Bien débuter avec les tableaux de nombres

Par dramac.nicolas

www.siteduzero.com

Dernière mise à jour le 25/09/2011

Sommaire

Bien débuter avec les tableaux de nombres

Sommaire du chapitre :

Informations sur le tutoriel

Auteur :

- dramac.nicolas

Difficulté :

Définition et installation

Définition d'un tableau de nombres :

Qu'est ce que j'appelle un tableau de nombre : simplement un vecteur, une matrice, ou tout regroupement organisé de nombre. Ce regroupement doit avoir une dimension, un nombre d'éléments, et un type (entier, flottant, etc.).

De manière informatique : une collection indexable et contigüe d'éléments de même type.

Pour disposer d'un tel objet nous allons utiliser le paquet Numpy (url) qui va mettre à notre disposition l'objet ndarray[], un objet type dérivé de "list" ne contenant QUE des nombres du même type (entier, flottant, flottant double précision, logique), comportant un indice (sa position dans le tableau), un certain nombre de méthodes utiles (minimum, maximum, tri, etc.) et pour finir, cet objet est à la base d'autres paquets fondamentaux pour toutes applications scientifiques, mais nous y reviendront plus tard.

Pour avoir cet objet disposition il faut installer le paquet numpy, 3 cas :

- **Linux ou Mac OS/X :** via votre gestionnaire de paquets préféré (apt-get, yum, pacman, etc) ou directement via le gestionnaire de paquet python pip :
  Exemple sous Ubuntu:
  Code : Console
  ```
  sudo apt-get install python-numpy
  ```
- **Mac OS/X :** via macport :
  Code : Console
  ```
  sudo port install py27-numpy
  ```
- **Windows :** via la distribution Python(x,y) (qui a le bon goût d'être en français), ou via les sources.

La dernière option qui devrait marcher dans tout les cas est de télécharger l'archive sur le site officiel, se mettre dans une console avec des droits administrateur, aller dans le dossier et entrer:

Code : Console
```
python setup.py install
```

Ensuite pour l'utiliser dans votre script (ou en console interactive) :

Code : Python
```
import numpy as np
```

Mon premier tableau de nombres

A partir de là les outils numpy sont disponibles via `np.truc_numpy()`.

Création et remplissage de Tableau

Création simple de tableau

Créons donc un tableau ndarray[], un tableau d'entiers, de dimension 2x3, par exemple une matrice 2x3, plein de "0", :

Code : Python

```
tableau_de_zero = np.zeros((2, 3), dtype='i')
```

Mon tableau s'appelle: "tableau de zero", il est créé par la méthode np.zeros(), à laquelle je donne les dimensions de mon tableau: 2 lignes, et 3 colonnes. Pour finir je définis le type de nombre de mon tableau: des entiers via dtype="i" pour "integer" (entier en anglais).

Si on vérifie :

Code : Python Console

```
>>> import numpy as np      # j'importe la bibliothèque Numpy
>>> tableau_de_zero = np.zeros((2, 3), dtype='i') # je crée mon
tableau 2x3 de type entier
>>> print "mon tableau de zero:", tableau_de_zero       # j'affiche
mon tableau
[[0 0 0]
[0 0 0]]
```

Notez que si on fait un tableau de flottants on obtiendra une notation différente:

Code : Python Console

```
>>> tableau_de_zero = np.zeros((2, 3), dtype='f')     # je crée le
même tableau en flottant
>>> print "mon tableau de zero :", tableau_de_zero
[[0. 0. 0.]
[0. 0. 0.]]
```

Notez la présence du "." après les zéros, il indique que les valeurs sont des flottants, mais vous le saviez déjà. 

Maintenant nous avons à notre disposition plusieurs alternatives pour créer des tableaux :

- np.ones() qui va créer un tableau plein de "1" du type choisi ;
- np.empty() qui va créer un tableau "vide" à savoir des nombres vaguement aléatoires ;
- np.indentity(x) qui va créer une matrice identité de taille $x * x$.

Et ces méthodes s'utilisent exactement de la même manière que np.zeros()

Création évoluée de tableau

On peut créer directement notre tableau avec des valeurs précises. Pour ce faire on va utiliser la fonction np.array() :

Code : Python Console

```
>>> tableau = np.array([[3, 2, 1], [4, 5, 6], [9, 8, 7]]) # je
spécifie le contenus des lignes une par une
>>> print tableau
[[3 2 1],
[4 5 6],
[9 8 7]]
```

tableau de dimension 3x3.

Si vous connaissez la fonction range(x, y, z) de Python qui renvoie une liste contenant des nombres de x jusqu'à y-1 par pas de z, sachez qu'il existe np.arange qui renvoie la même chose mais de type array.

On peut aisément convertir une liste en array via `tableau = np.array(list)` à condition bien sur qu'il n'y ai QUE des nombres dans votre list.
Le type de nombre contenu dans le tableau sera "le plus restrictif correspond à tout les nombres du tableaux": Si tout les nombre sont des entiers vous aurez un tableau d'entier, s'il y a au moins un flottant le tableau ne contiendra que des flottants.

De la même manière la fonction linspace, bien connu des utilisateurs d'IDL et Matlab existe : np.linspace(x, y, z) qui crée un tableau de z valeurs uniformément réparties de x à y.

Code : Python

```
tableau = np.linspace(0, 10, 4)
print tableau
[  0.        ,   3.33333333,   6.66666667,  10.        ]
```

4 valeurs (incluant les bornes) également réparti entre 0 et 10.

Et l'aléatoire? Il est bien évidement disponible. Il existe plusieurs fonction le permettant qui font appel à différentes lois de répartition. On les trouve dans le sous-module de numpy np.random
Regardons un tableau aléatoire généré par loi normale.

Code : Python Console

```
>>> tableau = np.random.nomal(0, 1, (3, 3)) # je crée un tableau de
dimension 3x3, suivant une loi normale centrée sur 0 avec une
dispersion de 1
>>> print a
[[-0.46245589 -1.72892904 0.41090444]
[ 1.16450418 0.82389124 0.74499673]
[ 0.55323832 -0.00792723 -0.81915122]]
```

Accédons à nos valeurs

Accédons maintenant aux valeurs de notre tableau.
On donne l'indice de la valeur qu'on veut changer, sa valeur, et on regarde le résultat :

Code : Python Console

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

En Python comme en C et contrairement au Fortran la numérotation des indices commence à 0.

On peut toujours accéder aux dimensions de notre tableau via les attributs de nos nd.array.

- mon_tabbleau.shape -> dimensions du tableau ;
- mon_tableau.size -> nombre d'éléments du tableau.

Code : Python Console

```
>>> tableau = np.zeros((2, 3), dtype='i')
>>> print tableau.shape
(2,3)
>>> print tableau.size
6
```

j'ai donc bien mes dimensions 2x3 et mes 6 éléments.

Manipulons nos tableaux

La manipulation des tableaux est très simple avec Python/Numpy. En effet un grand nombre de fonctions simples mais indispensables sont présentes sous forme de méthodes de notre objet array, comme le tri, l'aplatissement, le "reformage", etc.

Manipulation des valeurs

Opérations mathématiques de base

Un des intérêts majeurs des ndarray[] est que toute opération mathématique standard s'appliquera à toutes les valeurs du tableau.

Code : Python Console

```
>>> a = np.arange(5) # je crée un tableau de 5 valeurs équiréparties
commençant à 0.
[0 1 2 3 4]
>>> b = 2 * a        # je crée un second tableau qui contient les
valeurs du premier, multipliées par 2
>>> print b
[0 2 4 6 8]
>>> c = a + b        # je crée un troisième tableau dont les valeurs
sont la sommes des valeurs de a et b
>>> print c
[0 3 6 9 12]
>>> d = a / b        # je crée un quatrième tableau dont les valeurs
sont les quotients des valeurs de a et b
Warning: divide by zero encountered in divide
>>> print d
[0 0 0 0 0]
```

J'ai une erreur de division par 0, et ensuite je n'ai que des 0.
Normal ici mon ndarray[] ne contient que des entiers, il fait donc des divisions entières.

Code : Python Console

```
>>> a = np.arange(5.) # je crée un tableau de 5 valeurs
flottantes équiréparties commençant à 0.
>>> b = a*2.  # je multiplie ce tableau par un flottant
>>> print a
[0. 1. 2. 3. 4.]
>>> print a/b
Warning: invalid value encountered in divide
[ nan 0.5 0.5 0.5 0.5]
```

J'ai toujours une division par 0, mais j'ai maintenant des divisions euclidienne. Le "nan" est un "Not a Number".

Opérations mathématiques avancées

La bibliothèque standard de python contient un module "math" contenant des fonctions similaires mais qui seront incapables de traiter un array. Il est donc à proscrire.

Pour les opérateurs mathématiques plus évolués type sinus, logarithme, etc. on va préférer ceux fournis par numpy, afin qu'ils se comportent comme les opérateurs de base:

Code : Python Console

```
>>> a = np.arange(5) # je crée un tableau de 5 valeurs entières
équiréparties commençant à 0.
>>> b = np.exp(a) # je crée un second tableau contenant les
exponentielles des valeurs du premier tableau
>>> print b
[ 1. 2.71828183 7.3890561 20.08553692 54.59815003]
>>> c = 2 * np.sin(a) # je crée un troisième tableau contenant les
doubles des sinus des valeurs du premier tableau
[ 0. 1.68294197 1.81859485 0.28224002 -1.51360499]
```

Notez ici que les entiers sont devenus automatiquement des flottants via np.exp() et np.sinus(), heureusement. 

Manipulations générales sur les tableaux

On va utiliser les propriétés objets de Python/Numpy car ces opérations mathématiques sont disponibles directement dans notre objet ndarray() :

Extraire un minimum, un maximum, une moyenne, une somme

Code : Python Console

```
>>> mon_tableau = np.array([5, 3, 4, 1, 2]) # création de mon
tableau
```

```
>>> print mon_tableau.min()  # recherche de minimum
1
>>> print mon_tableau.max()  # recherche de maximum
5
>>> print mon_tableau.mean()  # calcul de moyenne
3.0
>>> print mon_tableau.sum()  # calcul de la somme des élément du
tableau
15
```

Et les dimensions supérieures à 1 dans l'histoire ?

Tout ça fonctionne aussi avec des tableaux de dimension supérieure à 1. Mais, là où on commence à profiter réellement de Numpy, c'est qu'on peut spécifier des "dimension intermédiaire", à savoir que vous voulez peut-être les minimums de chaque colonne, ou le minimum de tout le tableau, etc. C'est possible très simplement.
Reprenons pour l'exemple un tableau de nombres aléatoires:

Code : Python Console

```
>>> tableau = np.random.nomal(0, 1, (3, 3)) # je crée un tableau de
dimension 3x3, suivant une loi normale centrée sur 0 avec une
dispersion de 1
>>> print a
[[-0.46245589 -1.72892904 0.41090444]
 [ 1.16450418 0.82389124 0.74499673]
 [ 0.55323832 -0.00792723 -0.81915122]]
>>> print a.min()     # recherche de minimum global
-1.7289290356369671
>>> print a.min(0)     # recherche de minimum de chaque colonne
[-0.46245589 -1.72892904 -0.81915122]
>>> print a.min(1)     # recherche de minimum de chaque ligne
[-1.72892904 0.74499673 -0.81915122]
```

Et l'agèbre dans tout ça?

la multiplication de 2 ndarrays de même dimension donnera un ndarray de même dimension contenant les produits terme à terme (équivalent de ".*" de Matlab).
Le produit matriciel est accessible via np.dot().

Code : Python Console

```
>>> a=np.array([[1, 2], [3, 4]])
>>> b=np.ones((2, 2))
>>> print a * b
[[ 1. 2.]
 [ 3. 4.]]
>>> print np.dot(a, b)
[[ 3. 3.]
 [ 7. 7.]]
>>> print np.dot(b, a)
[[ 4. 6.]
 [ 4. 6.]]
```

De manière générale une opération mathématique type multiplication, addition, puissance, racine, etc. s'appliquera à chaque élément. Pour avoir l'opération type matricielle il faudra passer par une méthode.

Toute la doc de Numpy est disponible à cette adresse et est assez claire pour peu qu'on soit à l'aise avec de l'anglais

informatico-scientifique.

Il est à noter qu'il est possible de créer des objets "np.matrix" qui ont un comportement similaire à matlab, cependant toute opération matricielle étant disponible pour les np.array il vaut mieux préférer les np.array aux np.matrix.

Petit tour d'horizon de fonctions utiles :

Le tri :

via la fonction mon_tableau.sort()

Code : Python Console

```
>>> mon_tableau = np.array([5, 3, 4, 1, 2])
>>> print mon tableau
[5 3 4 1 2]
>>> mon_tableau.sort()
>>> print mon_tableau
[1 2 3 4 5]
```

Autre possibilité :

Code : Python Console

```
>>> print np.sort(mon_tableau)
[1 2 3 4 5]
>>> print mon_tableau
[5 3 4 1 2]
```

le comportement des 2 méthodes est différent.
mon_tableau.sort() va agir directement sur votre tableau et le trier, alors que la fonction np.sort() ne fait qu'afficher le tableau trié, sans le modifier.

La transposition de tableau :

(on parlera plutôt ici de matrice) via mon_tableau.transpose() ou mon_tableau.T :

Code : Python

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

La méthode transpose() retourne une version transposée de notre tableau, sans modifier le tableau d'origine : on peut donc l'utiliser directement au milieu d'un calcul, par exemple. Pour disposer de mon_tableau transposé je dois faire `mon_tableau_transpose = mon_tableau.T`

L'aplatissement :

via np.flat()

Code : Python Console

```
>>> tableau = np.zeros((2, 3), dtype='i')
>>> print tableau.flat[:]
[0 0 0 0 0 0]
```

Un peu d'explication : mon_tableau.flat correspond à mon_tableau ramené à un vecteur, pour l'afficher je dois donc spécifier [:]. Si je ne veux que la 3ème valeur de ce vecteur je donne [2] en paramètre à flat.

la méthode flat() ne fait que retourner une version applatie de notre tableau, elle ne le modifie pas.

Le "reformage" ou reformer un tableau :

via reshape()

On donne en argument les nouvelles dimensions de notre tableaux. Le nombre d'éléments total doit bien évidement rester constant

Code : Python

```
>>> tableau = np.zeros((2, 3),dtype='i')
>>> print np.size(tableau)
6
>>> tableau_2 = np.shape(tableau.reshape(3, 2))
(3,2)
>>> print np.size(tableau_2)
6
```