Manipulation des matrices avec numpy

Création, accès, extraction, calculs

Ricco Rakotomalala

http://eric.univ-lyon2.fr/~ricco/cours/cours_programmation_python.html

- Numpy est un package pour Python spécialisé dans la manipulation des tableaux (array), pour nous essentiellement les vecteurs et les matrices
- Les tableaux « numpy » ne gère que les objets de même type
- Le package propose un grand nombre de routines pour un accès rapide aux données (ex. recherche, extraction), pour les manipulations diverses (ex. tri), pour les calculs (ex. calcul statistique)
- Les tableaux « numpy » sont plus performants (rapidité, gestion de la volumétrie) que les collections usuelles de Python
- Les tableaux « numpy » sont sous-jacents à de nombreux packages dédiés au calcul scientifique sous Python.
- Une matrice est un tableau (array) à 2 dimensions

Il n'est pas possible de tout aborder dans ce support. Pour aller plus loin, voir absolument le manuel de référence (utilisé pour préparer ce diaporama).

http://docs.scipy.org/doc/numpy/reference/index.html

Création à la volée, génération d'une séquence, chargement à partir d'un fichier

CRÉATION D'UNE MATRICE

Préalable important : importer le module « numpy »

```
import numpy as np
```

np sera l'alias utilisé pour accéder aux routines de la librairie « numpy ».

Création manuelle à partir d'un ensemble de valeurs

$$\begin{pmatrix} 1.2 & 2.5 \\ 3.2 & 1.8 \\ 1.1 & 4.3 \end{pmatrix}$$

```
a = np.array([[1.2,2.5],[3.2,1.8],[1.1,4.3]])
```

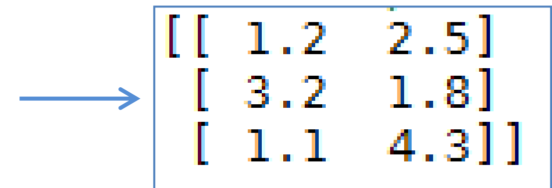
Noter le rôle des [ ] et [ ] pour délimiter les portions de la matrice

Informations sur la structure

```
#type de la structure
print(type(a))  #<class 'numpy.ndarray'>
#type des données
print(a.dtype)  #float64
#nombre de dimensions
print(a.ndim)  #2 (car c'est une matrice)
#nombre de lignes et col, shape renvoie un tuple
print(a.shape)  #(3,2) → 3 lignes et 2 colonnes
#nombre totale de valeurs
print(a.size)  #6, nb.lignes x nb.colonnes
```

Affichage d'une matrice dans la console (IPython)

```
#print de l'ensemble
print(a)
```



Le typage des valeurs peut être implicite ou explicite

```
#création et typage implicite
a = np.array([[1,2],[4,7]])
print(a.dtype) #int32

#création et typage explicite – préférable !
a = np.array([[1,2],[4,7]],dtype=float)
print(a.dtype) #float64
```

Tout comme pour les vecteurs, la création d'une matrice d'objets complexes (autres que les types de base) est possible

```
#création à partir d'une séquence
#attention les dim. doivent être compatibles
a = np.arange(0,10).reshape(2,5)
print(a)
```

arange() génère une séquence de valeurs, 0 à 9.
reshape() se charge de les réorganiser en matrice 2 lignes et 5 colonnes.

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

```
#un vecteur peut être converti en matrice
a = np.array([2.1,3.4,6.7,8.1,3.5,7.2])
print(a.shape) # (6,)
#redim. en 3 lignes x 2 col.
b = a.reshape(3,2)
print(b.shape) # (3, 2)
print(b)
```

```
[[ 2.1  3.4]
 [ 6.7  8.1]
 [ 3.5  7.2]]
```

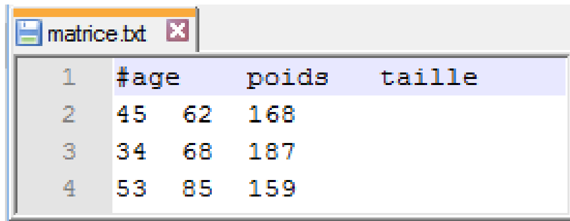
```
#matrices de valeurs identiques
#ex. pour une initialisation
a = np.zeros(shape=(2,4))
print(a)
```

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

```
#plus généralement
a = np.full(shape=(2,4),fill_value=0.1)
print(a)
```

```
[[ 0.1  0.1  0.1  0.1]
 [ 0.1  0.1  0.1  0.1]]
```

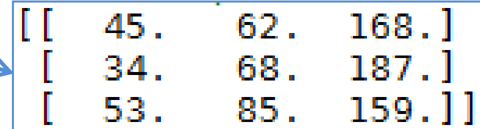
Les données peuvent être stockées dans un fichier texte (loadtxt pour charger, savetxt pour sauver)



La première ligne doit être ignorée dans ce fichier, d'où le symbole # en début de 1ère ligne.

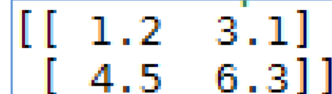
Remarque : si besoin, modifier le répertoire par défaut avec la fonction **chdir()** du module **os** (qu'il faut importer au préalable)

```
#charger à partir d'un fichier, typage explicite
#séparateur de colonne = tabulation « \t »
a = np.loadtxt("matrice.txt",delimiter="\t",dtype=float)
print(a)
```



Conversion d'une collection (type standard Python) en type array de « numpy »

```
#liste de valeurs
lst = [1.2,3.1,4.5,6.3]
print(type(lst)) # <class 'list'>
#conversion à partir d'une liste : 2 étapes asarray() et reshape()
a = np.asarray(lst,dtype=float).reshape(2,2)
print(a)
```



a =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#matrice de valeurs
a = np.array([[1.2,2.5],[3.2,1.8],[1.1,4.3]])
```

Accoler le vecteur **b** en tant que nouvelle ligne (axis = 0) de la matrice a

```
#ajouter une ligne – marche pour la concaténation de matrices
b = np.array([[4.1,2.6]])
c = np.append(a,b,axis=0)
print(c)
```

```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]
 [ 4.1  2.6]]
```

Accoler le vecteur **d** en tant que nouvelle colonne (axis = 1) de la matrice a

```
#ajouter une colonne
d = np.array([[7.8],[6.1],[5.4]])
print(np.append(a,d,axis=1))
```

```
[[ 1.2  2.5  7.8]
 [ 3.2  1.8  6.1]
 [ 1.1  4.3  5.4]]
```

Insertion de **b** en tant que nouvelle ligne (axis = 0) à la position n°1

```
#insertion
print(np.insert(a,1,b,axis=0))
```

```
[[ 1.2  2.5]
 [ 4.1  2.6]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

Suppression de la ligne (axis = 0) via son indice (n°1)

```
#suppression
print(np.delete(a,1,axis=0))
```

```
[[ 1.2  2.5]
 [ 1.1  4.3]]
```

Redimensionnement d'une matrice

```
#modifier la dimension d'une matrice existante
#parcourt les données lignes par ligne
h = np.resize(a,new_shape=(2,3))
print(h)
```

```
[[ 1.2  2.5  3.2]
 [ 1.8  1.1  4.3]]
```

Accéder aux valeurs via des indices ou des conditions

EXTRACTION DES VALEURS

```
v = np.array([[1.2,2.5],[3.2,1.8],[1.1,4.3]])

#affichage de la structure dans son ensemble
print(v)
```

v =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#accès indicé - première valeur
print(v[0,0]) # 1.2

#dernière valeur – noter l'utilisation de shape (qui est un tuple)
print(v[v.shape[0]-1,v.shape[1]-1]) # 4.3

#autre solution pour affichage de toutes les valeurs, noter le rôle des :
print(v[:,:])

#plage d'indices contigus : lignes 0 à 1 (2 non inclus), toutes les colonnes
print(v[0:2,:])
```

```
[[ 1.2  2.5]
 [ 3.2  1.8]]
```

```
#extrêmes, début to 2 (non-inclus)
print(v[:2,:])
```

```
[[ 1.2  2.5]
 [ 3.2  1.8]]
```

```
#extrêmes, lignes 1 à dernière
print(v[1:,:])
```

```
[[ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#indice négatif – dernière ligne et toutes les colonnes
print(v[-1,:])
```

```
[ 1.1  4.3]
```

```
#indices négatifs – deux dernières lignes et toutes les colonnes
print(v[-2:,:])
```

```
[[ 3.2  1.8]
 [ 1.1  4.3]]
```

Remarques :

(1) Mis à part les singletons, les matrices générées sont de type numpy.ndarray

(2) Toutes comme pour les vecteurs, il est possible d'utiliser un vecteur d'indices non contigus.

v =

```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#indiçage par vecteur de booléens
#si b trop court, tout le reste est considéré False
#si b trop long, erreur
b = np.array([True,False,True],dtype=bool)
print(v[b,:])
```

```
[[ 1.2  2.5]
 [ 1.1  4.3]]
```

#exemple illustratif : extraire la lignes dont la somme est la plus petite

```
#calculer la somme des colonnes pour chaque ligne
s = np.sum(v,axis=1)
print(s) # [ 3.7  5.  5.4 ]

#repérer les lignes dont la somme est égale au minimum
#il est possible qu'il y en ait plusieurs
b = (s == np.min(s))
print(b) # [ True  False  False]

#application du filtre booléen
print(v[b,:])
```

```
[[ 1.2  2.5]]
```

Remarquer la configuration des crochets [ ] : on a une **matrice** à 1 ligne et 2 colonnes.

v =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#recherche valeur max des lignes (axis = 0) pour chaque colonne
print(np.max(v,axis=0)) # [ 3.2   4.3 ] -- décryptage : 3.2 est la max des lignes pour la colonne 0, 4.3 est la max des lignes pour la colonne 1

#recherche valeur max des colonnes (axis = 1) pour chaque ligne
print(np.max(v,axis=1)) # [ 2.5  3.2  4.3]

#recherche indice de valeur max des lignes (axis = 0)pour chaque colonne
print(np.argmax(v,axis=0)) # [ 1   2 ]

#tri des lignes (axis = 0) pour chaque colonne
#la relation entre les valeurs d'une même ligne est perdue !!!
print(np.sort(v,axis=0))
```

```
[[ 1.1  1.8]
 [ 1.2  2.5]
 [ 3.2  4.3]]
```

```
#récupération des indices triés
print(np.argsort(v,axis=0))
```

```
[[2 1]
 [0 0]
 [1 2]]
```

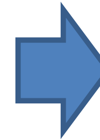
Stratégies pour parcourir une matrice

ITÉRATIONS

Avec les indices, nous pouvons accéder aux valeurs de la matrice comme bon nous semble (ligne par ligne ou colonne par colonne)

v =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#boucles indicées
s = 0.0
for i in range(0,v.shape[0]):
    for j in range(0,v.shape[1]):
        print(v[i,j])
        s = s + v[i,j]
print("Somme = ",s)
```



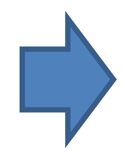
```
1.2
2.5
3.2
1.8
1.1
4.3
Somme =  14.1
```

Avec les itérateurs, nous pouvons accéder aux valeurs de la matrice sans avoir à recourir aux indices (ligne par ligne, colonne par colonne)

v =

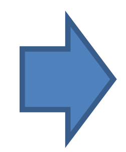
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#itérateur - accès ligne par ligne
s = 0.0
for x in np.nditer(v):
    print(x)
    s = s + x
print("Somme = ",s)
```



```
1.2
2.5
3.2
1.8
1.1
4.3
Somme =  14.1
```

```
#itérateur - accès colonne par colonne
#"F" pour " Fortran order "
s = 0.0
for x in np.nditer(v,order="F"):
    print(x)
    s = s +x
print("Somme = ",s)
```



```
1.2
3.2
1.1
2.5
1.8
4.3
Somme =  14.1
```

Les itérateurs de NumPy sont sophistiqués et puissants, voir : http://docs.scipy.org/doc/numpy/reference/arrays.nditer.html

Calcults statistiques

CALCULS SUR LES MATRICES

Principe : les calculs sont réalisés selon une certaine organisation des données (axis = None : toutes les valeurs prises globalement ; axis = 0 : traitement par colonne ; axis = 1 : traitement par ligne)

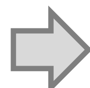
v =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#moyenne par colonne
print(np.mean(v,axis=0)) # [1.833   2.867]

#moyenne par ligne
print(np.mean(v,axis=1)) # [1.85   2.5   2.7]

#somme cumulée des valeurs pour chaque colonne
print(np.cumsum(v,axis=0))
```

```
[[ 1.2  2.5]
 [ 4.4  4.3]
 [ 5.5  8.6]]
```

```
#matrice de corrélation
#rowvar = 0 pour indiquer que les variables
#sont organisés en colonnes
m = np.corrcoef(v,rowvar=0)
print(m)
```

```
[[ 1.          -0.74507396]
 [-0.74507396  1.        ]]
```

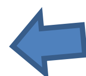


La librairie n'est pas très fournie, nous aurons besoin de SciPy (et autres)

Principe : A la manière de la fonction apply() de R, nous pouvons définir des calculs le long d'un axe d'une matrice (0 par colonne, 1 par ligne). Chaque colonne (ligne) est passée en paramètre à une fonction callback.

v =
```
[[ 1.2   2.5]
 [ 3.2   1.8]
 [ 1.1   4.3]]
```

```
#moyenne par colonne : [1.833   2.867]
print(np.apply_along_axis(func1d=np.mean,axis=0,arr=v))

#une fonction callback – étendue standardisée
def etendue_std(x):
    res = (np.max(x) - np.min(x))/np.std(x)
    return res

#étendue normalisée par colonne : [2.171   2.374]
print(np.apply_along_axis(func1d=etendue_std,axis=0,arr=v))

#fonction peut-être définie à la volée avec lambda
print(np.apply_along_axis(func1d=lambda x:(np.max(x)-np.min(x))/np.std(x),axis=0,arr=v))

#cas où la fonction callback renvoie un vecteur
#nous obtenons une matrice. Ex. centrage-réduction
print(np.apply_along_axis(func1d=lambda x:(x-np.mean(x))/np.std(x),axis=0,arr=v))
```



```
[[-0.65 -0.35]
 [ 1.41 -1.01]
 [-0.76  1.36]]
```

NumPy donne sa pleine mesure pour le calcul matriciel

CALCUL MATRICIEL

x =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

y =
```
[[ 2.1  0.8]
 [ 1.3  2.5]]
```

```
#transposition
print(np.transpose(x))
```

```
[[ 1.2  3.2  1.1]
 [ 2.5  1.8  4.3]]
```

```
#multiplication
print(np.dot(x,y))
```

```
[[  5.77   7.21]
 [  9.06   7.06]
 [  7.9   11.63]]
```

```
#déterminant
print(np.linalg.det(y))
```

4.21

```
#inversion
print(np.linalg.inv(y))
```

```
[[ 0.59382423 -0.19002375]
 [-0.3087886   0.49881235]]
```

x =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

y =
```
[[ 2.1  0.8]
 [ 1.3  2.5]]
```

Solution de

Y.**a** = z

```
#résolution d'équation
z = np.array([1.7,1.0])
print(np.linalg.solve(y,z)) # [0.8195   -0.0261]
```

On peut faire

a = Y^{-1}.z

```
#vérification
print(np.dot(np.linalg.inv(y),z)) # [0.8195   -0.0261]
```

```
#matrice symétrique avec X^TX
s = np.dot(np.transpose(x),x)
print(s)
```

```
[[ 12.89  13.49]
 [ 13.49  27.98]]
```

```
#val. et vec. propres d'une matrice symétrique
print(np.linalg.eigh(s))
```

```
(array([  4.97837925,  35.89162075]),
array([[-0.86259502,  0.50589508],
       [ 0.50589508,  0.86259502]]))
```

Références

De la documentation à profusion **(n'achetez pas des livres sur Python)**

Site du cours
http://eric.univ-lyon2.fr/~ricco/cours/cours_programmation_python.html

Site de Python
Welcome to Python - https://www.python.org/
Python **3.4.3** documentation - https://docs.python.org/3/index.html

Portail Python
Page Python de Developpez.com

Quelques cours en ligne
P. Fuchs, P. Poulain, « Cours de Python » sur Developpez.com
G. Swinnen, « Apprendre à programmer avec Python » sur Developpez.com
« Python », Cours interactif sur Codecademy

POLLS (KDnuggets)
Data Mining / Analytics Tools Used
Python, 4ème en 2015
What languages you used for data mining / data science?
Python, 3ème en 2014 (derrière R et SAS)