01_numpy_scipy_intro1

May 31, 2017

1 Librairies Python Numpy/Scipy

NumPy est une extension du langage de programmation Python, pour manipuler des matrices ou des tableaux multidimensionnels. Elle possède des fonctions mathématiques vectorielles opérant sur les tableaux.

NumPy est la base de la librairie SciPy, regroupement de librairies Python pour le calcul scientifique.

2 Tableau unidimensionnel

```
In [1]: import numpy as np
        a=np.arange(10)
        print(a)
```

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

Pour obtenir le type de l'objet et celui des données sous-jacentes

```
In [2]: print(type(a))
        print(a.dtype)
```

```
<class 'numpy.ndarray'>
int64
```

Pour créer un tableau de réels, il faut préciser le type explicitement

```
In [3]: import numpy as np
        a=np.arange(10, dtype='float')
        print(a)
```

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

La donnée membre shape retourne les dimensions du tableau sous la forme d'un tuple

```
In [4]: print(a.shape)
```

```
(10,)
```

Application des opérateurs de calcul arithmétique

```
In [5]: a=a+1      # ajout de 1 à tous les elements
        print(a)
        print(2*a)
        print(a*a) # carre de chaque element
        print(a/a) # division element par element  DANGER
```

```
[  1.   2.   3.   4.   5.   6.   7.   8.   9.  10.]
[  2.   4.   6.   8.  10.  12.  14.  16.  18.  20.]
[   1.    4.    9.   16.   25.   36.   49.   64.   81.  100.]
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

```
In [6]: a=np.array([0, 1, 2, 0, 2, 3, 5, 6, 6])
        print(a.shape)
        print(len(a))
        print('valeurs   : ', a.min(), ' ', a.max())
        print('positions : ', a.argmin(), ' ', a.argmax())
```

```
(9,)
9
valeurs   :  0   6
positions :  0   7
```

Pour transformer une liste python en numpy.ndarray, on appelle le constructeur ndarray ou la fonction membre asarray

```
In [7]: lst=list(range(10))
        print(lst)
        a=np.array(lst, dtype='float')
        print(a)
        a=np.asarray(lst, dtype='float') # plus efficace que le constructeur
        print(a)
```

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

3 Tableau multidimensionnel

But : création d'un tableau vide 2x2

```
In [8]: a=np.ndarray(shape=(2, 2)) # NON INITIALISE
        print(a)
        print('---------------')
        print(a.shape)
        print('---------------')
        print(a.ndim)
```

```
[[  0.00000000e+000   1.72723382e-077]
 [  2.16198396e-314   2.78136393e-309]]
---------------
(2, 2)
---------------
2
```

```
In [9]: a.fill(0)
        print(a)
```

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

```
In [10]: a=np.empty(shape=(2, 2), dtype='float64')
         print(a.dtype)
         print(a)
         print(type(a))
```

```
float64
[[ 0.  0.]
 [ 0.  0.]]
<class 'numpy.ndarray'>
```

bool Boolean (True or False) stored as a bit
inti Platform integer (normally either int32 or int64)
int8 Byte (-128 to 127)
int16 Integer (-32768 to 32767)
int32 Integer (-2^31 to 2^31 -1)
int64 Integer (-2^63 to 2 ^ 63 -1)
uint8 Unsigned integer (0 to 255)
uint16 Unsigned integer (0 to 65535)
uint32 Unsigned integer (0 to 2 ^ 32 - 1)
uint64 Unsigned integer (0 to 2 ^ 64 - 1)
float16 Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32 Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64 or float Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex64 Complex number, represented by two 32-bit floats (real and imaginary components)
complex128 or complex Complex number, represented by two 64-bit floats (real and imaginary components)

```
In [11]: a.fill(1)
         print(a)

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

3.1 vue d'un tableau

```
In [12]: a.fill(1)
         view=a  # ATTENTION shallow copy
         print('meme objet : ', id(a), ' ==', id(view))

         import copy as cp
         b=cp.deepcopy(a)
         print(id(b))

         c=a.copy()
         print(id(c))

         view[0, 0]=0  # modif de l'element [0, 0] de a !!!
         print(a)
         print(b)
         print(c)

meme objet :  4459667904  == 4459667904
4459667584
4459668704
[[ 0.  1.]
 [ 1.  1.]]
[[ 1.  1.]
 [ 1.  1.]]
[[ 1.  1.]
 [ 1.  1.]]
```

L'affectation ne fait qu'une copie superficielle ("shallow copy"), il faut utiliser le module copy avec sa fonction deepcopy pour dupliquer l'objet a avec ses valeurs.

3.2 cas exotique : tableau d'enregistrement

But : créer un tableau de structure (ou record)

```
In [13]: record=np.dtype([('name', str, 40), ('age', 'int32')])

In [14]: print(record)

[('name', '<U40'), ('age', '<i4')]
```

```
In [15]: item=np.array([('thomas', 17), ('nathan', 13)], dtype=record)
         print(item[0])

         with open('data', 'bw') as f:
             item.tofile(f)
         f.close

         with open('data', 'br') as f:
             a=np.fromfile(f, dtype=record)
         f.close
         print(a)
```

```
('thomas', 17)
[('thomas', 17) ('nathan', 13)]
```

4 Découpage et indexage

4.1 cas unidimensionnel

Le découpage (slicing) d'un tableau unidimensionnel NumPy est identique à celui d'une liste Python

```
In [16]: import numpy as np
         a=np.arange(10)
         print(a)
         # extraction des elements du tableau d'indice dans [3, 7)
         print(a[3:7])
          # extraction des elements avec un pas de 2 jusqu'a l'indice 8 exclu
         print(a[:8:2])
         # renverse le tableau
         print(a[::-1])
         print(a[-1::-1])
```

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

4.2 cas multidimensionnel

```
In [17]: # tableau unidimensionnel retransformé en tableau 2x3x4
         b=np.arange(24).reshape(2, 3, 4)
         print(b)
         print('--------------')
         print(b.shape)
```

```
         print('--------------')
         print(b.ndim)
```

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

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
--------------
(2, 3, 4)
--------------
3
```

On obtient 2 paquets, chacun formé d'une matrice 3x4

```
In [18]: # on selectionne l'element placé en [0, 0] de tous les paquets
         view=b[:, 0, 0]
         print(view)
         # dans le paquet 0, on selectionne la ligne 0
         view=b[0, 0, :]
         print(view)
         # dans le paquet 0, on selectionne dans la ligne 1 en se déplaçant de 2 en
         view=b[0, 1, ::2]
         print(view)
```

```
[ 0 12]
[0 1 2 3]
[4 6]
```

```
In [19]: view=b[..., 1] # on selectionne la colonne 1 de tous les paquets
         print(view)
```

```
[[ 1  5  9]
 [13 17 21]]
```

4.3 Exercice sur le slicing

Expliquer le comportement des cas suivants

```
In [20]: view=b[-1, ::-1, -1]
         print(view)
         view=b[0, ::-1, -1]
         print(view)
         view=b[0, :, -1]
         print(view)
```

```
[23 19 15]
[11  7  3]
[ 3  7 11]
```

5 Applatissage de tableaux (array flattening)

```
In [21]: b=np.arange(24).reshape(2, 3, 4)

         flat_nd_copy=b.flatten()
         print(flat_nd_copy)
         flat_nd_copy[0]=-1
         print(b)
         print('------------------------------')
         flat=b.ravel()  # nouvelle vue
         print(flat)

         flat[0]=-1  # modifie b
         print(b)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
------------------------------
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
[[[-1  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

6 Concaténation de tableaux

Soient 2 tableaux A et B, comportant le même nombre de ligne, on désire construire le tableau augmenté A | B

```
In [22]: A=np.arange(12).reshape(3, 4)
         B=A
         S=np.hstack((A, B))
         print(S)
```

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

Généralisation suivant les autres directions

```
In [23]: A=np.arange(12).reshape(3, 4)
         B=A
         S=np.vstack((A, B))
         print(S)
         print('------------------')
         S=np.dstack((A, B))
         print(S)
```

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

 [[ 4  4]
  [ 5  5]
  [ 6  6]
  [ 7  7]]

 [[ 8  8]
  [ 9  9]
  [10 10]
  [11 11]]]
```

6.1 zero-padding - rolling

```
In [24]: A=np.arange(12).reshape(3, 4)
         A=A+1
         print(A)
         print(type(A))
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
<class 'numpy.ndarray'>
```

```
In [25]: lx=A.shape[0]
         ly=A.shape[1]
         A=np.pad(A, ((0, lx), (0, ly)), mode='constant') # (before, after)
         print(A)
         print('----------------------------')
         A=np.roll(A, +ly//2, axis=1)
         print(A)
```

```
[[ 1  2  3  4  0  0  0  0]
 [ 5  6  7  8  0  0  0  0]
 [ 9 10 11 12  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]]
----------------------------
[[ 0  0  1  2  3  4  0  0]
 [ 0  0  5  6  7  8  0  0]
 [ 0  0  9 10 11 12  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]]
```

Application du zero padding dans le calcul de produit de convolution

7 création d'une matrice

La classe matrix dérive de la classe ndarray

```
In [26]: A=np.matrix([[0, 1], [2, 3]])
         print(type(A))

         M=np.asmatrix(np.diag([2, 1]))
         print(M)
         print(type(M))
```

```
<class 'numpy.matrixlib.defmatrix.matrix'>
[[2 0]
 [0 1]]
<class 'numpy.matrixlib.defmatrix.matrix'>
```

Création d'un vecteur

```
In [27]: V=np.matrix([0, 1]).transpose()
         print(V)
         print(A*V)
```

```
[[0]
 [1]]
[[1]
 [3]]
```

In [ ]: