

Numpy (tableaux de données multi-dimensionnels) et matplotlib (visualisation en 2D et 3D pour Python)

Slim Essid : slim.essid@telecom-paristech.fr

Alexandre Gramfort : alexandre.gramfort@telecom-paristech.fr

adapté du travail de J.R. Johansson (robert@riken.jp) <http://dml.riken.jp/~rob/> (<http://dml.riken.jp/~rob/>)

```
In [1]: # et JUSTE POUR MOI (pour avoir les figures dans le notebook)
        %matplotlib inline
```

Introduction

- numpy est un module utilisé dans presque tous les projets de calcul numérique sous Python
 - Il fournit des structures de données performantes pour la manipulation de vecteurs, matrices et tenseurs plus généraux
 - numpy est écrit en C et en Fortran d'où ses performances élevées lorsque les calculs sont vectorisés (formulés comme des opérations sur des vecteurs/matrices)
- matplotlib est un module performant pour la génération de graphiques en 2D et 3D
 - syntaxe très proche de celle de Matlab
 - supporte texte et étiquettes en $AT&E$ X
 - sortie de qualité dans divers formats (PNG, PDF, SV, EPS...)
 - interface graphique interactive pour explorer les figures

Pour utiliser numpy et matplotlib il faut commencer par les importer :

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
```

On peut plus simplement faire :

```
In [3]: from numpy import *
        from matplotlib.pyplot import *
```

Arrays numpy

Dans la terminologie numpy, vecteurs, matrices et autres tenseurs sont appelés *arrays*.

Création d'arrays numpy

Plusieurs possibilités:

- a partir de listes ou n-uplets Python
- en utilisant des fonctions dédiées, telles que `arange`, `linspace`, etc.
- par chargement à partir de fichiers

A partir de listes

Au moyen de la fonction `numpy.array` :

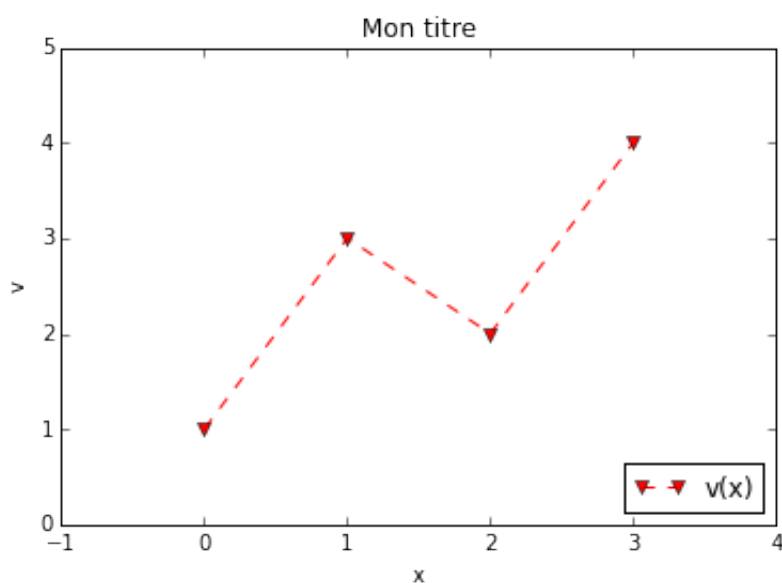
```
In [4]: # un vecteur: l'argument de la fonction est une liste Python
v = np.array([1, 3, 2, 4])
print v
print type(v)

[1 3 2 4]
<type 'numpy.ndarray'>
```

On peut alors visualiser ces données :

```
In [5]: v = np.array([1, 3, 2, 4])
x = np.array([0, 1, 2, 3])

plt.figure()
plt.plot(x,v, 'rv--', label='v(x)')
plt.legend(loc='lower right')
plt.xlabel('x')
plt.ylabel('v')
plt.title('Mon titre')
plt.xlim([-1, 4])
plt.ylim([0, 5])
plt.show()
plt.savefig('toto.png')
```



<matplotlib.figure.Figure at 0x1105f3310>

On peut omettre `show()`, lorsque la méthode `ion()` a été appelée ; c'est le cas dans Spyder et pylab

Pour définir une matrice (array de dimension 2 pour numpy):

```
In [14]: # une matrice: l'argument est une liste emboîtée
M = np.array([[1, 2], [3, 4]])
print M
```

```
[[1 2]
 [3 4]]
```

```
In [15]: M[0, 0]
```

```
Out[15]: 1
```

Les objets `v` et `M` sont tous deux du type `ndarray` (fourni par `numpy`)

```
In [16]: type(v), type(M)
```

```
Out[16]: (numpy.ndarray, numpy.ndarray)
```

`v` et `M` ne diffèrent que par leur taille, que l'on peut obtenir via la propriété `shape` :

```
In [17]: v.shape
```

```
Out[17]: (5, 1)
```

```
In [18]: M.shape
```

```
Out[18]: (2, 2)
```

Pour obtenir le nombre d'éléments d'un *array* :

```
In [19]: v.size
```

```
Out[19]: 5
```

```
In [20]: M.size
```

```
Out[20]: 4
```

On peut aussi utiliser `numpy.shape` et `numpy.size`

```
In [21]: np.shape(M)
```

```
Out[21]: (2, 2)
```

Les *arrays* ont un type qu'on obtient via `dtype`:

```
In [22]: print(M)
         print(M.dtype)
```

```
[[1 2]
 [3 4]]
int64
```

Les types doivent être respectés lors d'assignations à des *arrays*

```
In [23]: M[0,0] = "hello"
```

```
-----
-----
ValueError                                Traceback (most recent call
last)
<ipython-input-23-a09d72434238> in <module>()
----> 1 M[0,0] = "hello"

ValueError: invalid literal for long() with base 10: 'hello'
```

Attention !

```
In [24]: a = np.array([1,2,3])
a[0] = 3.2
print a
a.dtype
```

```
[3 2 3]
```

```
Out[24]: dtype('int64')
```

```
In [25]: a = np.array([1,2,3], dtype=np.int64)
b = np.array([2,2,3], dtype=np.int64)
b = b.astype(float)
print a / b
```

```
[ 0.5  1.   1. ]
```

On peut définir le type de manière explicite en utilisant le mot clé `dtype` en argument:

```
In [26]: M = np.array([[1, 2], [3, 4]], dtype=complex)
M
```

```
Out[26]: array([[ 1.+0.j,  2.+0.j],
                [ 3.+0.j,  4.+0.j]])
```

- Autres types possibles avec `dtype` : `int`, `float`, `complex`, `bool`, `object`, etc.
- On peut aussi spécifier la précision en bits: `int64`, `int16`, `float128`, `complex128`.

Utilisation de fonction de génération d'arrays

arange

```
In [27]: # create a range
x = np.arange(0, 10, 2) # arguments: start, stop, step
x
```

```
Out[27]: array([0, 2, 4, 6, 8])
```

```
In [28]: x = np.arange(-1, 1, 0.1)
x
```

```
Out[28]: array([ -1.00000000e+00,  -9.00000000e-01,  -8.00000000e-01,
 -7.00000000e-01,  -6.00000000e-01,  -5.00000000e-01,
 -4.00000000e-01,  -3.00000000e-01,  -2.00000000e-01,
 -1.00000000e-01,  -2.22044605e-16,   1.00000000e-01,
  2.00000000e-01,   3.00000000e-01,   4.00000000e-01,
  5.00000000e-01,   6.00000000e-01,   7.00000000e-01,
  8.00000000e-01,   9.00000000e-01])
```

linspace and logspace

```
In [29]: # avec linspace, le début et la fin SONT inclus
np.linspace(0, 10, 25)
```

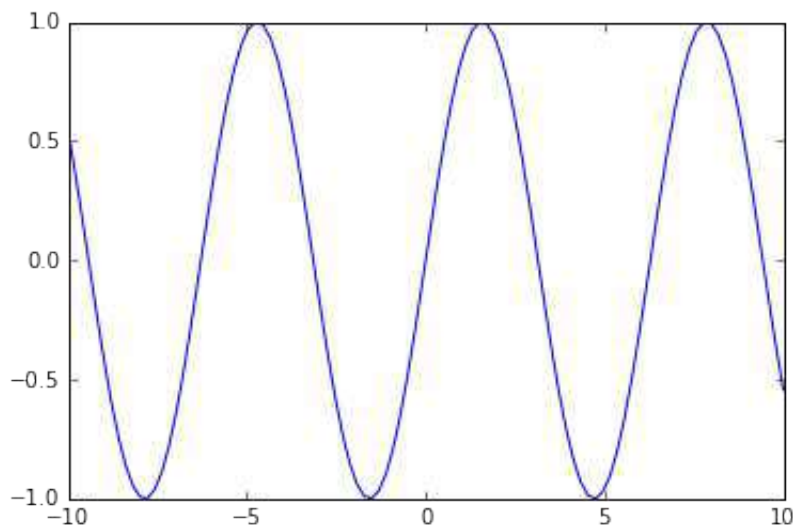
```
Out[29]: array([ 0.         ,  0.41666667,  0.83333333,  1.25        ,
 1.66666667,  2.08333333,  2.5         ,  2.91666667,
 3.33333333,  3.75        ,  4.16666667,  4.58333333,
 5.         ,  5.41666667,  5.83333333,  6.25        ,
 6.66666667,  7.08333333,  7.5         ,  7.91666667,
 8.33333333,  8.75        ,  9.16666667,  9.58333333, 10.
])
```

```
In [30]: np.linspace(0, 10, 11)
```

```
Out[30]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.,
 10.] )
```

```
In [31]: xx = np.linspace(-10, 10, 100)
plt.plot(xx, np.sin(xx))
```

```
Out[31]: [<matplotlib.lines.Line2D at 0x110d96d10>]
```



```
In [33]: print(np.logspace(0, 10, 10, base=np.e))
```

```
[ 1.00000000e+00  3.03773178e+00  9.22781435e+00  2.80316249e+01
 8.51525577e+01  2.58670631e+02  7.85771994e+02  2.38696456e+03
 7.25095809e+03  2.20264658e+04]
```

mgrid

```
In [34]: x, y = np.mgrid[0:5, 0:5]
```

```
In [35]: x
```

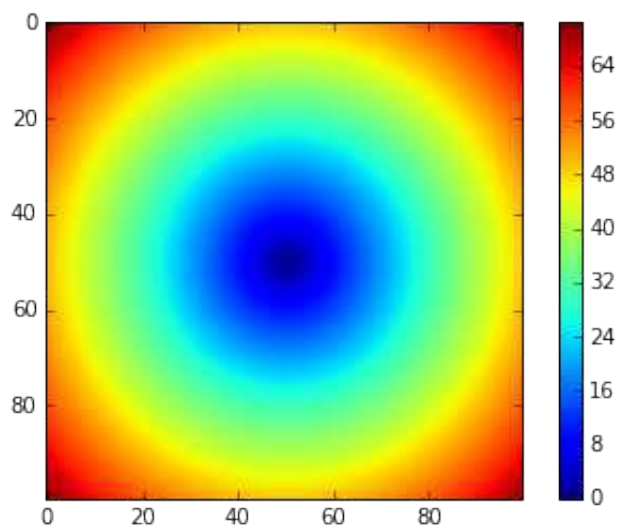
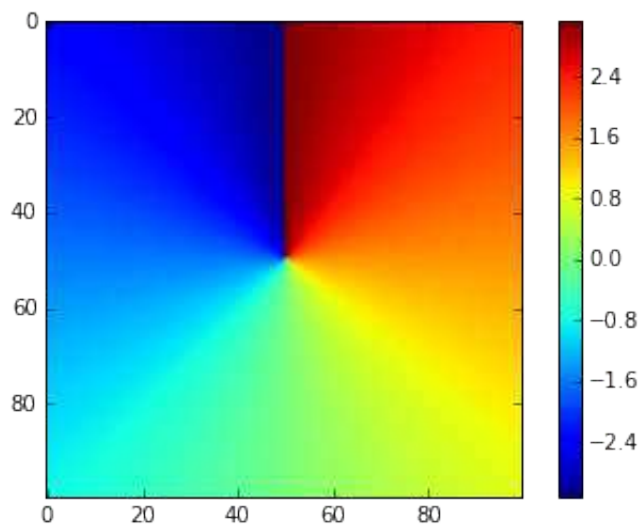
```
Out[35]: array([[0, 0, 0, 0, 0],
                [1, 1, 1, 1, 1],
                [2, 2, 2, 2, 2],
                [3, 3, 3, 3, 3],
                [4, 4, 4, 4, 4]])
```

```
In [36]: y
```

```
Out[36]: array([[0, 1, 2, 3, 4],
                [0, 1, 2, 3, 4],
                [0, 1, 2, 3, 4],
                [0, 1, 2, 3, 4],
                [0, 1, 2, 3, 4]])
```

```
In [38]: xx, yy = np.mgrid[-50:50, -50:50]
plt.imshow(np.angle(xx + 1j*yy))
plt.axis('on')
plt.colorbar()
plt.figure()
plt.imshow(np.abs(xx + 1j*yy))
plt.axis('on')
plt.colorbar()
```

Out[38]: <matplotlib.colorbar.Colorbar instance at 0x111da4320>



Données aléatoires

```
In [39]: from numpy import random
```



```
In [40]: # tirage uniforme dans [0,1]
random.rand(5,5) # ou np.random.rand
```

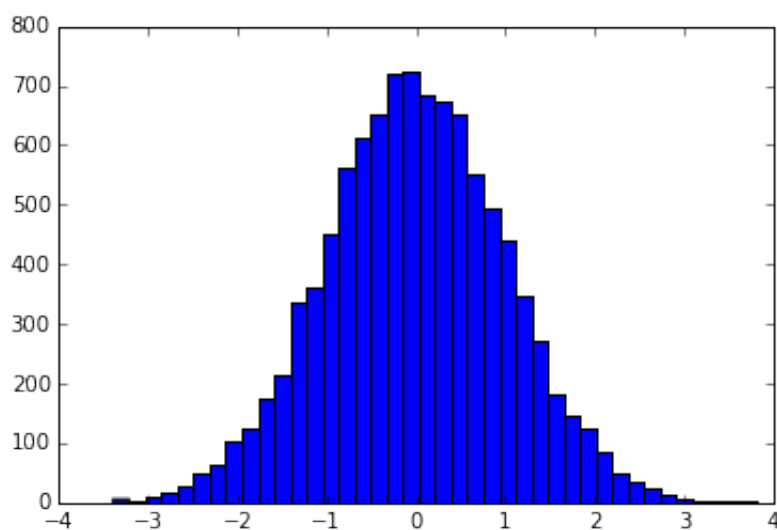
```
Out[40]: array([[ 0.04787772,  0.51323983,  0.9076947 ,  0.02667448,  0.34598
267],
 [ 0.11853056,  0.64242872,  0.31002472,  0.05069742,  0.63790
977],
 [ 0.5173889 ,  0.08046615,  0.68116042,  0.68278034,  0.68172
398],
 [ 0.25957184,  0.80848937,  0.11265106,  0.10062125,  0.86411
504],
 [ 0.18718648,  0.5938647 ,  0.06848848,  0.37451237,  0.23974
323]])
```

```
In [41]: # tirage suivant une loi normale standard
random.randn(5,5)
```

```
Out[41]: array([[ 1.77285213,  1.32680289, -0.2059008 , -1.49880077, -0.98383
064],
 [ 1.14162261,  0.27945307, -0.19274179, -2.32994527, -0.28799
258],
 [ 0.23126061,  0.54948059,  1.33094402,  1.200761 , -0.97121
56 ],
 [ 0.624536 ,  1.56482753, -0.02224487,  1.77068654,  1.05426
645],
 [ 0.88113704, -1.00861277,  0.17409557, -0.76698889, -0.37759
369]])
```

Affichage de l'histogramme des tirages

```
In [42]: a = random.randn(10000)
hh = plt.hist(a, 40)
```



diag

```
In [43]: # une matrice diagonale
np.diag([1,2,3])
```

```
Out[43]: array([[1, 0, 0],
               [0, 2, 0],
               [0, 0, 3]])
```

```
In [44]: # diagonale avec décalage par rapport à la diagonale principale
np.diag([1,2,3], k=1)
```

```
Out[44]: array([[0, 1, 0, 0],
               [0, 0, 2, 0],
               [0, 0, 0, 3],
               [0, 0, 0, 0]])
```

zeros et ones

```
In [45]: np.zeros((3,), dtype=int) # attention zeros(3,3) est FAUX
```

```
Out[45]: array([0, 0, 0])
```

```
In [46]: np.ones((3,3))
```

```
Out[46]: array([[ 1.,  1.,  1.],
               [ 1.,  1.,  1.],
               [ 1.,  1.,  1.]])
```

```
In [48]: print(np.zeros((3,), dtype=int))
print(np.zeros((1, 3), dtype=int))
print(np.zeros((3, 1), dtype=int))
```

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

Fichiers d'E/S

Fichiers séparés par des virgules (CSV)

Un format fichier classique est le format CSV (comma-separated values), ou bien TSV (tab-separated values). Pour lire de tels fichiers utilisez `numpy.genfromtxt`. Par exemple:

```
In [49]: !cat data.csv
```

```
1,2,3,4,5
6,7,8,9,10
1,3,3,4,6
1,2,3,4,20
```

```
In [50]: data = np.genfromtxt('data.csv', delimiter=',')
data
```

```
Out[50]: array([[ 1.,  2.,  3.,  4.,  5.],
                [ 6.,  7.,  8.,  9., 10.],
                [ 1.,  3.,  3.,  4.,  6.],
                [ 1.,  2.,  3.,  4., 20.]])
```

```
In [51]: data.shape
```

```
Out[51]: (4, 5)
```

A l'aide de `numpy.savetxt` on peut enregistrer un *array* numpy dans un fichier txt:

```
In [52]: M = random.rand(3,3)
M
```

```
Out[52]: array([[ 0.39914696,  0.6410838 ,  0.7911557 ],
                [ 0.8893561 ,  0.5710319 ,  0.57829609],
                [ 0.3105546 ,  0.4751752 ,  0.95008291]])
```

```
In [53]: np.savetxt("random-matrix.txt", M)
```

```
In [54]: !cat random-matrix.txt
#!type random-matrix.txt
```

```
3.991469606906968837e-01 6.410837975177376968e-01 7.9115570454630301
80e-01
8.893561001624668005e-01 5.710319003857974307e-01 5.7829609417067529
00e-01
3.105546045101991171e-01 4.751751950756873955e-01 9.5008291247079723
62e-01
```

```
In [55]: np.savetxt("random-matrix.csv", M, fmt='%.5f', delimiter=',') # fmt
         #spécifie le format

         !cat random-matrix.csv
         #!type random-matrix.csv

         0.39915,0.64108,0.79116
         0.88936,0.57103,0.57830
         0.31055,0.47518,0.95008
```

Format de fichier Numpy natif

Pour sauvegarder et recharger des *array* numpy : `numpy.save` et `numpy.load` :

```
In [56]: np.save("random-matrix.npy", M)

         !cat random-matrix.npy

         \NUMPYF{'descr': '<f8', 'fortran_order': False, 'shape': (3, 3), }
         .
         +?%)??%Q??D??u??&$??E??E??{?f???5k ??8Ei??Gg
         ??
```

```
In [57]: np.load("random-matrix.npy")

Out[57]: array([[ 0.39914696,  0.6410838 ,  0.7911557 ],
                [ 0.8893561 ,  0.5710319 ,  0.57829609],
                [ 0.3105546 ,  0.4751752 ,  0.95008291]])
```

Autres propriétés des *arrays* numpy

```
In [58]: M

Out[58]: array([[ 0.39914696,  0.6410838 ,  0.7911557 ],
                [ 0.8893561 ,  0.5710319 ,  0.57829609],
                [ 0.3105546 ,  0.4751752 ,  0.95008291]])
```

```
In [59]: M.dtype

Out[59]: dtype('float64')
```

```
In [60]: M.itemsize # octets par élément

Out[60]: 8
```

```
In [61]: M.nbytes # nombre d'octets
```

```
Out[61]: 72
```

```
In [62]: M.nbytes / M.size
```

```
Out[62]: 8
```

```
In [63]: M.ndim # nombre de dimensions
```

```
Out[63]: 2
```

```
In [65]: print(np.zeros((3,), dtype=int).ndim)
print(np.zeros((1, 3), dtype=int).ndim)
print(np.zeros((3, 1), dtype=int).ndim)
```

```
1
2
2
```

Manipulation d'arrays

Indexation

```
In [66]: # v est un vecteur, il n'a qu'une seule dimension -> un seul indice
v[0]
```

```
Out[66]: array([ 1.])
```

```
In [67]: # M est une matrice, ou un array à 2 dimensions -> deux indices
M[1,1]
```

```
Out[67]: 0.57103190038579743
```

Contenu complet :

```
In [68]: M
```

```
Out[68]: array([[ 0.39914696,  0.6410838 ,  0.7911557 ],
                [ 0.8893561 ,  0.5710319 ,  0.57829609],
                [ 0.3105546 ,  0.4751752 ,  0.95008291]])
```

La deuxième ligne :

```
In [69]: M[1]
```

```
Out[69]: array([ 0.8893561 ,  0.5710319 ,  0.57829609])
```

On peut aussi utiliser :

```
In [70]: M[1,:] # 2 ème ligne (indice 1)
```

```
Out[70]: array([ 0.8893561 ,  0.5710319 ,  0.57829609])
```

```
In [71]: M[:,1] # 2 ème colonne (indice 1)
```

```
Out[71]: array([ 0.6410838,  0.5710319,  0.4751752])
```

```
In [73]: print(M.shape)
print(M[1,:].shape, M[:,1].shape)
```

```
(3, 3)
((3,), (3,))
```

On peut assigner des nouvelles valeurs à certaines cellules :

```
In [74]: M[0,0] = 1
```

```
In [75]: M
```

```
Out[75]: array([[ 1.          ,  0.6410838 ,  0.7911557 ],
                [ 0.8893561 ,  0.5710319 ,  0.57829609],
                [ 0.3105546 ,  0.4751752 ,  0.95008291]])
```

```
In [76]: # on peut aussi assigner des lignes ou des colonnes
M[1,:] = -1
# M[1,:] = [1, 2, 3]
```

```
In [77]: M
```

```
Out[77]: array([[ 1.          ,  0.6410838 ,  0.7911557 ],
                [-1.          , -1.          , -1.          ],
                [ 0.3105546 ,  0.4751752 ,  0.95008291]])
```

Slicing ou accès par tranches

Slicing fait référence à la syntaxe `M[start:stop:step]` pour extraire une partie d'un *array* :

```
In [78]: A = np.array([1,2,3,4,5])  
A
```

```
Out[78]: array([1, 2, 3, 4, 5])
```

```
In [79]: A[1:3]
```

```
Out[79]: array([2, 3])
```

Les tranches sont modifiables :

```
In [80]: A[1:3] = [-2,-3]  
A
```

```
Out[80]: array([ 1, -2, -3,  4,  5])
```

On peut omettre n'importe lequel des argument dans M[start:stop:step]:

```
In [81]: A[:] # indices de début, fin, et pas avec leurs valeurs par défaut
```

```
Out[81]: array([ 1, -2, -3,  4,  5])
```

```
In [82]: A[::2] # pas = 2, indices de début et de fin par défaut
```

```
Out[82]: array([ 1, -3,  5])
```

```
In [83]: A[:3] # les trois premiers éléments
```

```
Out[83]: array([ 1, -2, -3])
```

```
In [84]: A[3:] # à partir de l'indice 3
```

```
Out[84]: array([4, 5])
```

```
In [85]: M = np.arange(12).reshape(4, 3)  
print M
```

```
[[ 0  1  2]  
 [ 3  4  5]  
 [ 6  7  8]  
 [ 9 10 11]]
```

On peut utiliser des indices négatifs :

```
In [86]: A = np.array([1,2,3,4,5])
```

```
In [87]: A[-1] # le dernier élément
```

```
Out[87]: 5
```

```
In [88]: A[-3:] # les 3 derniers éléments
```

```
Out[88]: array([3, 4, 5])
```

Le *slicing* fonctionne de façon similaire pour les *array* multi-dimensionnels

```
In [89]: A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
```

```
A
```

```
Out[89]: array([[ 0,  1,  2,  3,  4],
               [10, 11, 12, 13, 14],
               [20, 21, 22, 23, 24],
               [30, 31, 32, 33, 34],
               [40, 41, 42, 43, 44]])
```

```
In [90]: A[1:4, 1:4] # sous-tableau
```

```
Out[90]: array([[11, 12, 13],
               [21, 22, 23],
               [31, 32, 33]])
```

```
In [91]: # sauts
```

```
A[::2, ::2]
```

```
Out[91]: array([[ 0,  2,  4],
               [20, 22, 24],
               [40, 42, 44]])
```

```
In [92]: A
```

```
Out[92]: array([[ 0,  1,  2,  3,  4],
               [10, 11, 12, 13, 14],
               [20, 21, 22, 23, 24],
               [30, 31, 32, 33, 34],
               [40, 41, 42, 43, 44]])
```

```
In [93]: A[[0, 1, 3]]
```

```
Out[93]: array([[ 0,  1,  2,  3,  4],
               [10, 11, 12, 13, 14],
               [30, 31, 32, 33, 34]])
```


Indexation avancée (*fancy indexing*)

Lorsque qu'on utilise des listes ou des *array* pour définir des tranches :

```
In [94]: row_indices = [1, 2, 3]
         print(A)
         print(A[row_indices])
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]
[[10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]]
```

```
In [95]: A[[1, 2]][[:, [3, 4]]] = 0 # ATTENTION !
         print(A)
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]
```

```
In [96]: print(A[[1, 2], [3, 4]])
```

```
[13 24]
```

```
In [97]: A[np.ix_([1, 2], [3, 4])] = 0
         print(A)
```

```
[[ 0  1  2  3  4]
 [10 11 12  0  0]
 [20 21 22  0  0]
 [30 31 32 33 34]
 [40 41 42 43 44]]
```

On peut aussi utiliser des masques binaires :

```
In [98]: B = np.arange(5)
         B
```

```
Out[98]: array([0, 1, 2, 3, 4])
```

```
In [99]: row_mask = np.array([True, False, True, False, False])
         print(B[row_mask])
         print(B[[0,2]])
```

```
[0 2]
[0 2]
```

```
In [100]: # de façon équivalente
          row_mask = np.array([1,0,1,0,0], dtype=bool)
          B[row_mask]
```

```
Out[100]: array([0, 2])
```

```
In [101]: # ou encore
          a = np.array([1, 2, 3, 4, 5])
          print(a < 3)
          print(B[a < 3])
```

```
[ True  True False False False]
[0 1]
```

```
In [102]: print(A)
          print(A[:, a < 3])
```

```
[[ 0  1  2  3  4]
 [10 11 12  0  0]
 [20 21 22  0  0]
 [30 31 32 33 34]
 [40 41 42 43 44]]
[[ 0  1]
 [10 11]
 [20 21]
 [30 31]
 [40 41]]
```

Extraction de données à partir d'arrays et création d'arrays

where

Un masque binaire peut être converti en indices de positions avec where

```
In [103]: x = np.arange(0, 10, 0.5)
          print(x)
          mask = (x > 5) * (x < 7.5)
          print(mask)
          indices = np.where(mask)
          indices

[ 0.  0.5  1.  1.5  2.  2.5  3.  3.5  4.  4.5  5.  5.5  6.  6.5  7.
 7.5  8.  8.5  9.  9.5]
[False False False False False False False False False False False
 True
  True  True  True False False False False False]
```

```
Out[103]: (array([11, 12, 13, 14]),)
```

```
In [104]: x[indices] # équivalent à x[mask]
```

```
Out[104]: array([ 5.5,  6. ,  6.5,  7. ])
```

diag

Extraire la diagonale ou une sous-diagonale d'un *array* :

```
In [105]: print(A)
          np.diag(A)
```

```
[[ 0  1  2  3  4]
 [10 11 12  0  0]
 [20 21 22  0  0]
 [30 31 32 33 34]
 [40 41 42 43 44]]
```

```
Out[105]: array([ 0, 11, 22, 33, 44])
```

```
In [106]: np.diag(A, -1)
```

```
Out[106]: array([10, 21, 32, 43])
```

Algèbre linéaire

La performance des programmes écrit en Python/Numpy dépend de la capacité à vectoriser les calculs (les écrire comme des opérations sur des vecteurs/matrices) en évitant au maximum les boucles `for/while`

Opérations scalaires

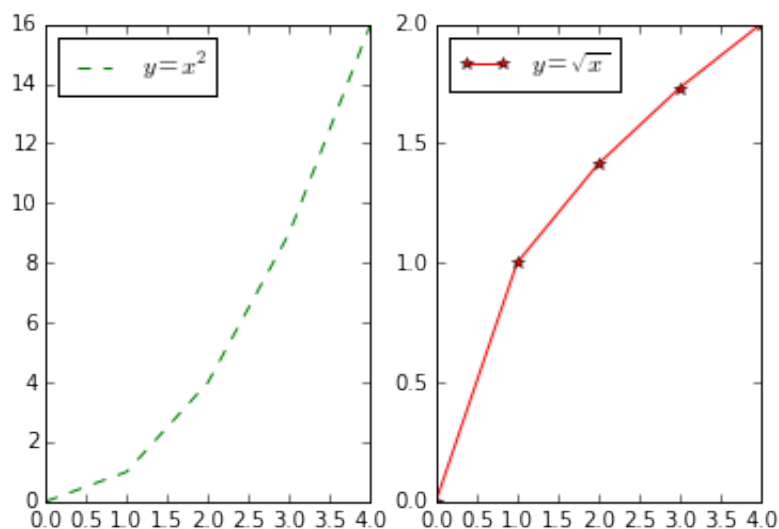
On peut effectuer les opérations arithmétiques habituelles pour multiplier, additionner, soustraire et diviser des *arrays* avec/par des scalaires :

```
In [107]: v1 = np.arange(0, 5)
          print(v1)
          [0 1 2 3 4]
```

```
In [108]: v1 * 2
Out[108]: array([0, 2, 4, 6, 8])
```

```
In [109]: v1 + 2
Out[109]: array([2, 3, 4, 5, 6])
```

```
In [110]: plt.figure()
          plt.subplot(1,2,1)
          plt.plot(v1 ** 2, 'g--', label='$y = x^2$')
          plt.legend(loc=0)
          plt.subplot(1,2,2)
          plt.plot(sqrt(v1), 'r*- ', label='$y = \sqrt{x}$')
          plt.legend(loc=2)
          plt.show()
```



```
In [111]: A = np.array([[n+m*10 for n in range(5)] for m in range(5)])  
print(A)
```

```
[[ 0  1  2  3  4]  
 [10 11 12 13 14]  
 [20 21 22 23 24]  
 [30 31 32 33 34]  
 [40 41 42 43 44]]
```

```
In [112]: print(A * 2)
```

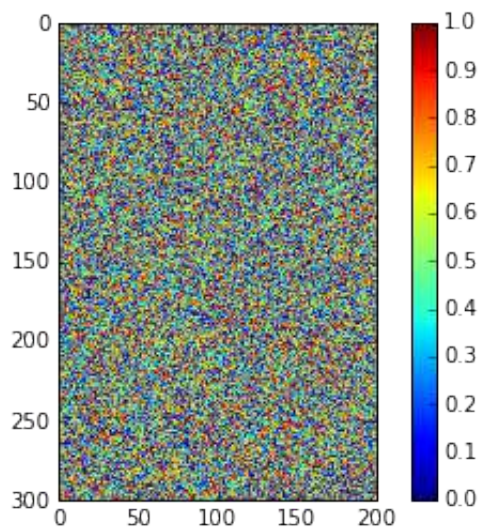
```
[[ 0  2  4  6  8]  
 [20 22 24 26 28]  
 [40 42 44 46 48]  
 [60 62 64 66 68]  
 [80 82 84 86 88]]
```

```
In [113]: print(A + 2)
```

```
[[ 2  3  4  5  6]  
 [12 13 14 15 16]  
 [22 23 24 25 26]  
 [32 33 34 35 36]  
 [42 43 44 45 46]]
```

Visualiser des matrices

```
In [114]: C = random.rand(300,200)  
plt.figure()  
plt.imshow(C)  
plt.colorbar()  
plt.show()
```



Opérations terme-à-terme sur les *arrays*

Les opérations par défaut sont des opérations **terme-à-terme** :

```
In [115]: A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
          print(A)
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]
```

```
In [116]: A * A # multiplication terme-à-terme
```

```
Out[116]: array([[ 0,  1,  4,  9, 16],
                 [100, 121, 144, 169, 196],
                 [ 400, 441, 484, 529, 576],
                 [ 900, 961, 1024, 1089, 1156],
                 [1600, 1681, 1764, 1849, 1936]])
```

```
In [117]: (A + A.T) / 2
```

```
Out[117]: array([[ 0,  5, 11, 16, 22],
                 [ 5, 11, 16, 22, 27],
                 [11, 16, 22, 27, 33],
                 [16, 22, 27, 33, 38],
                 [22, 27, 33, 38, 44]])
```

```
In [118]: print v1
          print v1 * v1
```

```
[0 1 2 3 4]
[ 0  1  4  9 16]
```

En multipliant des *arrays* de tailles compatibles, on obtient des multiplications terme-à-terme par ligne :

```
In [120]: A.shape, v1.shape
```

```
Out[120]: ((5, 5), (5,))
```

```
In [121]: print(A)
          print(v1)
          print(A * v1)

[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]
[0 1 2 3 4]
[[ 0  1  4  9 16]
 [ 0 11 24 39 56]
 [ 0 21 44 69 96]
 [ 0 31 64 99 136]
 [ 0 41 84 129 176]]
```

Exercice:

Sans utiliser de boucles (for/while) :

- Créer une matrice (5x6) aléatoire
- Remplacer une colonne sur deux par sa valeur moins le double de la colonne suivante
- Remplacer les valeurs négatives par 0 en utilisant un masque binaire

Algèbre matricielle

Comment faire des multiplications de matrices ? Deux façons :

- en utilisant les fonctions dot; (recommandé)
- en utiliser le type matrix. (à éviter)

```
In [122]: print(A.shape)
          print(A)
          print(type(A))

(5, 5)
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]
<type 'numpy.ndarray'>
```

```
In [123]: print(np.dot(A, A)) # multiplication matrice
          print(A * A) # multiplication élément par élément

[[ 300  310  320  330  340]
 [1300 1360 1420 1480 1540]
 [2300 2410 2520 2630 2740]
 [3300 3460 3620 3780 3940]
 [4300 4510 4720 4930 5140]]
[[   0   1   4   9  16]
 [ 100  121  144  169  196]
 [ 400  441  484  529  576]
 [ 900  961 1024 1089 1156]
 [1600 1681 1764 1849 1936]]
```

```
In [124]: A.dot(v1)
```

```
Out[124]: array([ 30, 130, 230, 330, 430])
```

```
In [125]: np.dot(v1, v1)
```

```
Out[125]: 30
```

Avec le type `matrix` de Numpy

```
In [126]: M = np.matrix(A)
          v = np.matrix(v1).T # en faire un vecteur colonne
```

```
In [127]: M * v
```

```
Out[127]: matrix([[ 30],
                  [130],
                  [230],
                  [330],
                  [430]])
```

```
In [128]: # produit scalaire
          v.T * v
```

```
Out[128]: matrix([[30]])
```

```
In [129]: # avec les objets matrices, c'est les opérations standards sur les
          matrices qui sont appliquées
          v + M*v
```

```
Out[129]: matrix([[ 30],
                  [131],
                  [232],
                  [333],
                  [434]])
```


Si les dimensions sont incompatibles on provoque des erreurs :

```
In [130]: v = np.matrix([1,2,3,4,5,6]).T
```

```
In [131]: np.shape(M), np.shape(v)
```

```
Out[131]: ((5, 5), (6, 1))
```

```
In [132]: M * v
```

```
-----
-----
ValueError                                Traceback (most recent call
last)
<ipython-input-132-995fb48ad0cc> in <module>()
----> 1 M * v

/Users/alex/anaconda/lib/python2.7/site-packages/numpy/matrixlib/def
matrix.pyc in __mul__(self, other)
    341     if isinstance(other, (N.ndarray, list, tuple)) :
    342         # This promotes 1-D vectors to row vectors
--> 343         return N.dot(self, asmatrix(other))
    344     if isscalar(other) or not hasattr(other, '__rmul__')
:
    345         return N.dot(self, other)

ValueError: shapes (5,5) and (6,1) not aligned: 5 (dim 1) != 6 (dim
0)
```

Voir également les fonctions : `inner`, `outer`, `cross`, `kron`, `tensordot`. Utiliser par exemple `help(kron)`.

Transformations d'arrays ou de matrices

- Plus haut `.T` a été utilisé pour transposer l'objet matrice `v`
- On peut aussi utiliser la fonction `transpose`

Autres transformations :

```
In [133]: C = np.matrix([[1j, 2j], [3j, 4j]])
          C
```

```
Out[133]: matrix([[ 0.+1.j,  0.+2.j],
                  [ 0.+3.j,  0.+4.j]])
```

```
In [134]: np.conjugate(C)
```

```
Out[134]: matrix([[ 0.-1.j,  0.-2.j],
                  [ 0.-3.j,  0.-4.j]])
```

Transposée conjuguée :

```
In [135]: C.H
```

```
Out[135]: matrix([[ 0.-1.j,  0.-3.j],
                  [ 0.-2.j,  0.-4.j]])
```

Parties réelles et imaginaires :

```
In [136]: np.real(C) # same as: C.real
```

```
Out[136]: matrix([[ 0.,  0.],
                  [ 0.,  0.]])
```

```
In [137]: np.imag(C) # same as: C.imag
```

```
Out[137]: matrix([[ 1.,  2.],
                  [ 3.,  4.]])
```

Argument et module :

```
In [138]: np.angle(C + 1)
```

```
Out[138]: array([[ 0.78539816,  1.10714872],
                 [ 1.24904577,  1.32581766]])
```

```
In [139]: np.abs(C)
```

```
Out[139]: matrix([[ 1.,  2.],
                  [ 3.,  4.]])
```

Caclul matriciel

Analyse de données

Numpy propose des fonctions pour calculer certaines statistiques des données stockées dans des *arrays* :

```
In [140]: data = np.vander([1, 2, 3, 4])
          print(data)
          print(data.shape)

[[ 1  1  1  1]
 [ 8  4  2  1]
 [27  9  3  1]
 [64 16  4  1]]
(4, 4)
```

mean

```
In [141]: # np.mean(data)
          print(np.mean(data, axis=0))

[ 25.    7.5    2.5    1. ]
```

```
In [142]: # la moyenne de la troisième colonne
          np.mean(data[:,2])
```

```
Out[142]: 2.5
```

variance et écart type

```
In [143]: np.var(data[:,2]), np.std(data[:,2])
```

```
Out[143]: (1.25, 1.1180339887498949)
```

min et max

```
In [144]: data[:,2].min()
```

```
Out[144]: 1
```

```
In [145]: data[:,2].max()
```

```
Out[145]: 4
```

```
In [146]: data[:,2].sum()
```

```
Out[146]: 10
```

```
In [147]: data[:,2].prod()
```

```
Out[147]: 24
```

sum, prod, et trace

```
In [148]: d = np.arange(0, 10)
          d
```

```
Out[148]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [149]: # somme des éléments
          np.sum(d)
```

```
Out[149]: 45
```

ou encore :

```
In [150]: d.sum()
```

```
Out[150]: 45
```

```
In [151]: # produit des éléments
          np.prod(d + 1)
```

```
Out[151]: 3628800
```

```
In [152]: # somme cumulée
          np.cumsum(d)
```

```
Out[152]: array([ 0,  1,  3,  6, 10, 15, 21, 28, 36, 45])
```

```
In [153]: # produit cumulé
          np.cumprod(d + 1)
```

```
Out[153]: array([ 1,  2,  6, 24, 120, 720, 5040,
                , 40320, 362880, 3628800])
```

```
In [154]: # équivalent à diag(A).sum()
          np.trace(data)
```

```
Out[154]: 9
```

EXERCICE :

Calculer une approximation de π par la formule de Wallis sans boucle `for` avec Numpy

$$\pi = 2 \prod_{i=1}^{\infty} \frac{4i^2}{4i^2 - 1}$$

Calculs avec parties d'arrays

en utilisant l'indexation ou n'importe quelle méthode d'extraction de données à partir des *arrays*

```
In [155]: data
```

```
Out[155]: array([[ 1,  1,  1,  1],
                 [ 8,  4,  2,  1],
                 [27,  9,  3,  1],
                 [64, 16,  4,  1]])
```

```
In [156]: np.unique(data[:,1])
```

```
Out[156]: array([ 1,  4,  9, 16])
```

```
In [157]: mask = data[:,1] == 4
```

```
In [158]: np.mean(data[mask,3])
```

```
Out[158]: 1.0
```

Calculs avec données multi-dimensionnelles

Pour appliquer `min`, `max`, etc., par lignes ou colonnes :

```
In [159]: m = random.rand(3,4)
          m
```

```
Out[159]: array([[ 0.1107825 ,  0.13145548,  0.46566654,  0.2465301 ],
                 [ 0.63008442,  0.95829525,  0.90675777,  0.87015917],
                 [ 0.7042647 ,  0.68990414,  0.95861916,  0.55068237]])
```

```
In [160]: # max global
          m.max()
```

```
Out[160]: 0.95861916063175645
```

```
In [161]: # max dans chaque colonne  
m.max(axis=0)
```

```
Out[161]: array([ 0.7042647 ,  0.95829525,  0.95861916,  0.87015917])
```

```
In [162]: # max dans chaque ligne  
m.max(axis=1)
```

```
Out[162]: array([ 0.46566654,  0.95829525,  0.95861916])
```

Plusieurs autres méthodes des classes `array` et `matrix` acceptent l'argument (optional) `axis` keyword argument.

Copy et "deep copy"

Pour des raisons de performance Python ne copie pas automatiquement les objets (par exemple passage par référence des paramètres de fonctions).

```
In [163]: A = np.array([[0, 2],[ 3, 4]])  
A
```

```
Out[163]: array([[0, 2],  
                [3, 4]])
```

```
In [164]: B = A
```

```
In [165]: # changer B affecte A  
B[0,0] = 10  
B
```

```
Out[165]: array([[10, 2],  
                [ 3, 4]])
```

```
In [166]: A
```

```
Out[166]: array([[10, 2],  
                [ 3, 4]])
```

```
In [167]: B = A  
print B is A
```

```
True
```

Pour éviter ce comportement, on peut demander une *copie profonde* (*deep copy*) de A dans B

```
In [168]: # B = np.copy(A)
          B = A.copy()
```

```
In [169]: # maintenant en modifiant B, A n'est plus affecté
          B[0,0] = -5

          B
```

```
Out[169]: array([[ -5,  2],
                 [ 3,  4]])
```

```
In [170]: A # A est aussi modifié !
```

```
Out[170]: array([[10,  2],
                 [ 3,  4]])
```

```
In [171]: print(A - A[:,0]) # FAUX
          print(A - A[:,0].reshape((2, 1))) # OK

          [[ 0 -1]
           [-7  1]]
          [[ 0 -8]
           [ 0  1]]
```

Changement de forme et de taille, et concaténation des *arrays*

```
In [172]: A
```

```
Out[172]: array([[10,  2],
                 [ 3,  4]])
```

```
In [173]: n, m = A.shape
```

```
In [174]: B = A.reshape((1, n * m))
          B
```

```
Out[174]: array([[10,  2,  3,  4]])
```

```
In [175]: B[0,0:5] = 5 # modifier l'array

          B
```

```
Out[175]: array([[5, 5, 5, 5]])
```

```
In [176]: A
```

```
Out[176]: array([[5, 5],
                 [5, 5]])
```

Attention !

La variable originale est aussi modifiée ! B n'est qu'une nouvelle *vue* de A.

Pour transformer un *array* multi-dimmsionnel en un vecteur. Mais cette fois-ci, une copie des données est créée :

```
In [177]: B = A.flatten()
          B
```

```
Out[177]: array([5, 5, 5, 5])
```

```
In [178]: B[0:5] = 10
          B
```

```
Out[178]: array([10, 10, 10, 10])
```

```
In [179]: A # A ne change pas car B est une copie de A
```

```
Out[179]: array([[5, 5],
                 [5, 5]])
```

Ajouter une nouvelle dimension avec `newaxis`

par exemple pour convertir un vecteur en une matrice ligne ou colonne :

```
In [180]: v = np.array([1,2,3])
```

```
In [181]: np.shape(v)
```

```
Out[181]: (3,)
```

```
In [182]: # créer une matrice à une colonne à partir du vecteur v
          v[:, np.newaxis]
```

```
Out[182]: array([[1],
                 [2],
                 [3]])
```

```
In [183]: v[:, np.newaxis].shape
```

```
Out[183]: (3, 1)
```

```
In [184]: # matrice à une ligne
          v[np.newaxis,:].shape
```

```
Out[184]: (1, 3)
```


Concaténer, répéter des *arrays*

En utilisant les fonctions `repeat`, `tile`, `vstack`, `hstack`, et `concatenate`, on peut créer des vecteurs/matrices plus grandes à partir de vecteurs/matrices plus petites :

repeat et tile

```
In [185]: a = np.array([[1, 2], [3, 4]])  
a
```

```
Out[185]: array([[1, 2],  
                [3, 4]])
```

```
In [186]: # répéter chaque élément 3 fois  
np.repeat(a, 3) # résultat 1-d
```

```
Out[186]: array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4])
```

```
In [187]: # on peut spécifier l'argument axis  
np.repeat(a, 3, axis=1)
```

```
Out[187]: array([[1, 1, 1, 2, 2, 2],  
                [3, 3, 3, 4, 4, 4]])
```

Pour répéter la matrice, il faut utiliser `tile`

```
In [188]: # répéter la matrice 3 fois  
np.tile(a, 3)
```

```
Out[188]: array([[1, 2, 1, 2, 1, 2],  
                [3, 4, 3, 4, 3, 4]])
```

concatenate

```
In [189]: b = np.array([[5, 6]])
```

```
In [190]: np.concatenate((a, b), axis=0)
```

```
Out[190]: array([[1, 2],  
                [3, 4],  
                [5, 6]])
```

```
In [191]: np.concatenate((a, b.T), axis=1)
```

```
Out[191]: array([[1, 2, 5],
                 [3, 4, 6]])
```

hstack et vstack

```
In [192]: np.vstack((a,b))
```

```
Out[192]: array([[1, 2],
                 [3, 4],
                 [5, 6]])
```

```
In [193]: np.hstack((a,b.T))
```

```
Out[193]: array([[1, 2, 5],
                 [3, 4, 6]])
```

Itérer sur les éléments d'un *array*

- Dans la mesure du possible, il faut éviter l'itération sur les éléments d'un *array* : c'est beaucoup plus lent que les opérations vectorisées
- Mais il arrive que l'on n'ait pas le choix...

```
In [194]: v = np.array([1,2,3,4])
```

```
for element in v:
    print element
```

```
1
2
3
4
```

```
In [195]: M = np.array([[1,2], [3,4]])
```

```
for row in M:
    print "row", row

    for element in row:
        print element
```

```
row [1 2]
1
2
row [3 4]
3
4
```

Pour obtenir les indices des éléments sur lesquels on itère (par exemple, pour pouvoir les modifier en même temps) on peut utiliser `enumerate` :

```
In [196]: for row_idx, row in enumerate(M):
          print "row_idx", row_idx, "row", row

          for col_idx, element in enumerate(row):
              print "col_idx", col_idx, "element", element

          # update the matrix M: square each element
          M[row_idx, col_idx] = element ** 2

row_idx 0 row [1 2]
col_idx 0 element 1
col_idx 1 element 2
row_idx 1 row [3 4]
col_idx 0 element 3
col_idx 1 element 4
```

```
In [197]: # chaque élément de M a maintenant été élevé au carré
          M
```

```
Out[197]: array([[ 1,  4],
                 [ 9, 16]])
```

Utilisation d'arrays dans des conditions

Lorsqu'on s'intéresse à des conditions sur tout ou une partie d'un *array*, on peut utiliser `any` ou `all` :

```
In [198]: M
```

```
Out[198]: array([[ 1,  4],
                 [ 9, 16]])
```

```
In [199]: if (M > 5).any():
          print("au moins un élément de M est plus grand que 5")
          else:
              print("aucun élément de M n'est plus grand que 5")
```

```
au moins un élément de M est plus grand que 5
```

```
In [200]: if (M > 5).all():
          print("tous les éléments de M sont plus grands que 5")
          else:
              print("tous les éléments de M sont plus petits que 5")
```

```
tous les éléments de M sont plus petits que 5
```

Type casting

On peut créer une vue d'un autre type que l'original pour un *array*

```
In [201]: M = array([[ -1, 2], [ 0, 4]])  
          M.dtype
```

```
Out[201]: dtype('int64')
```

```
In [202]: M2 = M.astype(float)  
          M2
```

```
Out[202]: array([[ -1.,  2.],  
                [  0.,  4.]])
```

```
In [203]: M2.dtype
```

```
Out[203]: dtype('float64')
```

```
In [204]: M3 = M.astype(bool)  
          M3
```

```
Out[204]: array([[ True,  True],  
                [False,  True]], dtype=bool)
```

Pour aller plus loin

- <http://numpy.scipy.org> (<http://numpy.scipy.org>)
- [http://scipy.org/Tentative NumPy Tutorial](http://scipy.org/Tentative%20NumPy%20Tutorial) (<http://scipy.org/Tentative NumPy Tutorial>)
- [http://scipy.org/NumPy for Matlab Users](http://scipy.org/NumPy%20for%20Matlab%20Users) (<http://scipy.org/NumPy for Matlab Users>) - Un guide pour les utilisateurs de MATLAB.