

# Manipuler les matrices avec Numpy

Ô toi camarade qui s'apprête à lire ces diapositives, tu as décidé d'apprendre à manipuler les matrices avec le module numpy et je t'en félicite !

Avant de commencer, tu dois savoir que numpy ne sert pas seulement à manipuler les matrices mais contient également d'autres fonctions diverses qui pourront te servir dans l'avenir.

Tu vas ainsi en découvrir quelques-unes et tu te documentera toi-même si tu as soif de pouvoir !

# Prérequis

Dans ces diapositives tu vas apprendre à « manipuler » les matrices 1D et 2D (création, modification, etc.). J'estime que tu sais ce qu'est une matrice et toutes les notions mathématiques qui y sont associées (calcul matriciel, déterminant, ...). Si tel n'est pas le cas, je t'invite à revoir tes cours sur les matrices !

Les matrices ressemblent très fortement aux listes en programmation (surtout en python). Je pourrais même dire que ce sont des listes, mais dans un format différent. Donc si tu ne sais pas manipuler les listes, je t'invite encore à revoir tes cours sur les listes avant de débiter. C'est un vrai plus.

J'admire ton courage et ta détermination si tes yeux sont rivés sur cette diapositive ! Cela signifie que tu as (revu tes cours et a maintenant) toutes les bases nécessaires pour la suite des événements !

Tu es prêt ? Tu vas découvrir le saint-graal dans les diapositives qui suivent, accroche-toi bien ! Passé la diapositive suivante, tu auras atteint le point de non retour !

# Voici un aperçu de l'aventure qui t'attend

- Avant de commencer
- Création de matrices
- Sélection de données
- Autres créations particulières
- Fonctions
- Algèbre linéaire
- Lien avec matplotlib

# Avant de commencer

- Assure toi que numpy est bien installé sur ton environnement de travail.
- Importe le au début de ton/tes script(s).

```
import numpy
```

- Numpy est généralement abrégé par « np », mais libre à toi de choisir le mot raccourci que tu souhaite.

```
import numpy as np  
import numpy as boomboomboom  
import numpy as unmodule  
import numpy as quelquechose
```

# Création de matrices

# Quelques petits rappels tout de même

Vous devez tous le savoir, une matrice c'est un tableau de données de certaines dimensions, données entourées par des ( ) ou des [ ]. 2 exemples ci-dessous :

$$\begin{pmatrix} 30 & 5 & 80 \\ 45 & 87 & 18 \\ 14 & 11 & 55 \end{pmatrix} \text{ ou } \begin{bmatrix} 56 & 20 & 18 \\ 1 & 11 & 36 \\ 30 & 47 & 25 \end{bmatrix}$$

En langage de programmation python, pour pourrez créer les matrices avec des ( ) ou des [ ]. Mais leur affichage (dans le terminal) se fera toujours avec des [ ], comme les listes.



# Type matrice (array)

Une matrice peut être reconnue avec son type « **array** » (<'numpy.ndarray'>), mais également avec son affichage dans le terminal.

```
[[1, 2], [2, 3]]  
<type 'list'>  
[[1 2]  
 [2 3]]  
<type 'numpy.ndarray'>
```

Les données d'une liste ou d'un tuple seront affichées tout en ligne, alors que chaque ligne la matrice sera affichée automatiquement sur une ligne différente (pas besoin de '\n') dans le terminal.

# Créer une matrice

Il existe plusieurs façon de créer une matrice.

La plus simple est de créer une liste (ou un tuple), puis de changer son type en la mettant dans les ( ) de **numpy.array()**. Vous pouvez également transformer votre matrice en une simple liste par le même procédé.

Remarque : Avec cette manière de créer des matrices, vous pouvez insérer des éléments autres que des entiers ou réels. Mais sachez qu'avec les autres manières de créer les matrices, vous ne pourrez pas faire cette manipulation.

# Example

```
p = [[1,2], [2,3]]  
P = numpy.array(p)
```

```
print type(p)  
print p
```

```
print type(P)  
print P
```

```
<type 'list'>  
[[1, 2], [2, 3]]  
<type 'numpy.ndarray'>  
[[1 2]  
 [2 3]]
```

# Créations particulières

Il est courant d'initialiser une matrice de dimension **n x m** avec certaines valeurs lors de la création.

Initialisation	Commande
Avec des 0	<code>numpy.zeros((n, m))</code>
Avec des 1	<code>numpy.ones((n, m))</code>
Avec la valeur a	<code>numpy.full((n, m), a)</code>
Identité carré	<code>numpy.eye(n)</code>

Pour des matrices initialisées avec une de ces 4 commandes, celles-ci ne pourront contenir que des valeurs entières ou réelles (pas de caractères ou autre).

# Examples

```
p = numpy.zeros((3,4))  
print p
```

```
[[ 0.  0.  0.  0.]  
 [ 0.  0.  0.  0.]  
 [ 0.  0.  0.  0.]]
```

```
p = numpy.ones((2,2))  
print p
```

```
[[ 1.  1.]  
 [ 1.  1.]]
```

```
p = numpy.full((5, 4), 9)  
print p
```

```
[[ 9.  9.  9.  9.]  
 [ 9.  9.  9.  9.]  
 [ 9.  9.  9.  9.]  
 [ 9.  9.  9.  9.]  
 [ 9.  9.  9.  9.]]
```

```
p = numpy.eye(4)  
print p
```

```
[[ 1.  0.  0.  0.]  
 [ 0.  1.  0.  0.]  
 [ 0.  0.  1.  0.]  
 [ 0.  0.  0.  1.]]
```

# Sélection de données

La sélection des valeurs d'une matrice s'effectue de la même manière qu'avec les listes, avec les `[ ]`, puis en les répétant, selon les dimensions de la liste.

Toutefois, il existe quelques sélections particulières propres aux matrices, non applicables aux listes.

Prenons une liste 2D.

`L = [[1, 2, 3], [4, 5, 6]]`

Pour sélectionner la valeur 4, vous taperiez : `L[1][0]`.

Dans le cas où L est une matrice (type « **array** »), vous pouvez sélectionner la valeur 4 de cette façon : `L[1,0]`.

Si elle est plus simple que la sélection avec les listes ? Je ne saurais y répondre. Mais cela permet de faire des sélections particulières beaucoup plus aisées qu'avec les listes.



# Exemple

```
L = [[1, 2, 3], [4, 5, 6]]  
print L[1][0]
```

```
M = numpy.array(L)  
print M[1][0]  
print M[1,0]
```

```
4  
4  
4
```

# Sélectionner une ligne

Une ligne dans une matrice correspond à une liste dans une liste de listes.

Pour sélectionner la première liste de  $L$ , il suffit de taper :  $L[0]$ . Et pour sélectionner la  $x$ -ème liste de  $L$  :  $L[x-1]$  (indice commençant à 0).

En ce qui concerne les matrices, vous pouvez soit utiliser la même notation, soit la suivante :  $L[0,:]$  pour la première ligne, et  $L[x-1,:]$  pour la  $x$ -ème ligne.

Vous trouverez cela peut être bizarre, mais c'est comme cela.

# Example

```
L = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]  
print L[0]  
print L[2]
```

```
M = numpy.array(L)  
print M[0]  
print M[0,:]  
print M[2]  
print M[2,:]
```

```
[1, 2, 3]  
[7, 8, 9]  
[1 2 3]  
[1 2 3]  
[7 8 9]  
[7 8 9]
```

# Sélectionner plusieurs lignes

Comme pour les listes, vous pouvez choisir plusieurs lignes à la fois avec la syntaxe suivante :

**$M[x : y]$  ou bien  $M[x : y,:]$**

Dans ce cas, vous prenez les listes/lignes d'indices  $x$  à  $y$  (exclu). Les espaces de part et d'autre du « : » ne sont pas nécessaires.

Si vous souhaitez prendre des listes/lignes qui ne se suivent pas, vous devez le faire « à l'ancienne » :  $M[0]$ ,  $M[4]$ ,  $M[7]$ , ...

# Exemple

```
L = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

```
M = numpy.array(L)
print M
print M[1:3,:]
print M[1:3]
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[[4 5 6]
 [7 8 9]]
[[4 5 6]
 [7 8 9]]
```

# Sélectionner une colonne

Pour une liste, sélectionner toutes les valeurs d'une même « colonne » (ayant le même indice dans chaque liste puisqu'on ne parle pas de colonne dans une liste), il fallait faire une boucle qui parcourait toutes les listes de la liste.

Dans le cas d'une matrice, récupérer une colonne peut se faire plus simplement. Pour récupérer la première colonne (indice 0) : `L[:,0]`. Et pour la x-ème colonne : `L[:,x-1]`.

Avouez que c'est beaucoup plus simple ?

# Exemple

```
L = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

```
M = numpy.array(L)
```

```
print M
```

```
print M[:,1]
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[ 2  5  8 11]
```

La colonne récupérée est écrite en une ligne, mais il y a bien 4 lignes !

# Sélectionner plusieurs colonnes

Même principe que pour récupérer plusieurs lignes.

**$M[:,x : y]$**

Ici, les colonnes  $x$  à  $y$  (exclu) sont récupérées et les données sont bien affichées en colonnes !



# Example

```
L = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

```
M = numpy.array(L)
```

```
print M
```

```
print M[:,0:2]
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

```
[[ 1  2]
 [ 4  5]
 [ 7  8]
 [10 11]]
```

# Autres créations particulières

## Autres créations particulières

Je vous ai précisé que numpy regorge de pleins de petites fonctions. Certaines d'en elles permettent également la création de matrices. En voici quelques-unes.

# numpy.arange()

- **numpy.arange(x)**

Matrice 1 colonne, x valeurs (x lignes), valeurs allant de 0 à x (exclu), avec un pas de 1.

- **numpy.arange(x, y)**

Matrice 1 colonne, y-x valeurs (y-x lignes), valeurs allant de x à y (exclu), avec un pas de 1.

- **numpy.arange(x, y, s)**

Même chose que la précédente avec un pas de valeur « s ».

# Examples

```
A = numpy.arange(10)  
print A  
[0 1 2 3 4 5 6 7 8 9]
```

```
A = numpy.arange(3, 17)  
print A  
[ 3  4  5  6  7  8  9 10 11 12 13 14 15 16]
```

```
A = numpy.arange(0, 20, 2)  
print A  
[ 0  2  4  6  8 10 12 14 16 18]
```

# numpy.linspace()

- **numpy.linspace(x, y, num=n)**

Matrice 1 colonne, n valeurs (lignes) allant de x à y (inclu), espacées équitablement.

Si num n'est pas précisé, 50 valeurs par défaut.

# Exemple

```
A = numpy.linspace(1, 50)  
print A
```

```
B = numpy.linspace(10, 20, 14)  
print B
```

Ou bien

```
B = numpy.linspace(10, 20, num=14)
```

```
[ 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. 50.]  
[ 10.          10.76923077  11.53846154  12.30769231  13.07692308  
13.84615385  14.61538462  15.38461538  16.15384615  16.92307692  
17.69230769  18.46153846  19.23076923 20.          ]
```

# numpy.random()

- **numpy.random.rand(x)**

Matrice 1 colonne, x (lignes) valeurs aléatoires entre 0 et 1.

- **numpy.random.rand(x, y)**

Matrice x lignes, y colonnes, valeurs aléatoires entre 0 et 1.

- **numpy.random.randint(x, size=(n, m)) (ou size=n)**

Matrice n lignes, m colonnes (ou 1 ligne, n colonnes (valeurs)), avec des entiers aléatoires entre 0 et x (exclu).

- **numpy.random.randint(x, y, size=(n, m)) (ou size=n)**

Matrice n lignes, m colonnes (ou 1 ligne, n colonnes), avec des entiers aléatoires entre x et y (exclu).



# Examples

```
A = numpy.random.rand(5)
print A
```

```
B = numpy.random.rand(10)
print B
```

```
[ 0.25019985  0.64859322  0.27171108  0.37350872  0.22237431]
[ 0.37636846  0.48536035  0.93342663  0.88912143  0.49778167  0.87610012
 0.13641639  0.58145372  0.46764224  0.7017668 ]
```

```
A = numpy.random.rand(4 , 6)
print A
```

```
[[ 0.90572236  0.55496755  0.22632975  0.48813049  0.85824463  0.26726096]
 [ 0.46989997  0.85516619  0.48184179  0.31479757  0.60974637  0.35450319]
 [ 0.67146989  0.22324678  0.06459359  0.97300147  0.16206329  0.9286243 ]
 [ 0.61280314  0.26528358  0.01148543  0.74129269  0.29352513  0.05971014]]
```

```
A = numpy.random.randint(10, size=4)
print A
```

```
B = numpy.random.randint(10, size=(3, 5))
print B
```

```
[4 8 0 3]
[[6 6 0 3 3]
 [5 7 9 3 4]
 [4 6 3 8 7]]
```

```
A = numpy.random.randint(10, 20, size=8)
print A
```

```
B = numpy.random.randint(5, 100, size=(5, 5))
print B
```

```
[16 16 14 14 10 12 10 17]
[[23 70 20 93 26]
 [93 40 34 73 42]
 [50  7 44 61 59]
 [76 83 72 87 49]
 [52 89 22 69 27]]
```

# Fonctions

# Transposée d'une matrice

La transposée d'une matrice, c'est la même matrice mais les lignes deviennent les colonnes, et les colonnes deviennent les lignes.

Soit votre matrice `M`.

Vous pouvez transposer votre matrice avec :

- `M.T`
- `M.transpose()`
- `numpy.transpose(M)`

Je vous conseille d'affecter votre matrice transposée à une nouvelle variable (ou la même) afin de pouvoir la réutiliser.

# Exemple

```
M = numpy.array([[1, 2, 3, 4], [5, 6, 7, 9]])  
print M  
M2 = M.T  
print M2
```

Ou bien

```
M.transpose()  
numpy.transpose(M)
```

```
[[1 2 3 4]  
 [5 6 7 9]]  
[[1 5]  
 [2 6]  
 [3 7]  
 [4 9]]
```

# Dimensions d'une matrice

- M.shape

Vous retourne un tuple (n, m), matrice M de dimensions n lignes, m colonnes.

Vous pouvez accéder à la valeur n ou m de la même manière qu'une liste ou un tuple avec :

- M.shape[0] pour les lignes
- M.shape[1] pour les colonnes

# Examples

```
M = numpy.array([[1, 2, 3, 4], [5, 6, 7, 9]])  
print M.shape  
print M.shape[0]  
print M.shape[1]
```

```
(2, 4)  
2  
4
```

```
M2 = numpy.array([1, 2, 3, 4])  
print M2.shape  
print M2.shape[0]  
print M2.shape[1]
```

```
(4,)  
4  
Traceback (most recent call last):  
  File "test2.py", line 19, in <module>  
    print M2.shape[1]  
IndexError: tuple index out of range
```

```
M3 = numpy.array([[1], [2], [3], [4]])  
print M3.shape  
print M3.shape[0]  
print M3.shape[1]
```

```
(4, 1)  
4  
1
```

# Produit matriciel

Soit les deux matrices suivantes :

$$M1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad M2 = \begin{bmatrix} 2 & 5 \\ 2 & 4 \end{bmatrix}$$

Si vous souhaitez faire le produit matriciel de ces deux matrices, il ne suffit pas de faire  $M1 \times M2$ . En effet, cette opération ne vous donnera que le produit terme à terme des deux matrices. En gros, vous obtiendrez :

$$\begin{bmatrix} 2 & 10 \\ 6 & 16 \end{bmatrix}$$

# Exemple

```
M1 = numpy.array([[1, 2], [3, 4]])  
M2 = numpy.array([[2, 5], [2, 4]])
```

```
print M1*M2
```

```
[[ 2 10]  
 [ 6 16]]
```



Pour faire le produit matriciel de M1 par M2, il vous suffit d'utiliser la fonction **dot()** comme suit :

**numpy.dot(M1, M2)**

Vous obtiendrez le résultat suivant :

$$\begin{bmatrix} 6 & 13 \\ 14 & 31 \end{bmatrix}$$

Attention ! Il faut le nombre de colonnes de la matrice M1 soit égal au nombre de lignes de la matrice M2, sans quoi vous ne pourrez pas faire cette opération. Mais vous le savez déjà !

# Examples

```
M1 = numpy.array([[1, 2], [3, 4]])  
M2 = numpy.array([[2, 5], [2, 4]])  
  
R = numpy.dot(M1, M2)
```

```
print R
```

```
[[ 6 13]  
 [14 31]]
```

```
4 M1 = numpy.array([[1, 2, 3], [3, 4, 6]])  
5 M2 = numpy.array([[2, 5], [2, 4]])  
6  
7 R = numpy.dot(M1, M2)
```

```
Traceback (most recent call last):  
  File "test.py", line 7, in <module>  
    R = numpy.dot(M1, M2)  
ValueError: objects are not aligned
```

# Minimum, maximum

- **`numpy.minimum(a, b)`**

Donne le minimum entre a et b.

- **`numpy.minimum(M1, M2)`**

M1 et M2 matrices ou listes. Retourne une nouvelle matrice de même dimension que M1 et M2 (doivent être de même dimension), en prenant pour chaque emplacement la valeur minimale des deux matrices.

- **`numpy.minimum(M, x)`**

Pareil qu'au dessus, mais entre une matrice M, et une valeur x.

Même chose avec **`numpy.maximum()`**.

# Examples

```
A = numpy.minimum(34, 12)
```

```
print A
```

```
12
```

```
A = numpy.minimum([1, 4, 3], [12, 1, 19])
```

```
print A
```

```
M1 = numpy.array([[14, 1, 13], [12, 11, 10]])
```

```
M2 = numpy.array([[11, 2, 15], [2, 17, 9]])
```

```
B = numpy.minimum(M1, M2)
```

```
print B
```

```
[1 1 3]
```

```
[[11 1 13]
```

```
 [ 2 11 9]]
```

```
M = numpy.array([[12, 14, 11, 18], [12, 10, 14, 16]])
```

```
A = numpy.minimum(M, 13)
```

```
print A
```

```
[[12 13 11 13]
```

```
 [12 10 13 13]]
```

# Indice minimum, maximum

- **`numpy.argmin(M)`** ou bien **`M.argmin()`**

Retourne l'indice de la valeur minimale de la matrice M. L'indice retournée suit un axe linéaire : si la matrice est de dimension 2x2, les valeurs auront pour indice 0, 1, 2, et 3.

- **`numpy.argmin(M, axis=0)`** ou bien **`M.argmin(0)`**

Retourne une matrice 1 colonne et autant de lignes qu'il y a de colonnes dans la matrice M. Chaque valeur représente l'indice de la ligne qui contient la valeur minimale sur la colonne (0 pour la valeur sur la première ligne, puis augmentation croissante lorsque l'on change de ligne vers le bas).

- **`numpy.argmin(M, axis=1)`** ou bien **`M.argmin(1)`**

Retourne une matrice 1 colonne et autant de lignes qu'il y a de colonnes dans la matrice M. Chaque valeur représente l'indice de la colonne qui contient la valeur minimale sur la ligne.

Même chose pour **`numpy.argmax()`**.

# Exemples

```
M = numpy.array([12, 14, 11, 18])
```

```
I = numpy.argmin(M)
```

```
print I
```

Ou bien

```
I = M.argmin()
```

```
2
```

```
M = numpy.array([[12, 14, 11], [10, 1, 30]])
```

```
I = numpy.argmin(M)
```

```
print I
```

```
4
```

```
M = numpy.random.randint(100, size=(5, 6))  
print M
```

```
I = numpy.argmin(M, axis=0)  
print I
```

Ou bien

```
I = M.argmax(0)
```

```
[[62 95 89 39 68  6]  
 [ 4 80 14 96 56  3]  
 [35  8 66 64 59  0]  
 [67 15 68 39 58 24]  
 [52 12 14 53 22 61]]  
[1 2 1 0 4 2]
```

```
M = numpy.random.randint(100, size=(5, 6))  
print M
```

```
I = numpy.argmin(M, axis=1)  
print I
```

```
[[94 22 60 55 67 39]  
 [81 72 73 42 93 34]  
 [47 19 12 94 63 38]  
 [94 96  0 95  4 55]  
 [69 16 86 59 56 82]]  
[1 5 2 2 1]
```

# Arrondis

- **`numpy.round(M)`** ou bien **`numpy.round(M, decimals)`**

Retourne une matrice de même dimensions que M dont les valeurs ont été arrondis à la valeur entière la plus proche (si valeurs décimales). « decimals » représente de le nombre de décimales après la virgule (0 par défaut).

- **`numpy.ceil(M)`**


Arrondi de chaque valeur de M à la valeur entière « supérieure » la plus proche (1.3 arrondi à 2 par exemple).

- **`numpy.floor(M)`**

Arrondi de chaque valeur de M à la valeur entière « inférieure » la plus proche (1.7 arrondi à 1 par exemple).



# Exemples



```
N = 1.78756467  
R = numpy.round(N)
```

Ou bien

```
R = numpy.around(N)  
2.0
```

```
N = 1.78756467  
R = numpy.round(N, 2)  
  
print R  
1.79
```

```
N = 1.3  
R = numpy.ceil(N)  
  
print R  
2.0
```

```
N = 1.7  
R = numpy.floor(N)  
  
print R  
1.0
```

# Tri

- **`numpy.sort(M)`**

Tri les valeurs de la matrice M, ligne par ligne, de façon croissante

- **`numpy.sort(M, 0)`**

Tri colonne par colonne, de façon croissante

- **`numpy.sort(M, 1)`**

Même chose que `numpy.sort(M)`

# Examples

```
M = numpy.random.randint(100, size=(3, 4))  
print M
```

```
S = numpy.sort(M)  
print S
```

```
[[84 14 66 34]  
 [18 97 47 71]  
 [73 93 68 63]]  
[[14 34 66 84]  
 [18 47 71 97]  
 [63 68 73 93]]
```

```
M = numpy.random.randint(100, size=(3, 4))  
print M
```

```
S = numpy.sort(M, 0)  
print S
```

```
[[28 42 13 87]  
 [49 17  4 78]  
 [35 61 60 95]]  
[[28 17  4 78]  
 [35 42 13 87]  
 [49 61 60 95]]
```

```
M = numpy.random.randint(100, size=(3, 4))  
print M
```

```
S = numpy.sort(M, 1)  
print S
```

```
[[65 88 77 42]  
 [72 61 52 50]  
 [44 85 34  8]]  
[[42 65 77 88]  
 [50 52 61 72]  
 [ 8 34 44 85]]
```

# Opérations de base

- **`numpy.sum(M)`, `numpy.sum(M, axis=0/1)` ou `numpy.sum(M, 0/1)`**

Somme de toutes les valeurs de la matrice M.

Si **axis = 0**, retourne une matrice 1 colonne et autant de lignes que la matrice M a de colonnes. Chaque valeur correspond à la somme de chaque valeur sur une colonne de la matrice M. Si **axis = 1**, somme sur ligne.

- **`numpy.prod(M)`, `numpy.prod(M, axis=0/1)` ou `numpy.prod(M, 0/1)`**

Produit, mêmes règles que pour `numpy.sum()`.

- **`numpy.cumsum(M)`, `numpy.cumsum(M, axis=0/1)` ou `numpy.cumsum(M, 0/1)`**

Somme cumulée, mêmes règles que pour `numpy.sum()`.

- **`numpy.mean(M)`, `numpy.mean(M, axis=0/1)` ou `numpy.mean(M, 0/1)`**

Moyenne, mêmes règles que pour `numpy.sum()`.

Attention !  
Vague d'exemples dans les diapos qui  
suivent !

```
M = numpy.random.randint(100, size=(2, 4))  
print M
```

```
S = numpy.sum(M)  
print S
```

```
[[68 38 17 22]  
 [34 52 11 96]]  
338
```

```
M = numpy.random.randint(100, size=(2, 4))  
print M
```

```
S = numpy.sum(M, axis=0)  
print S
```

Ou bien

```
S = numpy.sum(M, 0)
```

```
[[43 94 36 29]  
 [ 4 60 93 41]]  
[ 47 154 129 70]
```

```
M = numpy.random.randint(100, size=(2, 4))  
print M
```

```
S = numpy.sum(M, axis=1)  
print S
```

```
[[ 5 71 71 93]  
 [23 28 38 89]]  
[240 178]
```

```
M = numpy.random.randint(100, size=(2, 2))  
print M
```

```
S = numpy.prod(M)  
print S
```

```
[[52 31]  
 [80 73]]  
9414080
```

```
M = numpy.random.randint(100, size=(2, 2))  
print M
```

```
S = numpy.prod(M, axis=0)  
print S
```

```
[[39 63]  
 [28 85]]  
[1092 5355]
```

```
M = numpy.random.randint(100, size=(2, 2))  
print M
```

```
S = numpy.prod(M, axis=1)  
print S
```

```
[[68 90]  
 [95 32]]  
[6120 3040]
```



```
M = numpy.random.randint(100, size=(2, 2))  
print M
```

```
S = numpy.cumsum(M)  
print S
```

```
[[ 0 64]  
 [66 18]]  
[ 0 64 130 148]
```

```
M = numpy.random.randint(100, size=(2, 2))  
print M
```

```
S = numpy.cumsum(M, axis=0)  
print S
```

```
[[38 46]  
 [69 47]]  
[[ 38 46]  
 [107 93]]
```

```
M = numpy.random.randint(100, size=(2, 2))  
print M
```

```
S = numpy.cumsum(M, axis=1)  
print S
```

```
[[70 51]  
 [52 57]]  
[[ 70 121]  
 [ 52 109]]
```

```
M = numpy.random.randint(100, size=(2, 2))  
print M
```

```
S = numpy.mean(M)  
print S
```

```
[[99 59]  
 [ 3 12]]  
43.25
```

```
M = numpy.random.randint(100, size=(2, 2))  
print M
```

```
S = numpy.mean(M, axis=0)  
print S
```

```
[[42 40]  
 [ 1 93]]  
[ 21.5  66.5]
```

```
M = numpy.random.randint(100, size=(2, 2))  
print M
```

```
S = numpy.mean(M, axis=1)  
print S
```

```
[[51 53]  
 [46 11]]  
[ 52.   28.5]
```

# Fusionner des matrices

- **`numpy.hstack((a, b))`**

Retourne une matrice dont les colonnes de la matrice b sont à la suite colonnes de la matrice a.

- **`numpy.vstack((a, b))`**

Retourne une matrice dont les lignes de la matrice b sont sous les lignes de la matrice a.

# Example

```
a = numpy.array([1, 2, 3])  
b = numpy.array([4, 5, 6])  
  
H = numpy.hstack((a, b))  
V = numpy.vstack((a, b))
```

```
print H  
print V
```

```
[1 2 3 4 5 6]  
[[1 2 3]  
 [4 5 6]]
```

# Diviser des matrices

- **`numpy.hsplit(M, x)` ou `numpy.split(M, x, axis=1)`**

Divise la matrice M en x matrices, de répartition équitable. Découpage « vertical » de la matrice M. x doit être un multiple du nombre de colonnes de M.

- **`numpy.vsplit(M, y)` ou `numpy.split(M, y, axis=0)`**

Divise la matrice M en y matrices, de répartition équitable. Découpage « horizontal » de la matrice M. y doit être un multiple du nombre de lignes de M.

Il est possible de préciser où les coupures doivent se situer. Pour cela, il suffit de passer à x ou y des listes d'indices [a, b, c, ...]. Par exemple, **[2, 3]** divisera la matrice de la façon suivante : les lignes ou colonnes d'indice 0 et 1 seront ensemble ([0:2]), la ligne ou colonne d'indice ([2:3]) seule, et enfin toutes les lignes ou colonnes qui sont d'indice 3 et plus ensemble ([3:]).

Attention !

Encore une vague d'exemples dans les  
diapos qui suivent !

```
M = numpy.random.randint(10, size=(4,4))  
print M
```

```
H = numpy.hsplit(M, 2)  
print H
```

Ou bien

```
H = numpy.split(M, 2, axis=1)
```

```
[[9 6 2 7]  
 [9 6 0 7]  
 [8 2 3 4]  
 [3 7 9 1]]  
[array([[9, 6],  
        [9, 6],  
        [8, 2],  
        [3, 7]]), array([[2, 7],  
        [0, 7],  
        [3, 4],  
        [9, 1]])]
```

```
M = numpy.random.randint(10, size=(4,4))  
print M
```

```
V = numpy.vsplit(M, 4)  
print V
```

Ou bien

```
V = numpy.split(M, 4, axis=0)
```

```
[[7 2 4 9]  
 [3 2 2 3]  
 [1 4 2 1]  
 [9 6 9 2]]  
[array([[7, 2, 4, 9]]), array([[3, 2, 2, 3]]), array([[1, 4, 2, 1]]), array([[9, 6, 9, 2]])]
```

```
M = numpy.random.randint(10, size=(3,7))  
print M
```

```
H = numpy.hsplit(M, [2, 4, 6])  
print H
```

```
[[3 8 1 3 4 1 0]  
 [4 5 3 5 6 7 1]  
 [0 0 5 1 9 9 5]]  
[array([[3, 8],  
        [4, 5],  
        [0, 0]]), array([[1, 3],  
        [3, 5],  
        [5, 1]]), array([[4, 1],  
        [6, 7],  
        [9, 9]]), array([[0],  
        [1],  
        [5]])]
```



```
M = numpy.random.randint(10, size=(8,7))
```

```
print M
```

```
V = numpy.vsplit(M, [2, 4, 5])
```

```
print V
```

```
[[3 4 4 9 6 5 0]
 [0 2 5 3 3 2 0]
 [4 2 5 9 4 0 9]
 [7 1 0 6 6 9 6]
 [0 8 6 7 3 0 1]
 [7 4 9 0 6 6 9]
 [1 9 4 4 6 7 5]
 [6 9 7 3 1 7 0]]
```

```
[array([[3, 4, 4, 9, 6, 5, 0],
        [0, 2, 5, 3, 3, 2, 0]]), array([[4, 2, 5, 9, 4, 0, 9],
        [7, 1, 0, 6, 6, 9, 6]]), array([[0, 8, 6, 7, 3, 0, 1]]), array([[7, 4, 9, 0, 6, 6, 9],
        [1, 9, 4, 4, 6, 7, 5],
        [6, 9, 7, 3, 1, 7, 0]])]
```

```
M = numpy.random.randint(10, size=(4, 4))
```

```
print M
```

```
V = numpy.vsplit(M, 3)
```

```
print V
```

```
[[7 5 4 8]
 [7 6 8 5]
 [6 7 0 0]
 [4 9 7 3]]
```

```
Traceback (most recent call last):
```

```
File "test.py", line 9, in <module>
```

```
V = numpy.vsplit(M, 3)
```

```
File "/usr/lib/python2.7/dist-packages/numpy/lib/shape_base.py", line 591, in vsplit
```

```
return split(ary, indices_or_sections, 0)
```

```
File "/usr/lib/python2.7/dist-packages/numpy/lib/shape_base.py", line 476, in split
```

```
raise ValueError('array split does not result in an equal division')
```

```
ValueError: array split does not result in an equal division
```

# Where

- **numpy.where(condition, ...)**

Cette fonction est assez compliquée à expliquer. Un exemple serait plus simple.

Soit deux matrices M1 et M2.

- **numpy.where(M1>20, 0, 1)**

Retourne une matrice de même dimension que M1 avec des 0 et des 1. Les 0 signifient que la condition **M1>20** est vérifiée à l'emplacement du 0, et le 1, le contraire.

- **numpy.where(M1<M2, 0, 1)**

Même chose, mais comparaison de deux matrices (même dimension !).

Remarque : vous pouvez directement écrire M1>20 ou M1<M2 par exemple dans votre script. L'élément retournée sera une matrice de même dimension que M1 (et M2), mais avec le booléen **True** si la condition est vérifiée et **False** si elle ne l'est pas.

# Examples

```
M1 = numpy.random.randint(100, size=(2, 2))  
print M1
```

```
W = numpy.where(M1>20, 0, 1)  
print W
```

```
[[62 42]  
 [34 16]]  
[[0 0]  
 [0 1]]
```

```
M1 = numpy.random.randint(100, size=(2, 2))  
M2 = numpy.random.randint(100, size=(2, 2))  
print M1  
print M2
```

```
W = numpy.where(M1<M2, 0, 1)  
print W
```

```
[[61 26]  
 [ 5  7]]  
[[28 34]  
 [30 38]]  
[[1 0]  
 [0 0]]
```

```
M1 = numpy.random.randint(100, size=(2, 2))
M2 = numpy.random.randint(100, size=(2, 2))
print M1
print M2
```

```
print M1>20
print M1<M2
```

```
[[47 81]
 [ 2 95]]
[[47 38]
 [49 93]]
[[ True  True]
 [False  True]]
[[False False]
 [ True False]]
```

```
M1 = numpy.random.randint(100, size=(2, 2))
M2 = numpy.random.randint(100, size=(2, 3))
print M1
print M2
```

```
W = numpy.where(M1>M2, 0, 1)
print W
```

```
[[84 17]
 [75 83]]
[[98 29 91]
 [72 55 95]]
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    W = numpy.where(M1>M2, 0, 1)
ValueError: operands could not be broadcast together with shapes (2,2) (2,3)
```

# Algèbre linéaire

Qui dit matrice dit algèbre linéaire. Pour cela, numpy possède un sous-module qui se nomme « **linalg** », et contient plusieurs fonctions de calculs applicables aux matrices.

Par la suite, vous pouvez soit taper « **numpy.linalg.fonction()** », ou bien abréger cela tout au début lors de l'importation, avec le mot que vous souhaitez.

- Élévation à la puissance  $n$
- Déterminant et trace
- Valeurs et vecteurs propres
- Résolution d'équations linéaires

# Élévation à la puissance $n$

Pour élever une matrice  $M$  à la puissance  $n$ , utiliser la commande suivante :

**`numpy.linalg.matrix_power(M, n)`**



# Exemple

```
M = numpy.array([[2, 3], [3, 3]])  
print M  
  
P = numpy.linalg.matrix_power(M, 2)  
print P
```

```
[[2 3]  
 [3 3]]  
[[13 15]  
 [15 18]]
```

# Déterminant et trace

- Déterminant

**`numpy.linalg.det(M)`**

- Trace

**`numpy.trace(M)`**

# Exemple

```
M = numpy.array([[2, 3], [3, 3]])  
print M  
  
D = numpy.linalg.det(M)  
T = numpy.trace(M)  
  
print D  
print T
```

```
[[2 3]  
 [3 3]]  
-3.0  
5
```

# Valeurs et vecteurs propres

- `numpy.linalg.eig(M)`

Retourne un tuple de deux matrices :

- La première matrice comprend les valeurs propres de la matrice  $M$
- La seconde, les vecteurs propres associées aux valeurs propres (valeur propre d'indice  $i$  = vecteur propre colonne d'indice  $i$ ).  
Attention ! Un vecteur = 1 colonne !

Pour avoir les vecteurs propres en ligne, il suffit de transposer la matrice des vecteurs propres.

Les valeurs sont normalisées.

# Example

```
M = numpy.array([[2, 3], [8, 3]])  
print M
```

```
V = numpy.linalg.eig(M)  
print V
```

```
print V[1].T
```

```
[[2 3]  
 [8 3]]  
(array([-2.4244289,  7.4244289]), array([[ -0.56120793, -0.483969  ],  
      [ 0.82767485, -0.87508514]]))  
[[ -0.56120793  0.82767485]  
 [ -0.483969  -0.87508514]]
```

# Résolution d'équations linéaires

Soit le système d'équations linéaires suivant :

$$\begin{cases} 3x - y = 1 \\ 2x + 3y = 19 \end{cases}$$

En écriture matricielle, cela donne :

$$\begin{bmatrix} 3 & -1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 19 \end{bmatrix}$$

Soit les matrices a et b suivantes :

$$a = \begin{bmatrix} 3 & -1 \\ 2 & 3 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 19 \end{bmatrix}$$

Pour résoudre le système de la diapo précédente, il est possible d'utiliser la fonction **numpy.linalg.solve()** afin de trouver x et y comme suit :

**numpy.linalg.solve(a, b)**

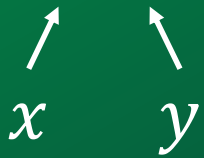
Cette fonction retourne une matrice 1 colonne et autant de lignes que d'inconnues. Les valeurs de cette matrice correspondent aux valeurs inconnues recherchées.

# Exemple

```
a = numpy.array([[3, -1], [2, 3]])  
b = numpy.array([1, 19])  
  
S = numpy.linalg.solve(a, b)  
print S
```

```
[ 2.  5.]
```

$x$     $y$





## Exemple avec un système de 3 équations

Soit le système d'équations linéaires suivant :

$$\begin{cases} 3x + 5y - 7z = -41 \\ 2x - 2y + z = 27 \\ 5x - 20y + 2z = 171 \end{cases}$$

En écriture matricielle, cela donne :

$$\begin{bmatrix} 3 & 5 & -7 \\ 2 & -2 & 1 \\ 5 & -20 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -41 \\ 27 \\ 171 \end{bmatrix}$$

Soit les matrices a et b suivantes :

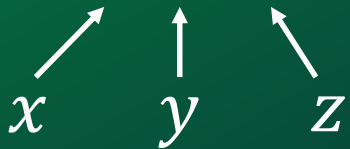
$$a = \begin{bmatrix} 3 & 5 & -7 \\ 2 & -2 & 1 \\ 5 & -20 & 2 \end{bmatrix} \quad b = \begin{bmatrix} -47 \\ 27 \\ 171 \end{bmatrix}$$

Résolution :

```
a = numpy.array([[3, 5, -7], [2, -2, 1], [5, -20, 2]])  
b = numpy.array([-47, 27, 171])  
  
S = numpy.linalg.solve(a, b)  
print S
```

```
[ 5. -7.  3.]
```

$x$     $y$     $z$



# Lien avec matplotlib

Numpy et matplotlib (qui est un module permettant de tracer des graphes) forment ensemble un module nommé **pylab**. Un troisième module fait également parti de ce grand module qui est **scipy**. Toutefois ce dernier ressemble fortement à numpy, et est donc souvent mis de côté.

Si vous utilisez numpy et matplotlib à la fois, vous pouvez importer pylab à la place de faire deux importations différentes, mais cela est plutôt déconseillé. En effet, dans plusieurs cas, des fonctions identiques de nom peuvent se trouver dans deux modules différents. Lorsque vous appelez cette fonction, vous ne savez pas de quelle module la fonction a été prise. D'un module à un autre, une fonction identique de nom peut donner des résultats différents (1 chiffre en plus par exemple, ou en moins).

C'est pour cela qu'il est conseillé d'importer les modules numpy et matplotlib séparément, mais libre à vous de faire ce que bon vous semble.

Vous savez maintenant comment manipuler (presque parfaitement) les matrices !

Tous les secrets de numpy n'ont pas tous été dévoilés, mais celles présentées ici sont les plus courantes.

Libre à vous de vous documenter si vous voulez devenir un as de numpy et ses matrices !

<http://www.numpy.org/>

# Exercices

1) Créer une matrice identité de dimensions de votre choix, puis placer des valeurs aléatoires à la place des 0.

2) Soit la matrice suivante :

$$\begin{bmatrix} 11 & 12 & 15 \\ 10 & 11 & 32 \\ 40 & 18 & 10 \end{bmatrix}$$

Quel(s) est/sont l'/les indice(s) de la/des valeur(s) minimale(s) ? Maximale(s) ?

3) Diviser la matrice en laissant la première ligne seule et les deux autres lignes ensemble.

4) Calculer la somme cumulée (par ligne) de la matrice de la question 2), ainsi que la moyenne sur l'ensemble des valeurs de la matrice. Optionnel : même chose avec la matrice de la question suivante.

5) Soit la matrice suivante :

$$\begin{bmatrix} 11 & 20 & 18 & 98 \\ 13 & 54 & 34 & 1 \\ 17 & 39 & 31 & 58 \\ 24 & 71 & 70 & 87 \end{bmatrix}$$

Placer les deux colonnes les plus à droite, à gauche des deux colonnes les plus à gauche. Faire de même avec les lignes.

6) Calculer le déterminant et la trace de la matrice suivante :

$$\begin{bmatrix} 35 & 40 \\ 78 & 98 \end{bmatrix}$$

7) Déterminer les valeurs et vecteurs propres de la matrice de la question 6).

8) Résoudre les systèmes d'équations suivants :

- $$\begin{cases} x + y = 3 \\ 2x + 5y = 12 \end{cases}$$

- $$\begin{cases} 2x + y + 4z = 16 \\ 4x - 2y + 3z = 9 \\ 8x + 3y + 2z = 20 \end{cases}$$