

### Question 1 : Proposez votre version de la fonction *searchsorted*

```
# Ma fct searchsorted

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 14])

def my_searchsorted(table : object, indice : integer )-> integer:
```

La fonction *where* permet de chercher des éléments dans un tableau et de renvoyer leurs places (indices) dans une liste, tester le code ci-dessous :

```
#Exemple d'usage de la fct where

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 4, 4])

x = np.where(arr==4)

print(x)#[3,5,6]
```

### Question 2 : Proposez votre version de la fonction *where*

```
# Ma fct where

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 4, 4])

def my_where(table : object, valeur : integer )-> list:

x = my_where(arr, 4)

print(x)#[3,5,6]
```

## Addition de matrice

Voici un exemple d'addition de matrice de même dimension (ici la dimension est 2)

$$R = A + B = \begin{pmatrix} 3 & 1 \\ 6 & 4 \end{pmatrix} + \begin{pmatrix} 1 & 8 \\ 4 & 2 \end{pmatrix} = \begin{pmatrix} 4 & 9 \\ 10 & 6 \end{pmatrix}$$

Avec NumPy c'est assez simple, vous pouvez définir A et B et faire simplement A + B (c'est faisable car elles ont la même dimension)

```
#Exemple d'addition de matrice
import numpy as np

A = np.array([[3,1],[6,4]])
B = np.array([[1,8],[4,2]])

A.shape == B.shape
# true (vérification de la dimension)

R = A + B

print(R)
```

**Question 3: Proposez une fonction qui prend en argument 2 tableaux, qui vérifie leur dimension et qui réalise l'addition.**

```
# Ma fct add

import numpy as np

def my_add(tableA : object, tableB : object)-> object:
```

## Autres exercices autour des matrices

### 1. Initialisation et affichage

- Créez un tableau numpy représentant une matrice M de dimensions 3x3 avec des valeurs allant de 1 à 9.
- Affichez cette matrice.

### 2. Opérations élémentaires

- Ajoutez 10 à chaque élément de la matrice M et affichez le résultat.
- Multipliez chaque élément de la matrice M par 2 et affichez le résultat.

### 3. Slicing et indexation

- Affichez la deuxième ligne de la matrice M.
- Affichez la troisième colonne de la matrice M.
- Extrayez une sous-matrice 2x2 du coin supérieur gauche de la matrice M et affichez-la.

## Exercice 2 : Manipulations avancées

A partir d'ici vous pouvez utiliser toutes les fonctions python disponibles dans numpy.

### 1. Création de matrices

- Créez une matrice A de dimensions 4x4 avec des valeurs aléatoires entre 0 et 10.
- Créez une matrice identité I de dimensions 4x4.

### 2. Fonctions à définir

- Écrivez une fonction `matrice_trace(matrice)` qui prend en entrée une matrice carrée et retourne sa trace.

Note : La trace d'une matrice est la somme des valeurs de la diagonale de la matrice. Cf. Image ci-dessous (extraite de <https://www.geeksforgeeks.org/trace-of-a-matrix/>).

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$\text{tr}(A) = a_{11} + a_{22} + a_{33}$$

- Écrivez une fonction `est_symetrique(matrice)` qui détermine si une matrice est symétrique ou non. Elle retournera `True` si la matrice est symétrique et `False` sinon.
  - Écrivez une fonction `produit_diagonal(matrice)` qui retourne le produit des éléments de la diagonale principale d'une matrice carrée.
3. **Application des fonctions**
- Calculez la trace de la matrice A en utilisant votre fonction `matrice_trace`.
  - Déterminez si la matrice  $(A + A.T)/2$  est symétrique en utilisant la fonction `est_symetrique`.
  - Calculez le produit des éléments de la diagonale de la matrice I en utilisant la fonction `produit_diagonal`.
4. **Manipulation supplémentaire**
- Inversez la matrice A et multipliez-la par A. Vérifiez que vous obtenez une matrice proche de la matrice identité I (en raison des erreurs de précision, elle ne sera pas exactement I).

*NB= une matrice identité est une matrice carrée (c'est-à-dire qu'elle a le même nombre de lignes et de colonnes) qui possède des "1" sur sa diagonale principale (allant du coin supérieur gauche au coin inférieur droit) et des "0" partout ailleurs.*

## Partie 2 – Application aux graphes

On considère un graphe orienté  $G(S,A)$  représenté deux listes S (liste des sommets) et A (listes des arcs)

1. Définissez une procédure **`matriceAdjacence(S,A)`** qui admet en paramètres une liste S de sommets et une liste A d'arcs (liste de tuples (i,j) avec  $i,j \in S$ ) et retourne la matrice d'adjacence associée (type array à 2 dimensions de numpy).
2. Définissez une procédure **`matriceAdjacencePond(S,A)`** qui admet en paramètres une liste S de sommets et une liste A d'arcs pondérés (liste de triplets (i,j,poids) avec  $i,j \in S$ ) et retourne la matrice d'adjacence associée (type array à 2 dimension de numpy).
3. Définissez une fonction **`lireMatriceFichier(nomfichier)`** qui renvoie une matrice carrée (type array de numpy) contenue dans le fichier dont le nom est passé en paramètre.

1	0	0	0	1	0	0	0
1	1	1	0	1	1	1	0
0	0	1	0	0	0	1	0
0	0	0	1	0	0	0	1
0	0	0	0	1	0	0	0
0	0	0	0	1	1	1	0
0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1

exemple de matrice :

Pour tester votre fonction, récupérez sur l'ENT les fichiers graph0.txt, graph1.txt, graph2.txt, graph3.txt et graph4.txt

4. Définissez la fonction **tousLesSommets(mat)** qui retourne une liste contenant tous les indices des sommets du graphe G défini par la matrice d'adjacence **mat**.
5. Définissez la fonction **listeArcs(mat)** qui retourne la liste des arcs (i,j) du graphe G défini par la matrice d'adjacence **mat**.
6. Définissez la fonction **matriceIncidence(mat)** qui retourne la matrice d'incidence associée au graphe défini par la matrice d'adjacence **mat**. Le graphe G est supposé sans boucle.

Exemple :

```
Liste des sommets = [0, 1, 2, 3, 4]
Liste des arcs : [(0, 1), (0, 2), (1, 2), (1, 4), (2, 3), (3, 4), (4, 2)]

Matrice d'adjacence :
[[ 0.  1.  1.  0.  0.]
 [ 0.  0.  1.  0.  1.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.]
 [ 0.  0.  1.  0.  0.]]
```

7. Définissez la fonction **est\_voisin** qui admet en paramètres une matrice d'adjacence et deux sommets S et V et renvoie un booléen vrai si les deux sommets sont voisins et faux dans le cas contraire.

## ANNEXE : Module numpy

```
>>> from numpy import * # chargement du module numpy
>>> M = array([[1, 2, 3], [4, 5, 6]])
>>> M
array([[1, 2, 3], [4, 5, 6]])
>>> M[0][1] # terme d'indice (0, 1)
2
>>> M[0,1] # terme d'indice (0, 1)
2
>>> size(M) # nombre de termes
6
>>> shape(M)
(2, 3)
>>> M.shape
(2, 3)
>>> M.shape[0]
2
>>> N = array([[0, 0, 0], [1, 2, 3]])
>>> M+N # somme terme à terme
array([[1, 2, 3],
       [5, 7, 9]])
>>> M*N # produit terme à terme
array([[ 0, 0, 0],
       [ 4, 10, 18]])
>>> M**2 # carré terme à terme
array([[ 1, 4, 9], [16, 25, 36]])
>>> P = M.T # transposée
```

```
>>> P
```

```
array([[1, 4], [2, 5], [3, 6]])
>>> Q = dot(M, P) # produit matriciel
>>> Q
array([[14, 32], [32, 77]])
>>> MZ=zeros([4,4]) #crée un array 4x4 rempli avec des 0
>>> MU=ones([4,4]) #crée un array 4x4 rempli avec des 1
>>> ME=empty([4,4]) #crée un array 4x4 vide
#il est possible de préciser le type des éléments à la création de la matrice
# ex:
>>> MU=empty([4,4],int)
>>> MZ=zeros([4,4],int)
```