TP2 — Classes et Objets

Objectif : Le but de ce TP est de créer une classe permettant de représenter des matrices de $\mathcal{M}_n(\mathbb{R})$, afin d'illustrer les concepts de classe, d'objet, et d'encapsulation.

Exercice 1 – *Matrice sous forme de liste de listes*

Le premier pas dans la représentation d'une matrice est de la voir comme une liste de listes. Par exemple, la matrice de $\mathcal{M}_2(\mathbb{R})$

$$\begin{pmatrix} 1 & 3 \\ 0 & 2 \end{pmatrix}$$

s'écrira `[[1, 3], [0, 2]]`.

Question 1.1. La liste `[[1, 3], [0, 2]]` contient deux éléments, `[1, 3]` et `[0, 2]`, qui sont eux-mêmes des listes. À quelles parties de la matrice ces deux listes correspondent-elles ?

Question 1.2. Écrire une fonction `get_M_ij` qui prend en argument une matrice `M` sous forme de liste de listes, ainsi que deux entiers `i` et `j` compris entre 0 et $n-1$, et qui renvoie l'élément M_{ij} de la matrice.

Tester cette fonction sur la matrice ci-dessus et sur une matrice de $\mathcal{M}_3(\mathbb{R})$ de votre choix.

Indications :

- Avec `M = [[1, 3], [0, 2]]`, à quoi correspondent `M[0]` et `M[1]` ?
- Quel sont les types de `M[0]` et `M[1]` ?

Question 1.3. Aurait-il été possible de représenter la matrice M d'une façon différente ?

Exercice 2 – *Classe `matrice`*

On va à présent créer une classe permettant de représenter une matrice carrée. Un intérêt d'implémenter une classe dans ce cas est de pouvoir regrouper toutes les méthodes dont on peut avoir besoin (affichage, addition, etc.).

Le seul attribut de cette classe sera `M`, une liste de listes représentant une matrice.

Les méthodes de cette classe seront les suivantes :

- un constructeur, qui prend en entrée une liste de listes et affecte l'attribut `M`,
- une fonction qui permet d'afficher la matrice avec `print`,
- une fonction `get_dim` qui renvoie la dimension de M (son nombre de lignes ou de colonnes),
- une fonction `get_M_ij` qui renvoie la valeur de M_{ij} (comme à l'**Exercice 1**),
- un mutateur pour M_{ij}, qui prend en entrée un `float` et deux `int`, et qui va modifier la valeur de M_{ij},
- une fonction `addition`, qui prend en entrée une autre instance `N` de la classe `matrice`, et qui renvoie la somme de `M` et de `N`.

Tester cette classe, ainsi que ses méthodes, sur les matrices

$$M = \begin{pmatrix} 1 & 3 & 0 \\ 0 & 2 & 7 \\ 4 & 1 & 0 \end{pmatrix} \quad \text{et} \quad N = \begin{pmatrix} 2 & 8 & 3 \\ 1 & 0 & 4 \\ 5 & 0 & 2 \end{pmatrix}.$$

Exercice 3 – *Encapsulation*

Reprendre l'**Exercice 2** en rendant l'attribut `M` privé. Seule la classe doit être modifiée (penser notamment à ajouter un accesseur pour `M`). Le corps du programme (les tests des méthodes de la classe) doit rester identique.

Exercice 4 – *Nouvelle représentation des matrices*

Plutôt que de représenter une matrice par une liste de listes, il est souvent plus avantageux, en termes de coût mémoire, de représenter une matrice en une seule liste : à présent, la matrice de $\mathcal{M}_2(\mathbb{R})$

$$\begin{pmatrix} 1 & 3 \\ 0 & 2 \end{pmatrix}$$

s'écrira `[1, 3, 0, 2]` (sous forme d'une liste plutôt qu'une liste de listes).

Reprendre l'**Exercice 3** en modifiant cette représentation des matrices. Il faudra notamment faire attention à bien modifier les fonctions `get_M_ij`, `set_M_ij`, `get_dim` et `addition`.

De même qu'à l'**Exercice 3**, on ne modifiera que la classe `matrice` et pas le corps du programme ayant servi à faire les tests (à part les définitions de `M` et `N`).

Ceci permet d'illustrer un avantage de l'encapsulation : on a modifié en profondeur la structure de données, mais l'utilisateur a toujours accès aux mêmes méthodes (`addition`, etc) qui se comportent de la même façon.