TP08 : boucles imbriquées

Lycée Chateaubriand – PCSI1

1 Matrices (ou listes de listes)

Ce TP porte sur la manipulation de matrices (ou tableaux), qui serviront par exemple pour la manipulation et le traitement d'images. En Python, on représente les matrices par des listes de listes.

Par exemple, `mat = [[3, 5, 1, 0], [0, 1, 1, 0], [2, 2, -1, 3]]` définit la matrice

$$\texttt{mat} = \begin{pmatrix} 3 & 5 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 2 & 2 & -1 & 3 \end{pmatrix}.$$

Syntaxe pour la manipulation d'une matrice `mat` :

- accès à la ligne d'indice `i` : `mat[i]` ; par exemple : `mat[0]` renvoie `[3, 5, 1, 0]`.
- accès à l'élément de coordonnées (`i`, `j`) : `mat[i][j]` ; par exemple : `mat[0][1]` renvoie `5`.
- nombre de lignes : `len(mat)`.
- nombre de cellules de la `i`ème ligne : `len(mat[i])`.
  Si `mat` représente bien une matrice, le nombre de cellules est le même sur chaque ligne. Le nombre de colonnes d'une matrice peut donc s'obtenir avec `len(mat[0]`, et le nombre total d'éléments de la matrice est `len(mat)*len(mat[0])`.

Remarque : `NumPy` est une bibliothèque Python qui définit toute une série de fonctions permettant de manipuler efficacement des listes et des tableaux. C'est très pratique pour réaliser des calculs efficaces sur ce genre de structures. Le but de ce TP n'est pas d'**utiliser** ce type de fonctions mais de réfléchir à la façon de les **définir**. On n'inclura donc pas le module `NumPy` dans ce TP.

1. Écrire une fonction `ligne` qui prend en paramètre une matrice `mat` et un entier `k` $<$ `len(mat)`, et renvoie la colonne d'indice `k` de la matrice `mat`.

Test. `ligne([[3, 5, 1, 0], [0, 1, 1, 0], [2, 2, -1, 3]],1) = [0, 1, 1, 0]`

2. Écrire une fonction `colonne` qui prend en paramètre une matrice `mat` et un entier `k` $<$ `len(mat[0])`, et renvoie la colonne d'indice `k` de la matrice `mat`.

Test. `colonne([[3, 5, 1, 0], [0, 1, 1, 0], [2, 2, -1, 3]],1) = [5, 1, 2]`

Syntaxe pour la création d'une matrice `mat` :

On utilise les opérations classiques de création de listes pour créer des matrices :

- définition explicite : `mat = [ [1,2], [3,4], [5,6] ]` définit une matrice à 3 lignes et deux colonnes ;
- définition en extension : `mat = [ [0 for j in range(5)] for i in range(15)]` définit une matrice de zéros à quinze lignes et cinq colonnes.

3. On a vu une autre façon de définir des grandes listes contenant une seule valeur : `liste = [0]*15` définit une liste de longueur 15 ne contenant que des zéros. On pourrait donc utiliser cette opération pour créer des matrices.

a. en utilisant l'opération `*`, créer une matrice carrée `mat` de 15 lignes et 15 colonnes ne contenant que des zéros. Vérifier que `mat` ne contient bien que des zéros.

b. en utilisant *une seule* boucle `for`, mettre des `1` sur la diagonale de cette matrice (les éléments de la diagonale sont les éléments `mat[i][i]`).

c. vérifier que `mat` ne contient que des `1`.

d. entrer votre programme sur la page suivante pour observer ce qui s'est passé : `https://yannsalmon.forge.apps.education.fr/python-debug/`.

e. créer une matrice similaire en utilisant la définition de listes en extension, et placer des `1` sur la diagonale. Vérifier que le résultat est correct.

Attention : problèmes de dépendances.

Quand on écrit `mat = [ [0]*m ]*n`, la matrice `mat` contient en fait `n` copies de **la même liste** `[0]*m`. Si on modifie un élément d'une de ces listes, alors cette modification apparaitra sur toutes les lignes, puisque chaque ligne correspond à une seule et même liste.

Pour créer une (grande) matrice, on utilisera donc exclusivement les définitions par extension :

```
mat = [ [0 for j in range(10)] for i in range(5) ]
```

On peut bien entendu créer des matrices plus compliquées avec ce type de définitions :

```
mat = [ [i*j for j in range(10)] for i in range(10) ]
```

Un problème similaire se produit quand on veut dupliquer une matrice. Si `mat` est une matrice, l'affectation `mat2 = mat` ne crée pas vraiment une nouvelle matrice : elle permet juste d'accéder à la matrice `mat` avec un autre nom de variable `mat2` (ce qui n'a généralement pas trop d'intérêt).

Ce problème se pose déjà pour les listes : si `liste` est une liste, on peut la dupliquer avec l'une des solutions ci-dessous :

- `liste2 = liste.copy()`
- `liste2 = [elt for elt in liste]`
- `liste2 = [liste[ind] for ind in range(len(liste))]`

Dans les trois cas, on obtient une nouvelle liste, indépendante de la première.

Pour dupliquer une matrice, on utilisera ce type de définitions deux fois (pour les lignes et pour les colonnes). Par exemple

```
mat2 = [ ligne.copy() for ligne in mat ]
```

On aurait aussi pu écrire

```
mat2 = [ [mat[i][j] for j in range(len(mat[0]))] for i in range(len(mat)) ]
```

2 Dessiner dans une matrice

Pour cette partie, on importe le sous-module `pyplot` du module `matplotlib`, pour pouvoir utiliser la fonction `matshow`. Cette fonction permet de visualiser le contenu d'une matrice.

On commencera par définir une fonction d'affichage :

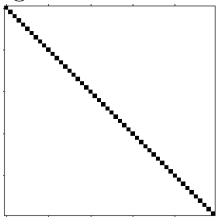
```
import matplotlib.pyplot as plt

def affiche(mat):
    plt.matshow(mat, cmap = 'Greys')
    plt.show()
```

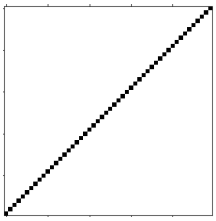
Dans chacune des fonctions des questions 4 à **??**, on créera une matrice carrée de taille `n` × `n` remplie de 0, et on ajoutera des 1 dans certaines cases afin d'obtenir le motif demandé.

4. Écrire une fonction `diag1` qui prend en paramètre un entier `n` et qui renvoie une matrice pleine de `0`, sauf sur une diagonale où il y a des `1` (sans utiliser d'instruction conditionnelle). Écrire une fonction `diag2` pour l'autre diagonale.

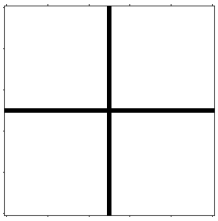
Test.



affichage de `diag1(50)`



affichage de `diag2(50)`

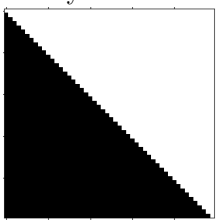


affichage de `croix(50)`

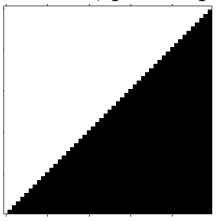
5. Écrire une fonction `croix` qui prend en paramètre un entier impair `n` et qui renvoie une matrice pleine de `0` sauf sur la ligne et la colonne médianes (indice `n//2`) où il y a des `1`. Éviter d'utiliser une instruction conditionnelle.

6. Écrire une fonction `triangle1` et une fonction `triangle2` qui prennent en paramètre un entier `n` et qui renvoient une matrice pleine de `0` sauf strictement sous une diagonale où il y a des `1` (sans utiliser d'instruction conditionnelle). On essayera de bien réfléchir aux indices, plutôt que procéder par essais-erreurs.

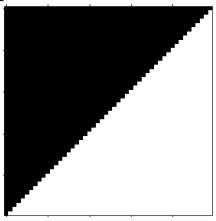
Test.



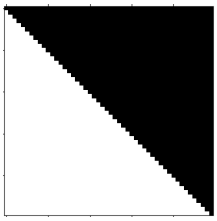
affichage de `triangle1(50)`



affichage de `triangle2(50)`



affichage de `triangle3(50)`



affichage de `triangle4(50)`

7. Écrire une fonction `triangle3` et une fonction `triangle4` qui prennent en paramètre un entier `n` et qui renvoient une matrice pleine de `0` sauf sur et au-dessus d'une diagonale où il y a des `1`.

3 Sommes doubles

8. Définit en Python une fonction `somme`, prenant en entrée une matrice `mat` contenant des nombres, et renvoyant la somme de tous les éléments de `mat`.

9. Écrire une fonction `somme_double`, qui prend en paramètre un entier `n`, crée une matrice carrée `mat` de taille `n` $\times$ `n` telle que `mat[i][j]` $= \frac{1}{i^2+j^2}$, et utilise cette matrice pour calculer $\sum_{i=1}^{n}\sum_{j=1}^{n}\frac{1}{i^2+j^2}$.

Test. `somme_double(100) = 6.425718420103783`

10. On souhaite maintenant calculer $\sum_{i=1}^{n}\sum_{j=1}^{i}\frac{1}{i^2+j^2}$. Noter que ce n'est pas la moitié de la somme précédente.

a. Écrire une fonction `somme_aux(i,n)` qui calcule $\sum_{j=1}^{n}\frac{1}{i^2+j^2}$.

b. Écrire une fonction `somme_double2(n)` qui utilise la fonction précédente et calcule la somme $\sum_{i=1}^{n}\sum_{j=i}^{n}\frac{1}{i^2+j^2}$.

c. Écrire une fonction `somme_double3(n)` qui calcule le même résultat sans utiliser la fonction intermédiaire `somme_aux`.

Test. `somme_aux(5,10) = 0.2002149449201179`. Pour les sommes doubles, on obtient `somme_double2(1000) = 5.424283767520265` et `somme_double3(1000) = 5.424283767520553`. On notera que, même si les deux méthodes semblent identiques, les résultats diffèrent légèrement à cause des erreurs d'arrondis.

4 Deux éléments les plus proches dans une liste

11. Écrire une fonction `plusProches` prenant en entrée une liste `l` et qui renvoie les indices des deux éléments les plus proches présents dans cette liste.
Indication : on adapte la recherche d'un minimum en parcourant un ensemble de couples d'indices. On peut utiliser la fonction valeur absolue `abs`.

Test. `plusProches([23, 15, 67, 32, 45, 89, 42, 6]) = (6, 4)` ou `(4, 6)`