

Logiciels scientifiques - HLMA310

Partie 1 : Python

Matrices et numpy

Joseph Salmon

<http://josephsalmon.eu>

Université de Montpellier



Sommaire

Introduction et présentations du package `numpy`

Nombres (pseudo)-aléatoires

Importations/Exportations externes

Affectation et copie

Introduction

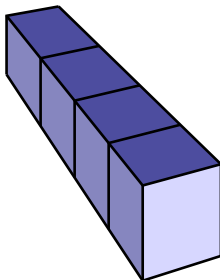
numpy ou comment se passer de Matlab :

- ▶ faire du calcul numérique
- ▶ faire du traitement sur des vecteurs, matrices et tenseurs

Ressources :

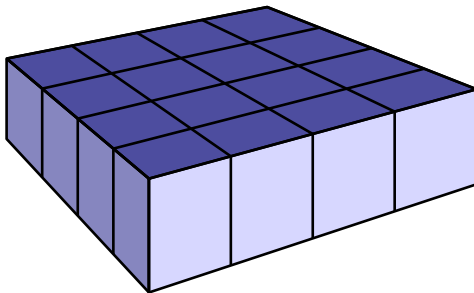
- ▶ Science des données : **Van der Plas (2016)** (en anglais) et dont le site internet associé est une mine de ressource :
<https://jakevdp.github.io/PythonDataScienceHandbook/index.html>
- ▶ numpy/scipy : <http://www.scipy-lectures.org/>

Représentation de tenseur 1D



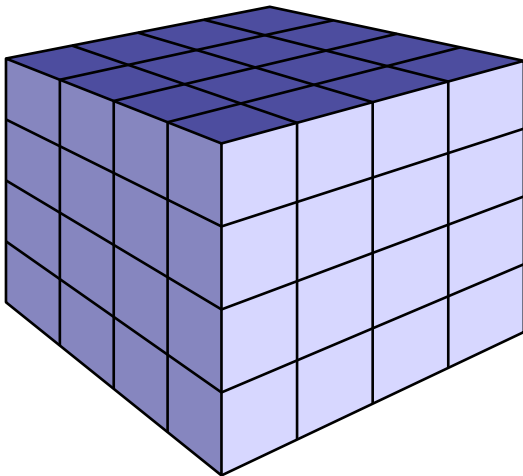
Cas vectoriel

Représentation de tenseur 2D



Cas matriciel

Représentation de tenseur 3D



Cas tensoriel

Forces et faiblesses

Points forts de numpy :

- ▶ gratuit / open source
- ▶ écrit en C et en Fortran \implies performances élevées (pour les calculs vectorisés, *i.e.*, calculs formulés comme des opérations sur des vecteurs/matrices) ⁽¹⁾

Points faibles de numpy :

- ▶ plus verbeux que Matlab ou Julia (open source), dont la syntaxe est plus proche des mathématiques
- ▶ prise en main moins intuitive

(1) interfaçant BLAS/LAPACK


Chargement classique de numpy

Chargement standard de numpy :

```
>>> import numpy as np # importe numpy  
>>> np.__version__ # vérifie la version utilisée  
'1.14.3'
```

Rem: L'extension `.np` est courante et sera employée dans la suite pour toute référence à numpy.

array/arrays

Dans la terminologie numpy : vecteurs, matrices et autres tenseurs sont appelés arrays ( : *tableaux*).

Comme de création d'array :

- ▶ à partir de listes ou n-uplets Python
- ▶ avec des fonctions dédiées, telles que `arange`, `linspace`, etc.
- ▶ par chargement, à partir de fichiers externes

Création d'array à partir de liste

Vecteur : l'argument est une liste Python

```
>>> liste = [1, 3, 2, 4]
>>> vecteur = np.array(liste)
>>> print(vecteur)
[1 3 2 4]
```

Matrice : l'argument est une liste de liste

```
>>> matrice = np.array([[1, 2], [3, 4]])
>>> print(matrice)
>>> print(matrice[0,0])
>>> print(matrice[0,1])
[[1 2]
 [3 4]]
1
2
```

Création d'array : les tenseurs

Tenseur :

```
>>> tenseur = np.array([[[1, 2], [3, 4]],  
                        [[21, 21], [23, 34]]])  
>>> print(tenseur)  
[[[ 1  2]  
  [ 3  4]]  
  
 [[21 21]  
  [23 34]]]
```

array : type, taille, etc.

Vecteurs et matrices ont même type : ndarray

```
>>> type(vecteur), type(matrice)
(numpy.ndarray, numpy.ndarray, numpy.ndarray)
```

Connaître les dimensions avec shape :

```
>>> np.shape(vecteur), matrice.shape, tenseur.shape
((4,), (2, 2), (2, 2, 2))
```

Forcer les types de données dans un array :

```
>>> matrice_cpx = np.array([[1, 2], [3, 4]], dtype=complex)
>>> matrice_cpx
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

Plus sur les types

Autres types possibles reconnus par l'argument optionnel dtype :

- ▶ int
- ▶ float
- ▶ complex
- ▶ bool

Rem: possibilité de donner la précision (en bits) également avec int64, int16, float128, complex128.

Plus sur ces types :

- ▶ <https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html>
- ▶ <https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html>

Création d'array par fonction de génération

arange : crée un array de nombres de type **float** ou **int**

```
>>> x = np.arange(0, 10, 2) # start, stop, step
>>> x
array([0, 2, 4, 6, 8])
```

ou pour des flottants :

```
>>> y = np.arange(-1, 1, 0.5)
>>> y
array([-1. , -0.5, 0. , 0.5])
```

Enfin pour connaître le type :

```
>>> print(x.dtype, y.dtype)
int64 float64
```

linspace, logspace

`linspace` :

```
>>> np.linspace(0, 5, 11) # début et fin sont inclus  
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
```

`logspace` : similaire mais travaille en échelle logarithmique

```
>>> np.set_printoptions(precision=2) # for display  
>>> np.logspace(0, 11, 10) # début et fin sont inclus  
array([1.00e+00, 1.67e+01, 2.78e+02, 4.64e+03, 7.74e+04, 1.29e+06,  
       2.15e+07, 3.59e+08, 5.99e+09, 1.00e+11])
```

Rem: 0, renvoie à 10^0 ; 11 renvoie à 10^{11} , et 10 permet d'afficher 10 nombres entre ces deux bornes (en progression géométrique)

Utilité : pour afficher la vitesse de convergence d'algorithmes d'optimisation, pour l'étude des prix en économie, ...

```
>>> 10**(np.linspace(np.log10(10), np.log10(10**11), 10))  
array([1.00e+00, 1.67e+01, 2.78e+02, 4.64e+03, 7.74e+04, 1.29e+06,  
       2.15e+07, 3.59e+08, 5.99e+09, 1.00e+11])
```

diag

```
>>> np.diag([1, 2, 3], k=0)
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

Rem: diag peut extraire d'un array sa diagonale

```
>>> np.diag(np.diag([1, 2, 3], k=0))
array([1, 2, 3])
```

sous/sur diagonale :

```
>>> np.diag([1, 2, 3], k=1)
array([[0, 1, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 3],
       [0, 0, 0, 0]])
```


Transposition

Pour obtenir la **transposition** d'une matrice il suffit d'appliquer l'argument `.T` à la matrice :

```
>>> np.diag([1, 2, 3], k=1).T  
array([[0, 0, 0, 0],  
       [1, 0, 0, 0],  
       [0, 2, 0, 0],  
       [0, 0, 3, 0]])
```

Aussi la fonction `transpose` permet la même opération :

```
>>> np.transpose(np.diag([1, 2, 3], k=1))  
array([[0, 0, 0, 0],  
       [1, 0, 0, 0],  
       [0, 2, 0, 0],  
       [0, 0, 3, 0]])
```

zeros

```
>>> np.zeros((3,), dtype=int)
array([0, 0, 0])
```

ou encore :

```
>>> print(np.zeros((3, 2), dtype=float))
>>> print(np.zeros((1, 3), dtype=float))
>>> print(np.zeros((3, 1), dtype=float))
[[0. 0.]
 [0. 0.]
 [0. 0.]]
[[0. 0. 0.]]
[[0.]
 [0.]
 [0.]]
```

ones, full et eye

ones :

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

full : remplit un array avec n'importe quel nombre

```
>>> np.full((3, 5), 3.14)  
array([[3.14, 3.14, 3.14, 3.14, 3.14],  
       [3.14, 3.14, 3.14, 3.14, 3.14],  
       [3.14, 3.14, 3.14, 3.14, 3.14]])
```

eye : matrice identité

```
>>> np.eye(3)  
array([[1., 0., 0.],  
       [0., 1., 0.],  
       [0., 0., 1.]])
```

Concaténation d'array

Concaténation verticale :

```
>>> A = np.array([[0, 2],[ 3, 4]])  
>>> B = np.array([[1, 2],[ 5, 4]])  
>>> np.vstack((A,B)) # concaténation verticale  
array([[0, 2],  
       [3, 4],  
       [1, 2],  
       [5, 4]])
```

Concaténation horizontale :

```
>>> np.hstack((A,B)) # concaténation horizontale  
array([[0, 2, 1, 2],  
       [3, 4, 5, 4]])
```

Fonctions sur les lignes / colonnes : moyenne (: *mean*)

Moyenne globale :

```
>>> np.mean(A)  
2.25
```

Moyenne des colonnes :

```
>>> np.mean(B,axis=0) # moyenne en colonne  
array([3., 3.])
```

Moyenne des lignes :

```
>>> np.mean(B,axis=1) # moyenne en ligne  
array([1.5, 4.5])
```

Fonctions sur les lignes / colonnes : somme (: *sum*)

Somme globale :

```
>>> np.sum(A)  
9
```

Somme des colonnes :

```
>>> np.sum(A, axis=0) # somme en colonne  
array([3, 6])
```

Somme des lignes :

```
>>> np.sum(A, axis=1) # somme en ligne  
array([2, 7])
```

Fonctions sur les lignes / colonnes :

Somme cumulée (: *cumsum*)

Somme cumulée globale :

```
>>> np.cumsum(A) # noter l'ordre en ligne  
array([0, 2, 5, 9])
```

Somme cumulée des colonnes :

```
>>> np.cumsum(A, axis=0) # somme cumulée en colonne  
array([[0, 2],  
       [3, 6]])
```

Somme cumulée des lignes :

```
>>> np.cumsum(A, axis=1) # somme cumulée en ligne  
array([[0, 2],  
       [3, 7]])
```

Rem: idem pour prod et cumprod

Slicing

slicing : disponible pour les arrays (même syntaxe que pour les listes et les chaînes de caractères, avec en plus la possibilité d'y avoir accès pour chaque dimension) :

Accès en colonne :

```
>>> A[:,0] # accès première colonne  
array([0, 3])
```

Accès en ligne :

```
>>> A[1,:] # accès deuxième ligne  
array([3, 4])
```


Masques

Utilisation de masques booléens pour extraire certains éléments :

```
>>> print(A < 2)
>>> print(A[A < 2])
>>> A[A < 2] = 10
>>> print(A)
>>> A[0, 0] = 0
>>> print(A)
[[ True False]
 [False False]]
[0]
[[10 2]
 [3 4]]
[[0 2]
 [3 4]]
```

Vectorisation des opérations

On peut appliquer les opérations usuelles sur un array : elles seront alors appliquées terme à terme :

```
>>> 2**A  
array([[ 1,  4],  
       [ 8, 16]])
```

```
>>> A**3  
array([[ 0,  8],  
       [27, 64]])
```

```
>>> A + 1  
array([[1, 3],  
       [4, 5]])
```



$\text{np.exp}(A) \neq \text{scipy.linalg.expm}(A)$
terme à terme \neq matriciel

Multiplication des matrices

Il y a plusieurs syntaxe pour faire le produit matriciel :

```
>>> A @ B  
array([[10, 8],  
       [23, 22]])
```

```
>>> A.dot(B)  
array([[10, 8],  
       [23, 22]])
```

```
>>> np.dot(A, B)  
array([[10, 8],  
       [23, 22]])
```

Sommaire

Introduction et présentations du package `numpy`

Nombres (pseudo)-aléatoires

Importations/Exportations externes

Affectation et copie

Génération aléatoire

Utilité : en statistique et en probabilité

- ▶ détails sur la manière dont Python gère la création de nombres pseudo-aléatoires :
<https://docs.python.org/3/library/random.html>
- ▶ détails sur le type d'algorithme utilisé (par défaut l'algorithme de Mersenne Twister), on pourra se référer à
https://fr.wikipedia.org/wiki/Mersenne_Twister



en informatique, les opérations sont déterministes : l'aléatoire n'existe pas, on parle de pseudo-aléatoire

Générateur de nombres aléatoires

“uniformes⁽²⁾” entre 0. et 1.

Module pour générer des nombres aléatoires :

- ▶ `import random` : le module aléatoire standard
- ▶ `np.random` : le module aléatoire de numpy

```
>>> np.random.rand(5, 5) # aléatoire entre 0. et 1.  
array([[0.2 , 0.63, 0.17, 0.11, 0.8 ],  
       [0.16, 0.16, 0.52, 0.03, 0.87],  
       [0.05, 0.41, 0.98, 0.36, 0.68],  
       [0.66, 0.45, 0.24, 1. , 0.66],  
       [0.14, 0.91, 0.13, 0.14, 0.13]])
```


En relançant la commande, la matrice créée est différente!!!

Rem: `np.set_printoptions(precision=2)` a restreint notre affichage aux deux premiers chiffres après la virgule, d'où le “1.” !

(2) un rappel sur la loi uniforme est donné ici : https://fr.wikipedia.org/wiki/Loi_uniforme_continue

Graine (: *seed*)

Pour la reproductibilité des résultats on peut souhaiter “geler” l'aléa (e.g., pour des tests ou du débogage)

Graine ( : *seed*) : nombre utilisé pour initialiser un générateur de nombres pseudo-aléatoires

Fonctionnement : fixer une graine permet de spécifier de manière déterministe l'appel des opérations pseudo-aléatoires

```
>>> np.random.seed(2018) # fixe la graine
>>> np.random.rand(3, ) # tirage aléatoire
array([0.88234931, 0.10432774, 0.90700933])
```

Résultat : cette commande produit alors toujours la même sortie

Autres lois : exemple gaussien

Construction similaire pour d'autres lois : la plus connue étant la loi **gaussienne** (ou loi normale), dont la fonction de densité est :

$$\varphi(x) = \frac{\exp\left(-\frac{x^2}{2}\right)}{\sqrt{2\pi}}$$

On remplace alors `np.random.rand` par `np.random.randn`

- ▶ Plus de détails sur cette loi :

https://fr.wikipedia.org/wiki/Loi_normale

- ▶ Pour d'autres lois classiques, cf. l'aide de numpy :

<https://docs.scipy.org/doc/numpy-1.15.0/reference/routines.random.html>

Rem: `scipy` : module d'outils scientifiques (statistiques, optimisation, etc.); `matplotlib` : module d'affichage graphique

Affichage : histogramme et densité

```
from scipy.stats import norm # module loi normale

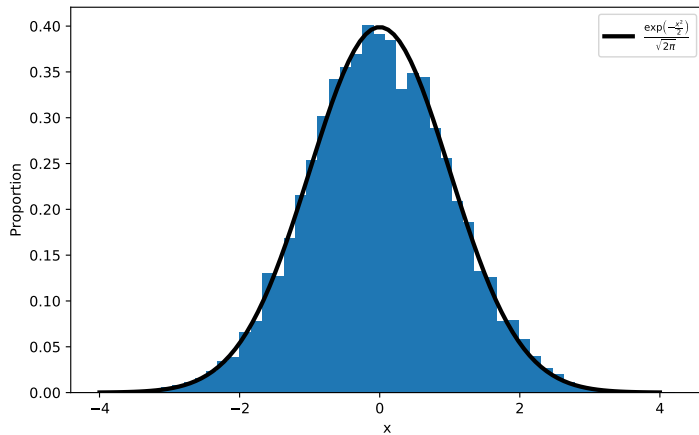
a = np.random.randn(10000)
x = np.linspace(-4, 4, 100)

# Affichage d'un histogramme normalisé
fig = plt.figure(figsize=(8, 5))
hitogramme = plt.hist(a, bins=50, density=True)

# Oublier les détails matplotlib et Latex si besoin
plt.plot(x, norm.pdf(x), linewidth=3, color='black',
         label=r"$\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)$")

plt.xlabel("x")
plt.ylabel("Proportion")
plt.legend()
plt.show()
```

Histogramme et densité : cas gaussien



Sommaire

Introduction et présentations du package `numpy`

Nombres (pseudo)-aléatoires

Importations/Exportations externes

- Importations

- Exportations

Affectation et copie

Import de fichiers en numpy

But : importer un fichier en ligne de commande (sans télécharger à la main avec un navigateur)

Format visé (ici) : .csv ( : *comma separated values*).

```
>>> import os # interface système d'exploitation
>>> from download import download
```

Localiser le répertoire courant :

```
>>> !pwd # unix command "print working directory"
/home/jo/Documents/Mes_cours/Montpellier/HLMA310/Poly/codes
```

Sommaire

Introduction et présentations du package `numpy`

Nombres (pseudo)-aléatoires

Importations/Exportations externes

Importations

Exportations

Affectation et copie

Importation de fichiers (suite)

Url contenant le fichier de données :

```
>>> url = "http://josephsalmon.eu/enseignement/datasets/data_test.csv"
```

Téléchargement des fichiers :

```
>>> from download import download
>>> path_target = "./data_set.csv"
>>> download(url, path_target, replace=False)
Télécharge

file_sizes: 100% 30.0/30.0 [00:00<00:00, 37.0kB/s]

Downloading data from
http://josephsalmon.eu/enseignement/datasets/data_test.csv
(30 bytes)

Successfully downloaded file to ./data_set.csv
```

Visualisation du fichier brut

Après téléchargement, le fichier brut est visualisable :

```
>>> !cat data_set.csv # commande pour visualiser  
1,2,3,4,5  
6,7,8,9,10  
1,3,3,4,6
```

Rem: autres fonctions unix disponibles

- ▶ cd : changer de dossier (*change **directory*** en anglais)
- ▶ cp : copier des fichiers (***copy*** en anglais)
- ▶ ls : lister les fichiers à l'endroit courant (***list*** en anglais)
- ▶ man : avoir accès au manuel/aide (***manual*** en anglais)
- ▶ mkdir : créer un dossier (***make **directory***** en anglais)
- ▶ mv : déplacer un fichier (***move **directory***** en anglais)
- ▶ rm : supprimer un fichier (***remove*** en anglais)
- ▶ rmdir : supprimer un dossier (***remove **directory***** en anglais)

Import du fichier sous numpy

Import et lecture au format array de numpy :

```
>>> data_as_array = np.genfromtxt('data_set.csv',  
                                   delimiter=',')  
  
>>> data_as_array  
array([[ 1.,  2.,  3.,  4.,  5.],  
       [ 6.,  7.,  8.,  9., 10.],  
       [ 1.,  3.,  3.,  4.,  6.]])
```

Rappel : csv signifie “comma-separated values” (■ ■ : *valeurs séparées par des virgules*)

Sommaire

Introduction et présentations du package `numpy`

Nombres (pseudo)-aléatoires

Importations/Exportations externes

Importations

Exportations

Affectation et copie

Export sous forme de texte

Export standard en.csv ou en .txt :

```
np.savetxt("random_matrix.txt", data_as_array)
```

Vérification :

```
>>> !cat random_matrix.txt  
1.000000000000000000e+00 2.000000000000000000e+00 ..  
...
```

Solution alternative : ouvrir le fichier avec un éditeur de texte (gedit, mousepad, etc.) pour vérifier son contenu (pas avec Word ou LibreOffice)

Export sous format .npz

Sauvegarder ( : *save*) un array :

```
>>> np.save("random_matrix.npz", data_as_array)
```

Charger ( : *load*) :

```
>>> data_as_array2 = np.load("random_matrix.npz")
>>> data_as_array2
array([[ 1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10.],
       [ 1.,  3.,  3.,  4.,  6.]])
```

Rem: Pour effacer le fichier après usage (si besoin)

```
!rm data_set.csv
!rm random_matrix.txt
!rm random_matrix.npz
```

Sommaire

Introduction et présentations du package `numpy`

Nombres (pseudo)-aléatoires

Importations/Exportations externes

Affectation et copie

Affectation simple


```
>>> A = np.array([[0, 2], [3, 4]])  
>>> B = A
```



changer B va maintenant affecter A : l'affectation ne copie pas le tableau, mais "l'adresse mémoire" du tableau d'origine

```
>>> B[0,0] = 10  
>>> print(B)  
>>> print(B is A) # les deux objets sont les mêmes  
>>> print(A)  
array([[10, 2],  
       [ 3, 4]])  
True  
array([[10 2]  
       [ 3 4]])
```

Copie profonde (*deep copy*)

Copie profonde ( : *deep copy*) de A dans B :

```
>>> B = A.copy()
```

Maintenant on peut observer le comportement différent :

```
>>> B[0, 0] = 111
>>> B
array([[111, 2],
       [ 3,  4]])
```

A n'est alors plus modifié : on a créé une copie profonde de l'objet !

```
>>> A
array([[10, 2],
       [ 3,  4]])
```

Tests entre array

Test (exact) terme à terme :

```
>>> A == B  
array([[False,  True],  
       [ True,  True]])
```

Test approché :

```
>>> np.allclose(A, A+0.001)  
False
```

Test approché à précision choisie :

```
>>> np.allclose(A, A+0.001, atol=0.01)  
True
```

Rem: atol signifie *absolute tolerance*

Bibliographie I

- ▶ VANDERPLAS, J. *Python Data Science Handbook*. O'Reilly Media, 2016.