NUMPY ET SCIPY

J. Mellac

Mars 2014

Chapitre 1

Vecteurs, matrices et tableaux

Il est d'abord nécessaire d'importer la bibliothèque **numpy** par **import numpy** ou **import numpy as np** ou **from numpy import ***. C'est en général la deuxième méthode qui est utilisée. Les fonctions de numpy seront alors accessibles par leur nom qualifié **np.nomfonction**.

1.1 Notions de base.

La bibliothèque **numpy** introduit le type **array** qui est proche d'une liste avec la différence que tous les éléments doivent être de même type.
Les éléments du type **array** sont appelés tableaux, il peuvent être à une ou plusieurs dimension, en pratique 1 ou 2 qui dans ce cas sont des matrices. Vecteurs, matrices lignes, matrice colonne :

Un vecteur est un tableau à une dimension composé d'entiers ou de flottants.

```
import numpy as np # charge le numpy et le renomme np

 >>> # un vecteur
>>> np.array([1,3,5])
array([1, 3, 5])
>>> # une matrice-ligne
>>> np.array([[1,3,5]])
array([[1, 3, 5]])
>>> # une matrice-colonne
>>> np.array([[1],[3],[5]])
array([[1],
[3],
[5]])
```

Matrices :

```
 >>> A = np.array([[5,3,2],[7,4,1]]); A # forme et affiche une matrice d'entiers
array([[5, 3, 2],
       [7, 4, 1]])
```

Les coefficients entiers de A peuvent être convertis au format 'float' ou 'complex' :

```
>>> A.astype(float)
array([[ 5., 3., 2.],
       [ 7., 4., 1.]])
>>> A.astype(complex)
array([[ 5.+0.j, 3.+0.j, 2.+0.j],
       [ 7.+0.j, 4.+0.j, 1.+0.j]])
```

Attention : les indices de lignes et colonnes commmencent à 0. ainsi $A[0,0]$ repésente A_{11}, $A[1,2]$ représente $A_{2,3}$.

Comme pour les listes il est possible de définir des matrices en compréhension ,c'est à dire à partir de formules, sans citer tous les éléments de la matrices :

```
>>> A = np.array([[10*i+j for j in range(10)] for i in range(5)])
>>> print(A) # la matrice de terme général 10i+j, avec 0<=i<=4, et 0<=j<=9
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47 48 49]]
```

Il est possible d'appliquer une fonction à un tableau ou à une matrice. Il est à noter que **numpy** contient pratiquement tout le module **math** qu'il est donc inutile d'appeler ici pour utiliser la fonction **sqrt**.

```
>>> B=np.sqrt(A)
array([[ 0.        ,  1.        ,  1.41421356,  1.73205081,  2.        ,
         2.23606798,  2.44948974,  2.64575131,  2.82842712,  3.        ],
       [ 3.16227766,  3.31662479,  3.46410162,  3.60555128,  3.74165739,
         3.87298335,  4.        ,  4.12310563,  4.24264069,  4.35889894],
       [ 4.47213595,  4.58257569,  4.69041576,  4.79583152,  4.89897949,
         5.        ,  5.09901951,  5.19615242,  5.29150262,  5.38516481],
       [ 5.47722558,  5.56776436,  5.65685425,  5.74456265,  5.83095189,
         5.91607978,  6.        ,  6.08276253,  6.164414  ,  6.244998  ],
       [ 6.32455532,  6.40312424,  6.4807407 ,  6.55743852,  6.63324958,
         6.70820393,  6.78232998,  6.8556546 ,  6.92820323,  7.        ]])
```

La matrice A a été transformée, chaque terme de la matrice est remplacé par sa racine carrée.

```
B**2
array([[  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.],
       [ 10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.],
       [ 20.,  21.,  22.,  23.,  24.,  25.,  26.,  27.,  28.,  29.],
       [ 30.,  31.,  32.,  33.,  34.,  35.,  36.,  37.,  38.,  39.],
       [ 40.,  41.,  42.,  43.,  44.,  45.,  46.,  47.,  48.,  49.]])
```

Rétablit la matrice d'origine, à ceci prés que les nombres sont des flottants et non des entiers comme dans A, flottants qui sont apparus dans l'utilisation de la racine carrée.
Remarquer que $E**2$ ne réalise pas un calcul matriciel.

Ici les coefficients sont des flottants : si veut des entiers, il suffit de rajouter l'option **dtype='int'**.
Voici une autre méthode pour calculer la matrice A.

```
>>> B = np.fromfunction(lambda x,y : 10*x+y,(5,10))
>>> print(B)
[[ 0.  1.  2.  3.  4.  5.  6.  7.  8.   9.]
 [ 10. 11. 12. 13. 14. 15. 16. 17. 18. 19.]
 [ 20. 21. 22. 23. 24. 25. 26. 27. 28. 29.]
 [ 30. 31. 32. 33. 34. 35. 36. 37. 38. 39.]
 [ 40. 41. 42. 43. 44. 45. 46. 47. 48. 49.]]
```

Il est possible de créer des matrices aléatoires :

Matrice (3,5) d'entiers compris entre 0 et 9.

```
C=np.random.randint(10,size=(3,5))
>>> C
array([[9, 3, 1, 0, 1],
       [4, 7, 5, 1, 9],
       [7, 9, 9, 1, 4]])
```

quatre lignes, cinq colonnes, coefficients dans $[0, 1[$

```
>>> A = np.random.rand(4,5)
>>> print(A)
[[ 0.4672486 0.42452394 0.55652697 0.50122566 0.45039697]
 [ 0.10245706 0.39642783 0.18395313 0.43191611 0.59490124]
 [ 0.97133547 0.06577487 0.33274442 0.39404957 0.34601463]
 [ 0.60141568 0.01085386 0.69123586 0.33062893 0.68429281]]
```

Création de matrices nulles ou de matrices constantes.

```
Matrice nulle de type (3,4)

np.zeros([3, 4])
array([[ 0., 0., 0., 0.],
       [ 0., 0., 0., 0.],
       [ 0., 0., 0., 0.]])

Matrice composée de 1 de type (3,4)

np.ones([3,4])
array([[ 1., 1., 1., 1.],
       [ 1., 1., 1., 1.],
       [ 1., 1., 1., 1.]])
```

dimension d'un tableau.

Si A est un tableau de numpy, à une dimension ou une matrice.
np.shape(A) donne le nombre de lignes et de colonnes d'une matrice sous forme d'un tuple, ici un couple.
Pour une matrice **len(A)** donne le nombre de lignes de A et **np.alen(A)** le nombre de colonnes de A. Il faut remarquer que **len(A)** s'utilise sans **np.** contrairement à **np.len(A)**.

size donne le nombre total d'éléments, et par exemple np pour une matrice de type (n,p).

ndim renvoie le nombre d'indices nécessaires au parcours du tableau (usuellement : 1 pour un vecteur, 2 pour une matrice).

```
A=np.random.rand(5,4)
>>> A
array([[ 0.64627153,  0.83166078,  0.52450604,  0.25472639],
       [ 0.18879927,  0.70915753,  0.92600669,  0.62278505],
       [ 0.99826631,  0.13139493,  0.24614402,  0.9075739 ],
       [ 0.86785393,  0.14688378,  0.64664868,  0.54796176],
       [ 0.87425469,  0.01746309,  0.65253954,  0.94076863]])

>>> np.shape(A)
(5, 4)
>>> np.size(A)
20
>>> np.ndim(A)
2
```

Il est possible de considérer de considérer ces fonctions comme attributs d'un tableau, on peut par exemple utiliser **A.shape** au lieu de **np.shape(A)**.

coupe (ou slicing) d'un tableau.

Pour un tableau A de dimension 1 la méthode est identique à celle utilisée pour une liste.

On procède en effectuant une coupe suivant la première et/ou suivant la deuxième dimension.

```
m = np.array([[10*i+j for j in range(8)] for i in range(5)]); m
array([[ 0,  1,   2,  3,  4,  5,  6,  7],
       [10, 11, 12, 13, 14, 15, 16, 17],
       [20, 21, 22, 23, 24, 25, 26, 27],
       [30, 31, 32, 33, 34, 35, 36, 37],
       [40, 41, 42, 43, 44, 45, 46, 47]])

#  élément en position (3,5)
>>> m[3,5]
35
>>> m[3]
# vecteur ligne 4
array([30, 31, 32, 33, 34, 35, 36, 37])
# vecteur colonne 5
>>> m[:,5]
array([ 5, 15, 25, 35, 45])

# lignes 1 à 3, colonnes 2 à 5
m[1:4,2:6]
array([[12, 13, 14, 15],
       [22, 23, 24, 25],
       [32, 33, 34, 35]])
```

```
m[:3,:2]
trois premières lignes deux premières colonnes
array([[ 0, 1],
       [10, 11],
        [20, 21]])


>>> m[2:,4:]
# a partir de la ligne 2 a partir de la colonne 4
array([[24, 25, 26, 27],
       [34, 35, 36, 37],
       [44, 45, 46, 47]])

Les :: correspondent a un pas: une ligne sur deux, une colonne sur trois
m[::2,::3]
array([[ 0, 3, 6],
       [20, 23, 26],
       [40, 43, 46]])
```

1.2 Opérations sur les tableaux ou les matrices

Modification d'une matrice.

Copie d'une ligne ou d'une colonne d'une matrice

```
# Commençons par l'écriture d'un coefficient :
>>> m
array([[ 0, 1, 2, 3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])

>>> m[1,2] = 999; m

array([[ 0, 1, 2, 3],
       [ 10, 11, 999, 13],
       [ 20, 21, 22, 23]])

# Continuons par l'écriture d'une ligne entière :
>>> m
array([[ 0, 1, 2, 3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])

>>> m[2] = [6, 7, 8, 9]; m

array([[ 0, 1, 2, 3],
       [10, 11, 12, 13],
       [ 6, 7, 8, 9]])
```

```
# Terminons par l'écriture d'une colonne entière.
>>> m
array([[ 0, 1, 2, 3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])

>>> m[:,1] = [444, 555, 666]; m

array([[ 0, 444, 2, 3],
       [ 10, 555, 12, 13],
       [ 20, 666, 22, 23]])
```

La copie de matrices suit le même principe que la copie de listes python. L'instruction **B=A** ne cée pas une nouvelle matrice mais un alias de A. Toute modification ultérieure de B modifierait A. Une copie véritable de A est obtnue par $B = A.copy()$.

Redimensionnement d'un tableau.

La principale méthode pour modifier la taille d'un tableau est **reshape**. Le redimensionnement d'un tableau est quand même soumis à la contrainte que le nombre total d'éléments doit rester constant. On pourra ainsi passer (dans les deux sens) d'un vecteur de taille n à une matrice de taille (p, q) ou à une matrice de taille (r, s) à condition que n = pq = rs. Si A est un tableau numpy, l'expression **A.reshape(n,p)** renvoie une copie redimensionnée (le contenu de la variable initiale n'est donc pas affecté, comme on le constate ci-dessous).

```
a = np.array(range(6)); a
array([0, 1, 2, 3, 4, 5])

>>> a.reshape(2,3)
array([[0, 1, 2],
       [3, 4, 5]])
>>> a.reshape(3,2)
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> a.reshape(6,1)
array([[0],
       [1],
       [2],
       [3],
       [4],
       [5]])

 >>> a
array([0, 1, 2, 3, 4, 5])
```

Transposition d'une matrice.

La fonction (ou la méthode) transpose , ou plus simplement la méthode T , renvoient la transposée d'une matrice :

```
>>> a
array([[ 1, 2, 3, 4],
```

```
       [ 5, 6, 7, 8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])

>>> a.transpose()
array([[ 1, 5, 9, 13],
       [ 2, 6, 10, 14],
       [ 3, 7, 11, 15],
       [ 4, 8, 12, 16]])

>>> a.T
array([[ 1, 5, 9, 13],
       [ 2, 6, 10, 14],
       [ 3, 7, 11, 15],
       [ 4, 8, 12, 16]])
```

Opérations usuelles sur les matrices.

Etant donné deux matrices $A, B \in \mathcal{M}_{n,p}(K)$ leur somme est obtenue par $A + B$.

```
>>> C
array([[3, 2, 6],
       [0, 2, 0],
       [0, 1, 7],
       [3, 6, 8],
       [9, 7, 9]])
>>> D
array([[9, 6, 1],
       [3, 8, 1],
       [7, 6, 2],
       [1, 8, 7],
       [3, 4, 9]])
>>> C+D
array([[12,  8,  7],
       [ 3, 10,  1],
       [ 7,  7,  9],
       [ 4, 14, 15],
       [12, 11, 18]], dtype=int32)
```

L'instruction $A + 8$ ajoute 8 à chaque terme de A, L'instruction $A * 8$ multiplie par 8 chaque terme de A.

```
>>> C+8
array([[11, 10, 14],
       [ 8, 10,  8],
       [ 8,  9, 15],
       [11, 14, 16],
       [17, 15, 17]], dtype=int32)
>>> D*8
array([[72, 48,  8],
       [24, 64,  8],
       [56, 48, 16],
       [ 8, 64, 56],
       [24, 32, 72]], dtype=int32)
```

Etant donné deux matrices $(A, B) \in \mathcal{M}_{n,p}(K) \times \mathcal{M}_{p,r}(K)$, le produit $A * B$ est obtenue par **dot(A,B)**, l'instruction $A * B$ n'a de sns que si A et B sont de même type et ne réalise pas un produit matriciel mais donne la matrice dont le coefficient en position (i, j) est le produit des coefficients de A et de B en position (i, j).

```
>>> A
array([[7, 9, 0, 8],
       [7, 2, 7, 5],
       [0, 6, 1, 3],
       [6, 3, 2, 0],
       [8, 9, 9, 2]])
>>> B
array([[5, 0, 7, 6, 9, 4],
       [5, 2, 5, 6, 9, 8],
       [6, 9, 7, 5, 6, 4],
       [4, 6, 7, 9, 7, 3]])
>>> C=dot(A,B) #produit matriciel de A par B
array([[112,  66, 150, 168, 200, 124],
       [107,  97, 143, 134, 158,  87],
       [ 48,  39,  58,  68,  81,  61],
       [ 57,  24,  71,  64,  93,  56],
       [147, 111, 178, 165, 221, 146]])
```

Exercice

Ecrire une fonction **def produit(A,B) :** qui prend en entrée des matrices $(A, B) \in \mathcal{M}_{n,p}(K) \times \mathcal{M}_{p,r}(K)$ et qui renvoie C produit matriciel de A par B.

1.3 Matrices particulières

On a déjà rencontré les matrices **zeros** et **ones**. On peut également citer l'instruction $A.fill(b)$ qui remplit un tableau déjà existant avec la constante b ou les instructions $zeros\_like(A)$ et $ones\_like(A)$ qui crée une matrice de même type que A composée de zéros (respectivement de 1).

L'instruction **np.arange(n)** crée un tableau (un vecteur) dont les termes sont les entiers de 0 à $n - 1$.

La fonction **np.eye(n,k)** permet de fabriquer la matrice identité ou plus généralement une matrice dont tous les coefficients sont nuls sauf ceux d'une certaine « parallèle » à la diagonale et qui valent 1. Le premier argument (obligatoire) donne le nombre n de lignes. Le second argument (facultatif) donne le nombre p de colonnes (par défaut p = n donc la matrice est carrée). Un argument facultatif nommé k = : : permet de spécifier un décalage de la diagonale de 1 au dessus (si $k > 0$) ou en-dessous (si $k < 0$) de la diagonale principale. Enfin, on peut ajouter un argument pour fixer le type des données (float par défaut).

```
 np.eye(4)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])

>>> np.eye(6,k=-2)
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.,  0.]])np.diag(A,k=2)
array([7, 0, 6, 2])
```

La fonction **diag** renvoie la diagonale d'une matrice. Avec un deuxième argument (facultatif) k, on obtient une surdiagonale (si $k > 0$) ou une sous-diagonale (si $k < 0$).
Cette permet aussi de construire des matrices diagonales, dont les termes sont donnés.

```
A=np.random.randint(10,size=(6,6))
>>> A
array([[9, 9, 7, 9, 4, 9],
       [7, 4, 6, 0, 9, 1],
       [6, 3, 4, 0, 6, 8],
       [0, 5, 4, 6, 4, 2],
       [3, 9, 6, 4, 3, 5],
       [1, 8, 4, 9, 5, 7]])
>>> np.diag(A)
array([9, 4, 4, 6, 3, 7])

np.diag(A,k=2)
array([7, 0, 6, 2])

np.diag([1,2,3,4])
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

Exercice

Ecrire une fonction **def T(n)** qui renvoie la matrice strictement triangulaire d'ordre n donc tous les coefficients sont nuls, sauf ceux situés immédiatement au-dessus de la diagonale et qui valent 1.