Informatique Pour Tous

TD 14 – Matrices entières

Dans ce sujet, on n'aborde que des matrices « entières », c'est-à-dire des matrices de $M_{n,p}(\mathbb{Z})$ (à coefficients entiers). Ainsi, il sera possible de faire des tests d'égalité entre les éléments.

Rappel : En *Numpy*, une matrice `M` créée avec `np.zeros([n, p])` est par défaut en type *float*. Pour forcer son passage en type *int*, on utilise la commande `Mint = M.astype(int)`. Cette commande ne modifie pas `M` (portée locale).

A / Matrice symétrique

Q1 – Écrire une fonction `est_sym(M)` qui prend en argument une matrice **carrée** `M`, codée sous forme de **np.array**, et qui renvoie un booléen : `True` si la matrice est symétrique, `False` sinon.

Remarque : si on a testé que $M_{i,j} == M_{j,i}$, il est inutile de tester si $M_{j,i} == M_{i,j}$: on peut donc faire le balayage (index *i* et *j*) uniquement sur le triangle supérieur de la matrice, en comparant terme à terme avec le triangle inférieur.

Tests :
```
M_tst1 = np.array([[1,4,5],[4,2,6],[5,6,2]])
M_tst2 = np.array([[1,4,5],[4,2,6],[5,4,3]])
print(est_sym(M_tst1))     # doit afficher True
print(est_sym(M_tst2))     # doit afficher False
```

Remarque : une *assertion* est un test, généralement réalisé en tout début de fonction, pour vérifier que les arguments répondent bien à une hypothèse. En l'occurrence ici, que la matrice `M` est carrée. On écrit :

```
assert test_logique, message_en_cas_d_erreur
```

Si le test est `True`, on passe à la ligne suivante (pas d'erreur levée). Sinon, on affiche le message d'erreur et on quitte le script (donc *a fortiori* la fonction).

Q2 – Modifier la fonction précédente en ajoutant une assertion permettant de vérifier que la matrice `M` est carrée. Si ce n'est pas le cas, une erreur affichera *« La matrice n'est pas carrée ».*

Test :
```
M_tst3 = np.array([[1,4,5],[4,2,6],[5,4,3],[9,8,7]])
Tst = est_sym(M_tst3) # cause l'erreur AssertionError: La matrice n'est pas carrée
```

B / Transposée

B.1 / Portée globale (fonction procédurale)

Q3 – Écrire une fonction procédurale `transpose(M)` qui prend en argument une matrice **carrée** `M`, codée sous forme de liste de listes. Cette fonction ne renverra rien, mais transposera la matrice `M` avec une portée globale.

Test :
```
A = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]
print(transpose(A))    # doit afficher None car transpose ne renvoie rien
print(A)  # la transposition a quand même été effectuée à l'appel précédent,
      # affiche [[1, 5, 9, 13], [2, 6, 10, 14], [3, 7, 11, 15], [4, 8, 12, 16]]
```

B. 2 / Portée locale

La question précédente avait pour objectif de vous rappeler que les listes comme les `np.array`, lorsqu'ils sont mis en argument d'une fonction, ont par défaut une portée **globale**. Il conviendra donc, si vous devez les modifier, de faire une copie (si elle est nécessaire).

Q4 – Écrire une fonction `transpose_L(M)` qui prend en argument une matrice `M`, pas forcément carrée, codée sous forme de **liste de listes**. Cette fonction renverra la transposée de la matrice `M` (sous forme de liste de listes).

Test :
```
B = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
print(transpose_L(B))   # affiche[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
print(B)   # on doit retrouver [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

Q5 – ⌨ Écrire une fonction `transpose_np(M)` qui prend en argument une matrice `M`, pas forcément carrée, codée cette fois sous forme de **np.array**. Cette fonction renverra la transposée de la matrice `M` (sous forme de np.array).

Test :
```
B_np = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
print(transpose_L(B_np))   # affiche      [[ 1  5  9]
                                          [ 2  6 10]
                                          [ 3  7 11]
                                          [ 4  8 12]]

print(B_np)   # on doit retrouver   [[ 1  2  3  4]
                                    [ 5  6  7  8]
                                    [ 9 10 11 12]]
```

C / Retournement d'une matrice

Q6 – ⌨ Écrire une fonction `retourne(M)` qui prend en argument une matrice `M` (pas forcément carrée), codée sous forme de **np.array**, et qui renvoie une matrice de mêmes dimensions, correspondant à la rotation à 180° de `M`.

Test :
```
B_np = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
print(retourne(B_np))    # affiche [[12 11 10  9]
                                  [ 8  7  6  5]
                                  [ 4  3  2  1]]
```

Challenge : en vous souvenant que, pour une liste `L`, la commande `L[::-1]` inverse l'ordre de la liste, essayez de faire tenir la fonction `retourne(M)` en deux lignes seulement ! Une ligne pour le **`def`**, une ligne pour le **`return`**.

D / Permutation par blocs

Soit une matrice $M \in M_{n,p}(\mathbb{Z})$ (n lignes, p colonnes) pour laquelle n et p soient tous les deux pairs. On souhaite écrire une fonction `permut_blocs(M)` qui prenne en argument une matrice de ce type (implémentée sous forme de **np.array**, et la fonction lèvera une *assertion* si *n* ou *p* est impair) et qui fasse une permutation circulaire des 4 blocs, de 90° dans le sens trigonométrique :

$$\begin{bmatrix} \text{Bloc A} & \text{Bloc B} \\ \text{Bloc C} & \text{Bloc D} \end{bmatrix} \longrightarrow \begin{bmatrix} \text{Bloc B} & \text{Bloc D} \\ \text{Bloc A} & \text{Bloc C} \end{bmatrix}$$

Exemple : la matrice $\begin{bmatrix} 11 & 12 & 13 & 14 & 15 & 16 \\ 21 & 22 & 23 & 24 & 25 & 26 \\ 31 & 32 & 33 & 34 & 35 & 36 \\ 41 & 42 & 43 & 44 & 45 & 46 \end{bmatrix} \longrightarrow \begin{bmatrix} 14 & 15 & 16 & 34 & 35 & 36 \\ 24 & 25 & 26 & 44 & 45 & 46 \\ 11 & 12 & 13 & 31 & 32 & 33 \\ 21 & 22 & 23 & 41 & 42 & 43 \end{bmatrix}$

Q7 – ⌨ Copier-coller, et compléter le code à trou ci-dessous :

```
def permut_blocs(M):
    n, p = np.shape(M)
    assert ...    # test de parité de n et p, sinon assertion (erreur)

    M_s = np.zeros_like(M)    # même taille, même type que M
    M_s[..., ...] = M[..., ...]  # Emplacement_C <- Bloc_A
    M_s[..., ...] = M[..., ...]  # Emplacement_A <- Bloc_B
    M_s[..., ...] = M[..., ...]  # Emplacement_D <- Bloc_C
    M_s[..., ...] = M[..., ...]  # Emplacement_B <- Bloc_D
    return M_s
```

Test :
```
B_np = np.array([range(11,17),range(21,27), range(31,37), range(41,47)])
print(permut_blocs(Mtst))      # affiche [[14 15 16 34 35 36]
                                         [24 25 26 44 45 46]
                                         [11 12 13 31 32 33]
                                         [21 22 23 41 42 43]]
```