

Les bases de NumPy

NumPy est une extension du langage de programmation Python, destinée à manipuler des matrices ou tableaux multidimensionnels ainsi que des fonctions mathématiques opérant sur ces tableaux.

NumPy permet la manipulations des vecteurs, matrices et polynômes.

Directive d'importation

- standard :

```
import numpy as np
```

Tableaux numériques

On convertit facilement des listes Python en tableau numpy. Essayez ceci :

```
import numpy as np
a = np.array([[1,2],[3,4]])
print(a)
print(a.dtype)
```

Sortie :

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

Pour définir un tableau, appelez simplement la fonction `.array` avec une liste ou un tuple. Des fonctions spéciales **zero**, **ones**, **rand** permettent d'initialiser à des valeurs particulières (0 ou 1), ou aléatoires.

Les fonctions `arange` et `shape` sont bien pratiques pour générer des nombres en séquences et réarranger des listes de nombres. La fonction `linspace` est utile parce qu'elle impose exactement le nombre de valeurs créées entre un minimum et un maximum.

Vous pouvez consulter [cette page](#) pour consulter d'autres fonctionnalités, ou [celle-ci](#), plus documentée.

[arrays_01.py](#)

```
#!/usr/bin/env python
```

```
# -*- coding: utf-8 -*-
"""
Divers codes à essayer pour créer des tableaux "array"
"""
import numpy as np

a = np.array(((1,2),(3,4))) # on peut créer un "array" à partir d'un
tuple
# afficher a, le nombre de dimensions, les dimensions, le type de
donnée
print(a, a.ndim, a.shape, a.dtype)
# avec des "floats" :
b = np.array([
    [1.1, 2.2, 3.3, 4.4],
    [5.5, 6.6, 7.7, 8.8],
    [9.9, 0.2, 1.3, 2.4],
])
print(b, b.ndim, b.shape, b.dtype)
# un tableau de zéros
c = np.zeros((4,2))
print(c, c.ndim, c.shape, c.dtype)
# un tableau tridimensionnel de 1 "complexe"
d = np.ones((2,3,4),dtype=complex)
print(d, d.ndim, d.shape, d.dtype)
# un tableau avec arange, et ensuite reshape
e1 = np.arange(1,36,1)
e = np.reshape(e1,(5,7))
print(e, e.ndim, e.shape, e.dtype)
f = np.random.rand(3,3)
print(f, f.ndim, f.shape, f.dtype)
# utilisation de linspace pour imposer le nombre d'éléments générés :
g = np.linspace(0.,np.pi,11)
print(g, g.ndim, g.shape, g.dtype)
```

Quelques manipulations élémentaires :

[arrays_02.py](#)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Divers codes à essayer pour manipuler des tableaux "array"
"""
import numpy as np

a = np.array([[1,2],[3,4]])
b = np.array([[1,1],[1,1]])
c = a + b # addition terme à terme
print(c, c.ndim, c.shape, c.dtype)
```

```
d = a * b # multiplication terme à terme
print(d, d.ndim, d.shape, d.dtype)
e = np.dot(a,b) # multiplication matricielle
print(e, e.ndim, e.shape, e.dtype)
f = np.sin(np.pi*0.5*a) # fonction mathématique et adaptation
automatique du type
print(f, f.ndim, f.shape, f.dtype)
g = np.transpose(a) # transposition
print(g, g.ndim, g.shape, g.dtype)
print(np.sum(a), np.min(a), np.max(a)) # somme des éléments, minimum,
maximum
```

Fonctions mathématiques principales :

- abs, sign, sqrt
- logarithmes/exponentielles : log, log10, exp
- trigonométriques et inverses : sin, cos, tan, arcsin, arccos, arctan
- hyperboliques et inverses : sinh, cosh, tanh, arcsinh, arccosh, arctanh
- entiers inférieur, supérieur ou le plus proche : floor, ceil, rint

Autres fonctions

- *min* et *max* rendent le minimum et le maximum, *argmin* et *argmax* rendent les indices de ces éléments dans un tableau 1D (consulter la [documentation](#) pour les dimensions supérieures).
- *sorted* : tri
- *clip* : cliping permettant d'éliminer des valeurs inférieures à une borne minimale donnée ou supérieures à une borne maximale
- *unique* : élimine les "doublons"
- fonctions booléennes, pour des conditions, ou pour filtrer suivant des conditions (voir la documentation)
- *copy* : copie d'un tableau (pour éviter les modifications lors d'utilisation directe ou par référence)
- *.tolist()* : convertit un tableau numpy en liste standard de python

Algèbre linéaire

[simple_linear_system.py](#)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Solve a system of simultaneous equation in two variables of the form
    2 * x + 7 * y = 17.
    3 * x - 5 * y = -21.

reference :
```

```
http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.solve.html
"""
#

import numpy as np
a = np.array([[2.,7.],[3.,-5.]]) # coefs matrice
b = np.array([[17.],[-21.]]) # independent coef vector
print(np.linalg.solve(a,b)) # solution
```

Quelques possibilités supplémentaires :

[arrays_linalg_03.py](#)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Divers codes à essayer pour de l'algèbre linéaire avec des tableaux
"array"
"""
import numpy as np

a = np.array([[1,2],[3,4]])
print(a, a.ndim, a.shape, a.dtype)
b = np.linalg.inv(a) # matrice inverse
print(b, b.ndim, b.shape, b.dtype)
unit = np.eye(2) # matrice unitaire
print(unit, unit.ndim, unit.shape, unit.dtype)
v = np.array([[10.], [14.]]) # vecteur colonne
x1 = np.dot(b,v) # multiplication de l'inverse de a par v
x2 = np.linalg.solve(a,v) # solution du système linéaire de
coefficients
# des inconnues a et de coefficients indépendants b
# les deux techniques donnent évidemment le même résultat !
print(x1, x1.ndim, x1.shape, x1.dtype)
print(x2, x2.ndim, x2.shape, x2.dtype)
# valeurs propres et vecteurs propres de matrices :
d = np.array([[1,1],[-1,1]])
print(np.linalg.eig(d))
```

Numpy dispose aussi d'une classe particulière de "arrays" pour des matrices.

Autres fonctions

- inner : produit scalaire (équivalent à dot sur des tableaux 1D)
- cross : produit vectoriel

- det : déterminant

Statistiques élémentaires

[arrays_stats_elem_04.py](#)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Divers codes à essayer pour des statistiques élémentaires sur des
tableaux "array"
"""
import numpy as np

a = np.array([1.,2.,3.5,5.,6.,7.,7.4,7.8,8.2,8.4,8.5,9.,10.2,12.5])
print(a, a.ndim, a.shape, a.dtype)
print("médiane = ", np.median(a))
print("moyenne = ", np.mean(a))
print("variance = ", np.var(a))
print("Écart-type = ", np.std(a))
```

Références complémentaires

- [How to do Descriptives Statistics in Python using Numpy](#)

Itérations sur les tableaux

[arrays_iteration_05.py](#)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
itérations sur des tableaux "array"
"""
import numpy as np

a = np.array([1.,2.,3.5,5.,6.,7.,7.4,7.8,8.2,8.4,8.5,9.,10.2,12.5])
for x in a:
    print(x)
# l'itération sur un tableau multidimensionnel se fait sur un premier
niveau de sous-listes
b = np.array([
    [1.1, 2.2, 3.3, 4.4],
    [5.5, 6.6, 7.7, 8.8],
    [9.9, 0.2, 1.3, 2.4],
```

```
    ])  
for x in b:  
    print(x)  
    for y in x:  
        print(y, ", ", ",")  
    print
```

Manipulation de polynômes

Une nouvelle bibliothèque [polynomial](#) devrait remplacer l'ancien "poly1d"

poly1d & polynomial ordonnent les coefficients en sens inverses !!!

```
<sxh python; title : arrays_polynomes_06.py> #!/usr/bin/env python # -*- coding: utf-8 -*- ""  
Utilisation de tableaux "array" pour des polynômes "" import numpy as np
```

```
# les coefficients du polynômes sont donnés par ordre décroissance des degrés dans poly1d  
a=np.poly1d([1.,2.,3.,4.]) # =  $x^3 + 2x^2 + 3x + 4$ 
```

```
print "polynôme : \n",a, type(a) # les coefficients de a : print "coefficients : ",a.coeffs # les racines de  
a : print "racines : ",a.roots # l'ordre du polynôme : print "ordre : ",a.order # évaluations sur un  
vecteur x=np.linspace(0,2.,21) print "x = ",x print "évaluation en x : ",np.polyval(a,x) # dérivation  
print "dérivée : \n",np.polyder(a) # intégration print "intégrale : \n",np.polyint(a) # création d'un  
polynôme par ses racines b=a.roots c=np.poly1d(b,True) print "Polynômes recrées par les racines  
:\n", c # fitting polynomial xd=np.array([0.,1.,2.,3.,4.,5.]) yd1=np.array([0.05,0.99,3.95,  
9.17,15.86,24.93]) pfit=np.poly1d(np.polyfit(xd,yd1,2)) print "fit d'une parabole (polynôme d'ordre 2)  
sur ces x et y :" print xd print yd1 print "polynôme de fit : \n",pfit </sxh>
```

Autres fonctions : voir [ici](#)

L'ordre des coefficients peut facilement être inversé par un slice avec les paramètres [::-1]

Transformées de Fourier

Le module de [transformée de Fourier discrète](#) de numpy comprend de nombreuses variantes, et les transformées peuvent aussi être effectuées via le [module équivalent fftpack de Scipy](#).

[fonctions-FT-04.py](#)

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
# graphes de fonctions et des transformées de Fourier, utilisant numpy  
# et matplotlib pour les graphes
```

```

import numpy as np    # directive d'importation standard de numpy
from numpy import fft # importation spécifique du module fft de numpy
import matplotlib.pyplot as plt
#from scipy import fftpack # directive d'importation standard du
#module équivalent de scipy
#
#https://docs.scipy.org/doc/scipy-0.15.1/reference/api.html#guidelines-for-importing-functions-from-scipy
#from pylab import * # directive d'importation alternative en mode
# "pylab" --> supprimer les plt., fft.,

def f1(t):
    f = np.sin(np.pi*2.*t)
    return f

def f2(t):
    f = np.exp(-t/2.)*np.cos(np.pi*2.*t)
    return f

def f3(t):
    f =
(4./np.pi)*(np.sin(np.pi*2.*t)+np.sin(np.pi*6.*t)/3.+np.sin(np.pi*10.*t)
)/5.+np.sin(np.pi*14.*t)/7.+np.sin(np.pi*18.*t)/9.)
    return f

# une TF peut se faire via :
# fft.fft() du fait de la directive from numpy import fft
# , ou np.fft.fft() du fait de import numpy as np
# , ou fftpack.fft(y1) si on utilise le module de scipy
x = np.arange(0.0,10.0,0.025)
y1 = f1(x)
z1 = fft.fft(y1)
w1 = np.abs(z1[:len(z1)//2])
y2 = f2(x)
z2 = fft.fft(y2)
w2 = np.abs(z2[:len(z2)//2])
y3 = f3(x)
z3 = fft.fft(y3)
w3 = np.abs(z3[:len(z3)//2])

# doc subplot :
#http://matplotlib.org/api/pyplot_api.html?highlight=subplot#matplotlib.pyplot.subplot
plt.subplot(3,2,1) # sous-graphes en 3 lignes et 2 colonnes, graphe 1
plt.title('Fonctions')
plt.plot(x,y1)
plt.xlabel("t/s")
plt.ylabel("A(t)")

plt.subplot(3,2,2) # sous-graphes en 3 lignes et 2 colonnes, graphe 2

```

```
plt.title(u'Transformées de Fourier')
plt.plot(w1)
plt.xlabel("f/Hz")
plt.ylabel("A(f)")

plt.subplot(3,2,3) # sous-graphes en 3 lignes et 2 colonnes, graphe 3
plt.plot(x,y2)
plt.xlabel("t/s")
plt.ylabel("A(t)")

plt.subplot(3,2,4) # sous-graphes en 3 lignes et 2 colonnes, graphe 4
plt.plot(w2)
plt.xlabel("f/Hz")
plt.ylabel("A(f)")

plt.subplot(3,2,5) # sous-graphes en 3 lignes et 2 colonnes, graphe 5
plt.plot(x,y3)
plt.xlabel("t/s")
plt.ylabel("A(t)")

plt.subplot(3,2,6) # sous-graphes en 3 lignes et 2 colonnes, graphe 6
plt.plot(w3)
plt.xlabel("f/Hz")
plt.ylabel("A(f)")

plt.savefig('fonctions-fft.png')
plt.show()
```

Figure obtenue :



Références

- [Site officiel](#)
- [NumPy reference](#)
- [Page Wikipédia](#)
- [Guide to NumPy](#)
- [Tutoriel via l'exemple du jeu de la vie \(+ ici\)](#)
- http://wiki.scipy.org/Tentative_NumPy_Tutorial
- [Introduction à Numpy, Scipy et Matplotlib](#)
- [NumPy: creating and manipulating numerical data](#), de Emmanuelle Gouillart, Didrik Pinte, Gaël Varoquaux, and Pauli Virtanen
- [Getting the Best Performance out of NumPy](#)

From:

<https://dvillers.umons.ac.be/wiki/> - **Didier Villers, UMONS - wiki**

Permanent link:

https://dvillers.umons.ac.be/wiki/teaching:progappchim:numpy_simple?rev=1490946096

Last update: **2017/03/31 09:41**

