TP3 — Fonctions *built-in* et Surcharge d'opérateurs

Objectif : Le but de ce TP est de reprendre la classe du TP2 représentant des matrices de $\mathcal{M}_n(\mathbb{R})$, et de lui ajouter des surcharges d'opérateurs.

Rappels du TP2

Dans ce TP, on reprendra de la classe **`Matrice`** du TP2. On pourra par exemple partir du code suivant, qui implémente l'addition de deux matrices sous forme de liste de listes.

```python
class Matrice:

    # constructeur
    def __init__(self, liste_de_listes):
        self.M = liste_de_listes
        self.n = len(self.M)

    # surcharge de str(Matrice) pour le print
    def __str__(self):
        return str(self.M)

    # addition : renvoie une instance de la classe Matrice
    def addition(self, other):
        assert self.n == other.n, "Addition impossible ! les tailles sont inégales"
        somme = [[self.M[i][j] + other.M[i][j] for j in range(self.n)] for i in range(self.n)]
        return Matrice(somme)
```

On rappelle que l'instruction

```python
x = [[2 * i + j**2 for j in range(3)] for i in range(2)]
```

est équivalente à la double boucle

```python
x = []
for i in range(2):
    ligne = []
    for j in range(3):
        ligne.append(2 * i + j**2)
    x.append(ligne)
```

et crée la liste de listes `x` telle que

$$\forall i \in \{0,1\}, \forall j \in \{0,1,2\},\ \texttt{x[i][j] = 2 * i + j**2},$$

soit la liste de listes ci-dessous.

```python
x = [[0, 1, 4], [2, 3, 6]]
```

Surcharges et méthodes possibles pour la classe `Matrice`

On s'intéresse à la représentation de matrices $A = (a_{ij})_{(i,j)\in[\![1,n]\!]^2}$.

On pourra par exemple surcharger les opérateurs et ajouter les méthodes ci-dessous. Cette liste n'est pas exhaustive, d'autres surcharges sont possibles : faites parler votre créativité !

Le nombre d'étoiles représente la difficulté de l'implémentation.

1. Accesseurs et mutateurs

Ces fonctions permettent de surcharger l'opérateur `[]`.

On précise que cet opérateur prend en entrée un `tuple`, dont le premier élément représente `i` et le deuxième élément représente `j`. On pourra procéder de la manière suivante pour récupérer `i` et `j` :

```
class Matrice:

        # [...]

        def __getitem__(self, index):
                i, j = index  # index est un tuple
                # [...]
```

On pourra réutiliser les accesseurs et mutateurs du TP2 : fonctions `get_M_ij` et `set_M_ij`.

On pourra surcharger l'opérateur `[]` des façons suivantes :

* l'accès, avec `[]`, à un coefficient d'une instance de **`Matrice`**, opérateur **`__getitem__`** : cet opérateur renverra le coefficient voulu ;
* la modification, avec `[]`, d'un coefficient d'une instance de **`Matrice`**, opérateur **`__setitem__`** : cet opérateur ne renverra rien mais modifiera le coefficient voulu avec la valeur voulue.

2. Opérateurs arithmétiques

Tous les opérateurs ci-dessous devront renvoyer une instance de la classe **`Matrice`**.

* l'addition `+` entre deux instances de **`Matrice`**, opérateur **`__add__`**
* la soustraction `-` entre deux instances de **`Matrice`**, opérateur **`__sub__`**
* la multiplication par un scalaire `*` entre une instance de **`Matrice`** et un scalaire, opérateur **`__mul__`**
* la division par un scalaire non nul `/` entre une instance de **`Matrice`** et un scalaire non nul, opérateur **`__truediv__`**

** la multiplication à gauche par un scalaire `*` entre un scalaire et une instance de **`Matrice`**, opérateur **`__rmul__`**

*** la multiplication matricielle `@` entre deux instances de **`Matrice`**, opérateur **`__matmul__`** (on rappelle que $P = AB$ vérifie : $\forall (i,j) \in [\![1,n]\!]^2,\ p_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$)

*** l'exponentiation `**` entre une instance de **`Matrice`** et un entier, opérateur **`__pow__`**

3. Méthodes

- une méthode `trace` pour calculer la trace de la matrice (on rappelle que $\mathrm{Tr}\, A = \sum_{i=1}^{n} a_{ii}$)
- une méthode `transpose` pour calculer la transposée de la matrice (on rappelle que ${}^tA = (a_{ji})_{(i,j)\in[\![1,n]\!]^2}$)

** une méthode `norm_1` pour calculer la norme 1 de la matrice (cette norme est définie en prenant le maximum sur les colonnes de la somme de chaque ligne des valeurs absolues des coefficients de la matrice : $\|A\|_1 = \max\limits_{1\leq j\leq n} \sum\limits_{i=1}^{n} |a_{ij}|$)

*** une méthode `norm_Frobenius` pour calculer la norme de Frobenius de la matrice (cette norme est définie par $\|A\|_F = \sqrt{\mathrm{Tr}({}^tAA)}$)

4. Comparaisons

Tous les opérateurs ci-dessous devront renvoyer un `bool` : **`True`** ou **`False`**.

- l'égalité `==` entre deux instances de **`Matrice`**, opérateur **`__eq__`**
- la non-égalité `!=` entre deux instances de **`Matrice`**, opérateur **`__ne__`**

** la comparaison `<` entre deux instances de **`Matrice`**, opérateur **`__lt__`** :
on définit $A < B$ comme $\|A\|_1 < \|B\|_1$

** la comparaison `<=` entre deux instances de **`Matrice`**, opérateur **`__le__`** :
on définit $A \leq B$ comme $\|A\|_1 \leq \|B\|_1$
(les fonctions *built-in* `min` et `max` dérivent de cet opérateur)

** les comparaisons `>` et `>=` entre deux instances de **`Matrice`**, opérateurs **`__gt__`** et **`__ge__`** : on procédera de la même façon que les deux opérateurs ci-dessus

*** on peut aussi reprendre les quatre opérateurs ci-dessus en se basant sur la norme $\|A\|_F$ plutôt que sur la norme $\|A\|_1$