# Bibliothèque numpy
# Une introduction aux tableaux

Dans tout ce qui suit, on suppose que notre script comporte l'instruction :

```
from numpy import *
```

Pour vraiment creuser la question, je vous recommande la lecture du tutoriel suivant qui m'a servi de base de travail :

http://wiki.scipy.org/Tentative_NumPy_Tutorial#head-6a1bc005bd80e1b19f812e1e64e0d25d50f99fe2

## ATTRIBUTS

Soit `T` un tableau `numpy`. C'est une instance de la classe `ndarray` (mais on utilise couramment l'alias `array` dans les programmes Python).

Les attributs de `T` importants pour nous sont :

- `T.ndim` : le nombre de dimensions (ou axes) du tableau `T`.

  Si, par exemple, on souhaite représenter une matrice complexe $3\times7$ à l'aide d'un tableau `numpy`, celui-ci comportera 2 dimensions.
  Les tableaux `T1` et `T2` de la partie suivante comportent respectivement 1 et 2 dimensions.

- `T.shape` : les dimensions proprement dites du tableau `T`.

  Il s'agit d'un tuple comportant autant d'éléments qu'il y a de dimensions dans le tableau. Chaque élément du tuple correspond à la valeur de la dimension.
  Par exemple, `T2.shape` renverra le tuple : `(2,3)`.

- `T.size` : le nombre total d'éléments du tableau `T`.

  C'est le produit des dimensions (i.e. le produit des éléments du tuple renvoyé par `shape`).

- `T.dtype` : le type des éléments du tableau.

  Par exemple : `T2.dtype.name` renverra `'int32'`.

# CREATION

On peut créer un tableau numpy de diverses façons.

## Création explicite

On fera attention aux crochets !

```
T1 = array([1,2,3])
T2 = array([[1,2,3],[4,5,6]])
```

Le tableau T1 correspond à la matrice ligne : $\begin{pmatrix}1 & 2 & 3\end{pmatrix}$ et le tableau T2 à la matrice de $\mathcal{M}_{2\times 3}(\mathbb{R})$ : $\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix}$.

## Création avec spécification du type

Si on saisit : `T2 = array([[1,2,3],[4,5,6]],dtype=float64)`, l'instruction T2 dans la ligne de commande renverra :

```
array([[ 1., 2., 3.],
       [ 4., 5., 6.]])
```

Les éléments de T2 sont ainsi des flottants en précision 64 bits.

## Tableaux de 0, de 1, identités ou vides

**Les fonctions ci-après créent des tableaux dont les éléments sont, par défaut, des flottants.**

On peut créer un tableau rempli de 0 (resp. 1) à l'aide de la fonction zeros (resp. ones) :

```
Z = zeros( (3,7) )
U = ones( (6,2) , dtype = int16)
```

En saisissant Z et U sur la ligne de commande, on obtiendra :

```
array([[ 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0.]])

array([[1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1]], dtype=int16)
```

Pour créer la matrice identité d'ordre *n*, on utilise la commande `identity` avec *n* passé en argument. Par exemple :

```
I = identity(5)
```

En saisissant alors `I` sur la ligne de commande, on obtiendra l'affichage :

```
array([[ 1., 0., 0., 0., 0.],
       [ 0., 1., 0., 0., 0.],
       [ 0., 0., 1., 0., 0.],
       [ 0., 0., 0., 1., 0.],
       [ 0., 0., 0., 0., 1.]])
```

Plus généralement, on peut utiliser la fonction `eye`. Ainsi `eye(3,5)` donnera :

```
array([[ 1., 0., 0., 0., 0.],
       [ 0., 1., 0., 0., 0.],
       [ 0., 0., 1., 0., 0.]])
```

et `eye(6,3)` :

```
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.],
       [ 0., 0., 0.],
       [ 0., 0., 0.],
       [ 0., 0., 0.]])
```

Enfin, pour créer un tableau « vide », on utilise la commande `empty`. Par exemple :

```
E = empty( (3,3) )
```

En saisissant E sur la ligne de commande, on obtiendra un affichage (étonnant … non ?) du type :

```
array([[ 7.06740692e-294, 7.06778912e-294, 7.06817139e-294],
       [ 6.71897251e-294, 6.71935478e-294, 1.37601456e-275],
       [ 1.49157461e+214, 2.64624211e-260, 1.00000000e+000]])
```

### Avec la méthode copy

`T.copy` : crée une copie du tableau `T`.
Par exemple : `T2 = T.copy()` crée une copie du tableau `T` ayant pour nom `T2`.

### Tableaux définis à l'aide de séries de nombres

On dispose de deux fonctions essentielles : `arange` et `linspace`.

`arange` est aux tableaux `numpy` ce que `range` est à Python !

Considérons par exemple l'instruction :

```
S1 = arange(2,55,3).reshape(3,6)
```

`arange(2,55,3)` seule créerait un tableau comportant une seule dimension et contenant les 18 éléments : `2, 5, 8, 11, 14, …, 50, 53` (attention ! on s'arrête au premier entier de la forme $2+3n$ strictement inférieur à 55). En ajoutant `reshape(3,6)`, on réorganise les 18 éléments de sorte qu'ils soient les éléments d'un tableau à deux dimensions (3 et 6). En saisissant `S1` sur la ligne de commande, on obtiendra alors :

```
array([[ 2,  5,  8, 11, 14, 17],
       [20, 23, 26, 29, 32, 35],
       [38, 41, 44, 47, 50, 53]])
```

`linspace` est le pendant de `arange` lorsque l'on connaît le nombre d'éléments de la série de nombres souhaitée et que le pas (souvent un flottant) ne peut être précisé simplement. Cette fonction est très utile quand on souhaite, par exemple, une subdivision de pas constant d'un intervalle :

```
x = linspace(0,pi,501)
```

on obtient ainsi un tableau à une dimension de 501 éléments contenant les réels de la forme $\frac{k\pi}{500}$ pour $k \in \{0, 1, 2, 3, ..., 500\}$.
Si on écrit alors `y = exp(x)`, on obtient un nouveau tableau de 501 éléments contenant les réels de la forme $\exp\left(\frac{k\pi}{500}\right)$ pour $k \in \{0, 1, 2, 3, ..., 500\}$ …

Ce « comportement » est à la fois puissant et simple d'emploi. On peut utiliser de la sorte les fonctions `sin`, `cos`, l'exponentiation (`**`), … On parle alors de « universal function » (`ufunc`).

## Tableaux diagonaux

On peut créer des tableaux diagonaux à l'aide de la fonction `diag` qui accepte des syntaxes diverses :

Les instructions :

```
L=[1,2.5,66]
diag(L)
```

renverront :

```
array([[ 1. , 0. , 0. ],
       [ 0. , 2.5, 0. ],
       [ 0. , 0. , 66. ]])
```

On notera que la présence du flottant `2.5` dans la liste L a conduit à une conversion des autres éléments du tableau en flottants.

L'instruction :

```
diag(arange(6))
```

renverra :

```
array([[0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0],
       [0, 0, 2, 0, 0, 0],
       [0, 0, 0, 3, 0, 0],
       [0, 0, 0, 0, 4, 0],
       [0, 0, 0, 0, 0, 5]])
```

L'instruction :

```
diag(linspace(0,pi,5))
```

renverra :

```
array([[ 0.        , 0.        , 0.        , 0.        , 0.        ],
       [ 0.        , 0.78539816, 0.        , 0.        , 0.        ],
       [ 0.        , 0.        , 1.57079633, 0.        , 0.        ],
       [ 0.        , 0.        , 0.        , 2.35619449, 0.        ],
       [ 0.        , 0.        , 0.        , 0.        , 3.14159265]])
```

## ACCES AUX ELEMENTS

### Syntaxes fondamentales

On retrouve ici des syntaxes similaires à celle de Python pour les listes (par exemple) et autres séquences. On doit utiliser les crochets et autant de valeurs d'indice qu'il y a de dimensions.

Considérons le tableau `S1` ci-dessus.

`S1[1,3]` renverra `29`.

Pour la suite, on retrouve le principe du « slicing » des séquences de Python :

`S1[1:3,4]` renverra `array([32, 50])`, c'est-à-dire le tableau des éléments correspondant à un indice de première dimension égal à 1 puis 2 et à un indice de colonne égal à 4. Les éléments de ce nouveau tableau sont donc `S1[1,4]` et `S1[2,4]`.

`S1[:,3]` renverra le quatrième élément dans la deuxième dimension pour chaque valeur de l'indice de la première dimension (c'est-à-dire, la quatrième colonne du tableau vu comme une matrice !) : `array([11, 29, 47])`.

`S1[1:,:]` renverra pour chaque indice de la première dimension à partir de 1, tous les éléments selon la deuxième dimension (c'est-à-dire, la deuxième et la troisième ligne du tableau vu comme une matrice !) :

```
array([[20, 23, 26, 29, 32, 35],
       [38, 41, 44, 47, 50, 53]])
```

## Omission d'indices

A la place de `S1[1:,:]`, on peut écrire `S1[1:]`. Dans ce cas, un remplacement automatique a lieu : l'absence du deuxième indice (ou de la plage de variation du deuxième indice) est remplacée par « `:` ».

Une troisième syntaxe est disponible : `S1[1:,…]`. On l'utilisera plutôt lorsque le tableau comporte 3 (ou plus !) indices.

## Obtention d'une diagonale

Ici, on utilise encore la syntaxe `diag`.

Considérons par exemple le tableau `T` défini par :

```
T=arange(36).reshape(6,6)
```

c'est-à-dire :

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
```

La commande `diag(T)` donne alors le tableau :

```
array([  0,  7, 14, 21, 28, 35])
```

En fait, la fonction `diag` accepte un deuxième argument qui précise l'indice de la diagonale souhaitée. Ci-dessus, cet indice a été omis. Sa valeur par défaut est égale à 0 et nous avons ainsi eu accès à la diagonale principale du tableau considéré. Si l'indice fourni est positif (respectivement négatif), on aura accès à une diagonale située au-dessus (respectivement en dessous) de la diagonale principale. Ainsi, les commandes `diag(T,2)` et `diag(T,-1)`, fournissent les tableaux :

```
array([  2,  9, 16, 23])
```

et

```
array([  6, 13, 20, 27, 34])
```

## Quelques méthodes et fonctions utiles

Dans cette partie, A est un tableau numpy.

- Méthode **copy** :

  Permet d'obtenir une copie du tableau.
  On obtiendra une copie de A avec une instruction comme :

  ```
  B=A.copy()
  ```

- Méthode **astype** :

  Permet de copier le tableau en changeant le type de ses éléments.
  Par exemple, l'instruction `I=identity(5)` nous donne la matrice identité d'ordre 5 ayant des coefficients de type `float`.
  Si l'on souhaite changer le type en `int32` (type propre à `numpy`), on pourra utiliser l'instruction `J=I.astype(int32)` qui crée le tableau d'entiers 32 bits :

  ```
  array([[1, 0, 0, 0, 0],
         [0, 1, 0, 0, 0],
         [0, 0, 1, 0, 0],
         [0, 0, 0, 1, 0],
         [0, 0, 0, 0, 1]])
  ```

- Fonction **transpose**

  Comme son nom l'indique, cette fonction renvoie le tableau transposé du tableau passé en argument.
  Par exemple, si le tableau A est le tableau :

  ```
  array([[ 3., 2., -1., 15.],
         [ 2., 3., 1., 10.],
         [ 1., 1., 1., 6.]])
  ```

  L'instruction `B=transpose(A)` donne le tableau B suivant :

  ```
  array([[ 3., 2., 1.],
         [ 2., 3., 1.],
         [ -1., 1., 1.],
         [ 15., 10., 6.]])
  ```

- Fonction **concatenate**

  Elle permet d'accoler deux tableaux pour en fabriquer un troisième.
  Par exemple, si A est le tableau :

  ```
  array([[ 3., 2., -1., 15.],
         [ 2., 3., 1., 10.],
         [ 1., 1., 1., 6.]])
  ```

et si B est le tableau :

```
array([[ 7., 0., 31., 2.],
       [ -4., 2., -5., 1.],
       [ 3., -2., 77., 2.]])
```

alors l'instruction `C=concatenate((A,B))` permet de construire le tableau `C` :

```
array([[ 3., 2., -1., 15.],
       [ 2., 3., 1., 10.],
       [ 1., 1., 1., 6.]],
       [ 7., 0., 31., 2.],
       [ -4., 2., -5., 1.],
       [ 3., -2., 77., 2.]])
```

Note : on obtient le même résultat avec `C=concatenate((A,B),axis=0)`. La valeur 0 est donc la valeur par défaut de l'argument `axis` pour cette fonction.

Cette première concaténation s'est effectuée par les lignes, les deux tableaux devaient donc avoir le même nombre de colonnes.

Comme nos tableaux A et B ont le même nombre de lignes, on peut aussi les concaténer suivant les colonnes via l'instruction :

```
C=concatenate((A,B),axis=1)
```

On obtient cette fois le tableau :

```
array([[ 3., 2., -1., 15., 7., 0., 31., 2.],
       [ 2., 3., 1., 10., -4., 2., -5., 1.],
       [ 1., 1., 1., 6., 3., -2., 77., 2.]])
```