Unité d'enseignement

*Mathématiques et informatique*

# Synthèse des séances TD « Mise à niveau des bases Python »

Cette session de synthèse met l'accent sur des points importants abordés dans les séances TD de « Mise à niveau des bases Python », avec en particulier les tableaux N-dimensionnels `ndarray` du module `numpy`.

La lecture de ce document pourra être complétée au besoin par celle du Quick start tutorial `numpy` qui propose une prise en main progressive des tableaux `ndarray`.

Jean-Luc.Charles@ensam.eu
Eric.Ducasse@ensam.eu

## ➜ Performance des calculs numpy utilisant la syntaxe vectorisée

Le module **numpy** définit le type **ndarray** : **tableau multi-dimensionnel de taille fixe contenant des valeurs homogènes** (plus de détails sur ℹThe N-dimensional array et ℹArray objects).

Les **performances** du module **numpy** viennent de l'usage intensif de **bibliothèques performantes** écrites en C pour la manupilation des objets **ndarray** : BLAS, LAPACK... Des versions optimisées de ces bibliothèques sont proposées par Intel (MKL : Math Kernel Library) et AMD (ACML : AMD Core Math Library)...

L'**écriture vectorisée** des expressions utilisant les tableaux **ndarray** donne accès aux performances de **numpy** :

```
1  from time import time
2  import numpy as np
3  M = np.random.randn(500, 5000)
4  V = np.random.randn(5000)
5  N, Tn, Tb = 20, [], []
6  m = '(moyenne de  calculs) :'.format(N)
7  print('Calcul du produit M.V ' + m)

8  #### N calculs numpy vectorisés :
9  for k in range(N):
10    t0 = time()  # top chrono
11    R  = M@V     # <==> R = M.dot(V)
12    Tn.append(time() - t0)
13 m = '\t[numpy vectorisé] : {:.4f} s'.format\
14        (np.mean(Tn))
15 print(m)
```

```
16 #### N calculs par boucle indexée :
17 for k in range(N):
18    t0 = time() # top chrono
19    R = np.zeros(M.shape[0])
20    for i in range(M.shape[0]):
22         for j in range(M.shape[1]):
23             R[i] += M[i,j]*V[j]
24    Tb.append(time() - t0)
25 m = '\t[boucle indexée ] : {:.4f} s'.format\
26        (np.mean(Tb))
27 print(m)

28 X = int(round(np.mean(Tb)/np.mean(Tn)))
29 m = '[numpy vectorisé] {} fois'.format(X)+\
30 ' plus rapide que [boucle indexée] !'
31 print(m)
```

```
matrice M : taille 500 X 5000 ; vecteur V : taille 5000
Calcul du produit M.V  (moyenne de 20 calculs) :
        [numpy vectorisé] : 0.0023 s
        [boucle indexée ] : 1.5282 s
[numpy vectorisé]  674 fois plus rapide que [boucle indexée] !
```

## ➜ Création des tableaux `ndarray`

Pour créer des tableaux **`ndarray`**, le module **`numpy`** propose des **fonctions** variées ℹ Array creation routines :

| *fonction* | *objet renvoyé par la fonction* |
|---|---|
| `array(objet)` | le résultat de la conversion de `objet` (liste, tuple...) en tableau **`ndarray`** |
| `arange(start,stop,step,...)` | tableau de valeurs espacées de `step` dans l'intervalle `[start, stop[` |
| `linspace(start,stop,nb,...)` | tableau de `nb` valeurs équi-réparties dans l'intervalle `[start, stop]` |
| `logspace(e1,e2,nb,...)` | tableau de `nb` valeurs en progression géométrique dans `[10**e1, 10**e2]` |
| `empty(shape,...)` | tableau non-initialisé, dimensions données par le tuple `shape` |
| `zeros(shape,...)` | tableau de zéros, dimensions données par le tuple `shape` |
| `ones(shape,...)` | tableau de uns, dimensions données par le tuple `shape` |
| `full(shape,val,...)` | tableau rempli avec `val`, dimensions données par le tuple `shape` |
| `empty_like(a,...)` | tableau non-initialisé, mêmes type et dimensions que le tableau `a` |
| `zeros_like(a,...)` | tableau de zéros, mêmes type et dimensions que le tableau `a` |
| `ones_like(a,...)` | tableau de uns, mêmes type et dimensions que le tableau `a` |
| `full_like(a,val,...)` | tableau rempli avec `val`, mêmes type et dimensions que le tableau `a` |
| `fromfunction(f,...)` | tableau des valeurs de la fonction `f` calculées avec les indices du tableau |
| `eye(...)` | tableau avec des 1 sur une diagonale choisie, et des 0 ailleurs |
| `identity(n,...)` | matrice identité d'ordre `n` |

## ➜ Exemple de création de tableaux `ndarray`

```
>>> import numpy as np
>>> M = np.array([[1,2,3],[4,5,6]]) ; M
array([[1, 2, 3],
       [4, 5, 6]])

>>> M = np.arange(1, 5, 0.5) ; M        # la borne supérieure est exclue...
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5])
>>> M = np.arange(1, 5+0.5/2, 0.5) ; M  # ajouter un 1/2 pas pour inclure la borne supérieure...
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ])
>>> M = np.linspace(1, 2, 5) ; M
array([ 1.  ,  1.25,  1.5 ,  1.75,  2.  ])
>>> M = np.logspace(1, 4, 4) ; M             # 4 valeurs entre 10**1 et 10**4
array([    10.,    100.,   1000.,  10000.])
>>> M = np.logspace(0, 6, 7, base=2) ; M     # 7 valeurs entre 2**0 et 2**6
array([  1.,   2.,   4.,   8.,  16.,  32.,  64.])

>>> M = np.empty(3) ; M
array([  6.91450234e-310,   2.28541724e-316,   1.58101007e-322])
>>> M = np.zeros(3) ; M
array([ 0.,  0.,  0.])
>>> M = np.ones(3) ; M
array([ 1.,  1.,  1.])
>>> M = np.ones(3, dtype=int) ; M
array([ 1,  1,  1])
>>> M = np.full((2,4), 1.2e3) ; M
array([[ 1200.,  1200.,  1200.,  1200.],
       [ 1200.,  1200.,  1200.,  1200.]])
```

## ➜ Exemple de création de tableaux ndarray – suite

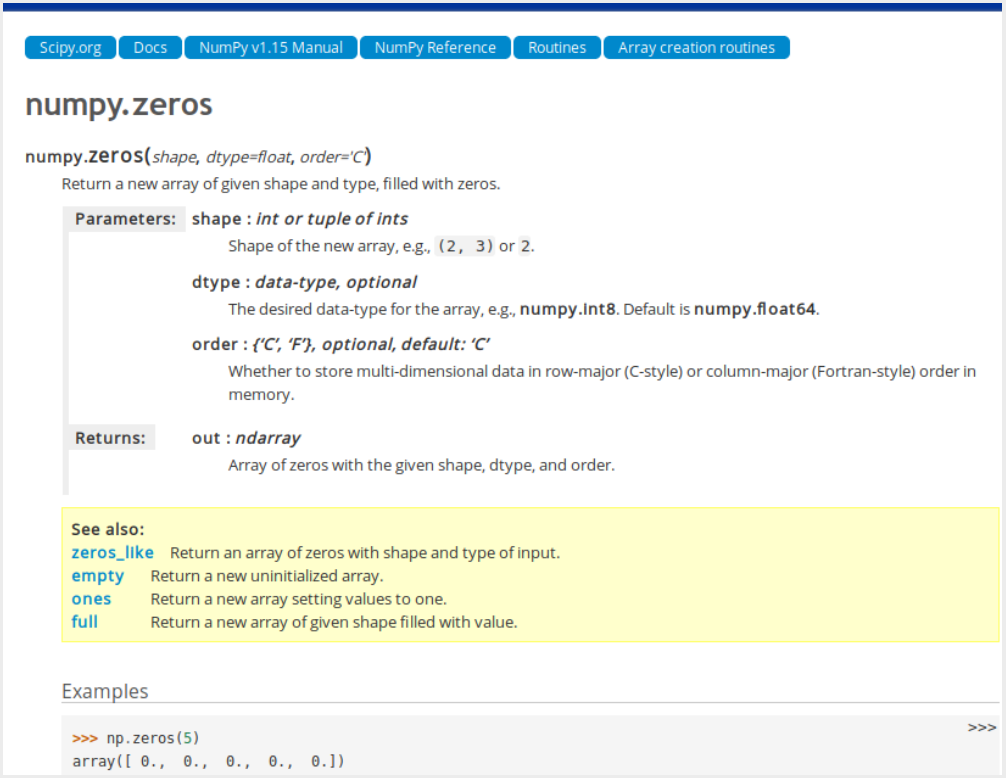
```
>>> def F(i,j): return 10*i + j
>>> M = np.fromfunction(F, (2,3), dtype=int) ; M     # avec la fonction F définie explicitement
array([[ 0,  1,  2],
       [10, 11, 12]])
>>> M = np.fromfunction(lambda i,j: 10*i+j, (2,3), dtype=int) ; M  # avec une fonction lambda
array([[ 0,  1,  2],
       [10, 11, 12]])

>>> M = np.eye(3) ; M
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> M = np.eye(3, k=-1, dtype=int) ; M
array([[0, 0, 0],
       [1, 0, 0],
       [0, 1, 0]])
>>> M = np.eye(3, 5, k=1, dtype=int) ; M
array([[0, 1, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 1, 0]])
>>> M = np.identity(3, dtype=int) ; M
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
```

## ➜ L'aide `numpy`

Sources d'aide : documentation **`numpy`** sur internet, aide en ligne dans le shell Python avec la fonction **`numpy.info`**.

mots clefs **`numpy`**, **`doc`** et **`zeros`** ⇝ page docs.scipy.org/doc/numpy/reference/generated/numpy.zeros.html



Scipy.org | Docs | NumPy v1.15 Manual | NumPy Reference | Routines | Array creation routines

### numpy.zeros

numpy.**zeros**(*shape*, *dtype=float*, *order='C'*)

Return a new array of given shape and type, filled with zeros.

**Parameters:** **shape : *int or tuple of ints***

Shape of the new array, e.g., `(2, 3)` or `2`.

**dtype : *data-type, optional***

The desired data-type for the array, e.g., **numpy.int8**. Default is **numpy.float64**.

**order : *{'C', 'F'}, optional, default: 'C'***

Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

**Returns:** **out : *ndarray***

Array of zeros with the given shape, dtype, and order.

**See also:**

| | |
|---|---|
| zeros_like | Return an array of zeros with shape and type of input. |
| empty | Return a new uninitialized array. |
| ones | Return a new array setting values to one. |
| full | Return a new array of given shape filled with value. |

Examples

```
>>> np.zeros(5)                                                    >>>
array([ 0.,  0.,  0.,  0.,  0.])
```

### Aide en ligne dans le shell Python

La fonction `numpy.info` affiche dans le shell Python des informations succinctes sur son argument :

```
>>> import numpy as np
>>> np.info(np.zeros)
zeros(shape, dtype=float, order='C')

Return a new array of given shape and type, filled with zeros.

Parameters
----------
shape : int or sequence of ints
    Shape of the new array, e.g., ``(2, 3)`` or ``2``.
...

Returns
-------
out : ndarray
    Array of zeros with the given shape, dtype, and order.

See Also
--------
zeros_like : Return an array of zeros with shape and type of input.
...

Examples
--------
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
...
```

## ➜ Les types numériques numpy

La plupart des fonctions `numpy` utilisées pour créer un objet `ndarray` acceptent un argument optionnel nommé `dtype` ℹ Data types, qui permet de spécifier le type numérique voulu (par défaut c'est le type `float`) :

### Exemples de valeurs possibles pour l'argument nommé `dtype=...`

| *type de base* | *type numpy* | *nom* | *description* |
|---|---|---|---|
| `bool` | `np.bool_` | `'bool'` | booléen `True` ou `False`, stocké sur un octet |
| `int` | `np.int_` | `'int'` | entier (`int32` ou `int64` selon plate-forme) |
| | `np.int8` | `'int8'` | entier sur 8 bit (octet) (-128 à 127) |
| | `np.int16` | `'int16'` | entier sur 16 bits (-32 768 à 32 767) |
| | `np.int32` | `'int32'` | entier sur 32 bits (-2 147 483 648 à 2 147 483 647) |
| | `np.int64` | `'int64'` | entier sur 64 bits (-9 223372 036854 775808 à 9 223372 036854 775807) |
| | `np.uint8` | `'uint8'` | entier non-signé sur 8 bits (0 à 255) |
| | `np.uint16` | `'uint16'` | entier non-signé sur 16 bits (0 à 65 535) |
| | `np.uint32` | `'uint32'` | entier non-signé sur 32 bits (0 à 4 294 967 295) |
| | `np.uint64` | `'uint64'` | entier non-signé sur 64 bits (0 à 18 446744 073709 551615) |
| `float` | `nb.float_` | `'float'` | flottant, équivalent de `np.float64` |
| | `np.float16` | `'float16'` | flottant demi-précision |
| | `np.float32` | `'float32'` | flottant simple précision |
| | `np.float64` | `'float64'` | flottant double précision |
| `complex` | `np.complex_` | `'complex'` | complexe, équivalent de `np.complexe128` |
| | `np.complex64` | `'complex64'` | complexe (2 flottants de 32 bits) |
| | `np.complex128` | `'complex128'` | complexe (2 flottants de 64 bits) |

## ➜ Les attributs de la classe `ndarray`

La classe **`ndarray`** possède plusieurs attributs qu'il faut savoir utiliser ℹ Array attributes :

### Principaux attributs de la classe `ndarray`

| *attribut* | *lecture* | *écriture* | *description* |
|---|---|---|---|
| **`dtype`** | ✓ | ✓ | type des données (modification possible mais **déconseillée** : *cf* méthode `astype`) |
| **`shape`** | ✓ | ✓ | tuple donnant les valeurs des dimensions du tableau |
| **`ndim`** | ✓ | ✗ | nombre de dimensions (`ndim` est égal à `len(shape)`) |
| **`size`** | ✓ | ✗ | nombre total d'éléments du tableau |
| **`itemsize`** | ✓ | ✗ | taille (en octets) d'un élément du tableau |
| **`nbytes`** | ✓ | ✗ | nombre d'octets en mémoire occupés par le tableau (`itemsize*size`) |
| **`real`** | ✓ | ✓ | tableau des parties réelles |
| **`imag`** | ✓ | ✓ | tableau des parties imaginaires |

L'attribut **`shape`** par exemple permet de changer la structure d'un tableau :

```
>>> import numpy as np
# liste de listes convertie en un tableau 2D :
>>> M = np.array([[1,2,3],[4,5,6]])
>>> M
array([[1, 2, 3],
       [4, 5, 6]])
>>> M.shape
(2,3)
```

```
>>> M.shape = (3,2)
>>> M
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> M.shape = 6
>>> M
array([1, 2, 3, 4, 5, 6])
```

## ➜ Les méthodes de la classe `ndarray`

La classe **`ndarray`** propose de très nombreuses méthodes pour le calcul, l'algèbre linéaire, la manipulation des données...

### Quelques méthodes de la classe `ndarray`

| *nom de la méthode* | *action* |
|---|---|
| `tolist()` | renvoie une liste (ou des listes emboîtées) contenant les données du tableau |
| `fill(val)` | remplit le tableau avec la valeur `val` |
| `resize(shape)` | change la structure du tableau (même effet que de modifier l'attribut **`shape`**) |
| `transpose()` | renvoie une vue du tableau transposé |
| `flatten()` | renvoie une copie du tableau « mis à plat » (1 seule dimension) |
| `copy()` | renvoie une copie (*deep copy*) du tableau |
| `astype(new_dtype)` | renvoie une copie du tableau en changeant le type des éléments comme demandé |
| `sort()` | trie le tableau « en place » |
| `take(indices)` | renvoie un tableau formé des éléments trouvés aux `indices` donnés |
| `nonzero()` | renvoie les indices des éléments non nuls du tableau |
| `min(axe=None)` | renvoie la valeur minimale du tableau (ou selon un **`axe`**) |
| `max(axe=None)` | renvoie la valeur maximale du tableau (ou selon un **`axe`**) |
| `mean(axe=None)` | renvoie la moyenne des éléments du tableau (ou selon un **`axe`**) |
| `std(axe=None)` | renvoie l'écart-type $\sigma_n$ des éléments du tableau (ou selon un **`axe`**) |
| `std(ddof=1,axe=None)` | renvoie l'écart-type $\sigma_{n-1}$ des éléments du tableau (ou selon un **`axe`**) |

Consulter la page ℹ Array methods pour une revue plus complète des méthodes de la classe **`ndarray`**.

## ➜ Expressions booléennes utilisant des tableaux `ndarray`

Les opérateurs de comparaison (`<`, `<=`, `>`, `>=`, `==`, `!=`) renvoient des tableaux `ndarray` de booléens, avec lesquels les opérations `*` et `+` correspondent respectivement à des opérations terme à terme AND et OR.
Ceci permet un style d'écriture très compact, puissant et performant :

```
>>> import numpy as np
>>> M = np.array([[1,2,3],[4,5,6]]) ; M
array([[1, 2, 3],
       [4, 5, 6]])

>>> M >= 3           # opération possible avec un ndarray
array([[False, False,  True],
       [ True,  True,  True]], dtype=bool)

>>> (M >= 3)*1       # la multiplication par 1 force la conversion booléen -> numérique
                     # /!\ Parenthèses indispensables pour forcer l'ordre des opérations
array([[0, 0, 1],
       [1, 1, 1]])

>>> (M >= 3)*(M <= 4)*1   # la multiplication '*' a ici le même sens qu'un AND terme à terme
array([[0, 0, 1],
    [1, 0, 0]])

>>> (M == 1)+(M ==6)*1
array([[1, 0, 0],
       [0, 0, 1]])

>>> M * (M >= 4)
array([[0, 0, 0],
    [4, 5, 6]])

>>> M * (M % 2 == 0)
array([[0, 2, 0],
       [4, 0, 6]])
```