I Approche en python

Une matrice de taille n, p est représentée en python par une liste de n listes, chacune des sous-listes comportant p éléments.
Par exemple la matrice $\mathrm{A} = \begin{pmatrix} 1 & 0 & -2 \\ 0 & -1 & 2 \end{pmatrix}$ est représentée en python par

```
A = [[1, 0, -2], [0, -1, 2]]
```

Exercice 1
Prévoir la valeur de retour des instructions `len(A)`, `A[0]`, `A[1][2]`.
D'une manière plus générale, on considère une matrice de taille n, p encodée en python par `A`. Pour quelles valeurs de i et j peut-on écrire `A[i][j]` ? Comment obtenir les valeurs de n et p en python ?

Exercice 2
Compléter les fonctions `contenu_colonne` et `extrait_colonne`.
Avons-nous besoin de fonctions similaires pour les lignes ?

Rappel/astuce En python, on peut accéder à toute une "tranche" d'une liste (qu'on appelle aussi *slice*) par la syntaxe

```
L[i:j]
```

qui crée la liste des éléments d'indices $i, i+1, \ldots, j-1$ de `L` (les mêmes nombres que dans `range(i, j)`)
Ainsi, pour accéder à la matrice ligne correspondant à la ligne d'indice i de A, on peut utiliser `A[i:i + 1]`

Exercice 3
Pour une matrice $\mathrm{A} = (a_{i,j})_{(i,j)\in[\![1,n]\!]\times[\![1,p]\!]}$, on appelle norme de A le maximum des $|a_{i,j}|$.
Compléter la fonction `norme`.

II En numpy

Il est très facile de convertir une matrice `A` en tableau numpy :

```
B = np.array(A)
```

Les éléments de `B` sont maintenant des tableaux numpy (des "array").
les deux syntaxes suivantes donnent accès au même élément

```
B[0][1]
B[0, 1]
```

On préfère souvent la deuxième forme, plus proche des notations mathématiques, et plus polyvalentes.
Pour avoir accès à la taille d'une matrice en numpy, on utilise

```
B.shape  # pas de parenthèses, retourne un 2-uple
# ou alors
n, p = B.shape  # n lignes, p colonnes
```

Comparer dans la console :

```
A[0:1][1:3]
B[0:1, 1:3]
```

Ainsi, numpy permet d'utiliser des slices sur les deux indices en même temps !

Exercice 4
Compléter la fonction `extrait_colonne_np`, en utilisant un slice.
Le tableau retourné n'est pas de la bonne forme, on utilisera `np.reshape(A, (n, p))` qui transforme le tableau numpy `A` en matrice de taille n, p.
Notez le nombre de parenthèses, la fonction `reshape` ne prend que 2 arguments.

Exercice 5
Compléter la fonction `norme_np`. On pourra utiliser les versions numpy de `max` et `abs`.

III Produit matriciel

On s'intéresse maintenant à l'algorithme du produit de deux matrices carrées de taille n.
Avec les notations classiques, $\mathrm{A} = (a_{i,j})$, $\mathrm{B} = (b_{i,j})$ et $\mathrm{C} = (c_{i,j})$, si on a $\mathrm{C} = \mathrm{AB}$ alors pour $i, j \in [\![1, n]\!]$

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

Le but de cette partie est d'estimer la complexité (en ordre de grandeur) de cet algorithme et de tester en pratique comment cette complexité se traduit en terme de temps de calcul.
Une traduction python directe donne :

```
def produit(A, B):
    """
    Calcule et retourne la matrice produit AB.
    Les matrices A et B doivent être carrées de même taille.
```

```
    """
    n = len(A)  # = len(B) = len(A[0]) = len(B[0])
    C = [[0 for j in range(n)] for i in range(n)]
    # C est la matrice nulle de taille n
    for i in range(n):
        for j in range(n):
            s = 0  # calcule d'un coeff par somme
            for k in range(n):
                s = s + A[i][k]*B[k][j]
            C[i][j] = s
    return C
```

Exercice 6
Donner en fonction de n (notation de la fonction) le nombre d'additions et le nombre de multiplications effectuées par cette fonction.

Exercice 7
Nous allons estimer le temps moyen pour effectuer le produit, par cette méthode, de deux matrices carrées de taille n.
La méthode en python est la suivante :

```
t0 = time.time()
# ici on fait un calcul
temps_calcul = time.time() - t0
```

La fonction `time.time` retourne le nombre de secondes écoulées depuis le 01/01/1970.
Compléter la fonction `temps_multiplication`. On évitera de compter le temps de création des matrices aléatoires.

Exercice 8
Pour essayer d'évaluer l'évolution du temps de calcul, on mesure le temps d'exécution pour des matrices de taille 2, puis de taille 4, 8, 16... et on observe si le facteur multiplicatif sur les temps de calcul est compatible avec le résultat théorique.

Exercice 9
Evidemment, numpy permet d'effectuer directement le produit matriciel.

```
# A et B sont des tableaux numpy ''compatibles''
A.dot(B)  # calcule A*B
```

Reprendre la fonction précédente, mais en numpy. Pour la création de matrices aléatoires, on utilisera

```
npr.randint(-100, 100, size=(n, n))  # n est la taille désirée
```

`npr` est ici l'alias donné au module `numpy.random`

Exercice 10
Comparer les temps d'exécution pour différente tailles de matrices, entre notre implémentation et la fonction numpy.

Exercice 11
On cherche à estimer le temps mis par notre fonction pour calculer un produit de deux matrices de même taille.

1. Créer une liste `tailles` comprenant tous les nombres pairs de 2 à 30.
2. Créer une liste `temps` qui contient les temps d'exécutions pour chacune des tailles précédentes.
3. Trouver une constante a telle que le temps mis pour calculer le produit de deux matrices de taille n soit $a \times n^3$ (environ). Valider en affichant sur un même graphique la courbe expérimentale (temps en fonction de la taille) et théorique ($a \times n^3$ en fonction de n).
   Avec mon matériel, je trouve $a \approx 1.39 \times 10^{-7}$

On trouve en faisant la même expérience en numpy, que la complexité est du même ordre de grandeur (n^3), mais la constante a est beaucoup plus petite, d'un facteur 175 (toujours dans ma configuration de test).

IV Extraction de sous-matrice

On considère 3 tableaux numpy :
- `A` carré de taille n
- `B` et `C` pas même taille p, q avec $p, q \neq n$.

Alors, on peut utiliser la syntaxe suivante :

```
D = A[B, C]
```

`D` est un tableau de taille p, q dont les éléments sont des éléments de A. Plus précisément, l'élément d'indices i, j de `D` est l'élément de `A` dans la ligne `B[i, j]` et colonne `C[i, j]`