L1 - Calcul Scientifique

Youssef Chahir
youssef.chahir@unicaen.fr

Chapitre II : NumPy
Manipulation des tableaux pour le calcul numérique

— Nombreuses fonctions de manipulation de tableaux
— Bibliothèque mathématique importante

NumPy : Wikipedia

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. Plus précisément, cette bibliothèque logicielle open source fournit de multiples fonctions permettant notamment de créer directement un tableau depuis un fichier ou au contraire de **sauvegarder un tableau dans un fichie**r, et manipuler des vecteurs, matrices et polynômes. NumPy est la base de SciPy, regroupement de bibliothèques Python autour du calcul scientifique .

Introduction

- Le module NumPy permet la manipulation simple et efficace des tableaux
- Représentés sous forme de tableau à 1 dimension (vecteur), 2 dimensions (matrices) ou plus.
- Permet de faire globalement des traitements sur l'ensemble des valeurs sans avoir à faire de boucles
- Les opérations sont exécutées rapidement (la librairie étant écrite en langage C) même si elle est utilisée à partir de programmes en Python
- Librairie de base pour d'autres librairies : optimisation, calcul symbolique, etc.
- Il faut au départ importer le package **numpy** :
  - **import numpy as np**
- Installation :
  - https://numpy.org/install/

Tableaux : numpy.array().

- L'objet array permet de représenter des tableaux multidimensionnels. On utilise des crochets pour délimiter les listes d'éléments dans les tableaux.

Attribute	Description
shape	A tuple that contains the number of elements (i.e., the length) for each dimension (axis) of the array.
size	The total number of elements in the array.
ndim	Number of dimensions (axes).
nbytes	Number of bytes used to store the data.
dtype	The data type of the elements in the array.

- Exemple :
  - tab = np.array([[1, 2], [3, 4], [5, 6]])
  - tab = np.array([[1, 2], [3, 4], [5, 6]], dtype=np.int)

Création des arrays

Attribute	Variants	Description
int	int8, int16, int32, int64	Integers.
uint	uint8, uint16, uint32, uint64	Unsigned (non-negative) integers. Boolean (True or False).
bool	bool	Floating-point numbers. Complex-valued floating-point numbers.
float	float16, float32, float64, float128	Integers.
complex	complex64, complex128, complex256	Unsigned (non-negative) integers. Boolean (True or False).

- Permet un contrôle fin de l'occupation mémoire et de la précision de la représentation
- Une fois le tableau créé, on peut changer de type :
  - Copie : tab = np.array(tab, dtype=np.int)
  - Changement de type : tab = tab.astype(np.float) ou tab.astype(float)

Création de tableau

- Création d'un array simple
  - A partir d'une liste python : tab = np.array([1, 2, 3.5])
  - A partir d'un tuple: tab = np.array((1, 2, 3.5))
- Copie indépendante :
  - b = np.copy(a) : renvoie une copie indépendante de l'array de départ.
  - b = np.array(a), b est une copie de a (si a changé, b ne l'est pas).
  - b=a.astype(int) b est une copie par conversion de « data type »
- Copie dépendante : b = np.asarray(a), b pointe vers la même array que a (si a modifiée, b l'est aussi).
- Création d'un array à plusieurs dimensions
  - A partir d'une liste de listes :
    - tab = np.array([[1, 2, 3], [4, 5, 6]])
    - Les valeurs sont par lignes
  - Accès aux éléments : index de ligne, puis index de colonne.
    - Exemple : tab[0, 1] donne ici 2.

Attributs d'un tableau

- Type d'une array : tab.dtype
- Accès à un élément d'indice i : tab[i].
  - Donne une valeur du même type que le type de l'array (type numpy).
  - Attention, si on veut un type python, il faut le convertir : int(a[0]) par exemple.
- Accès aux éléments : index de ligne, puis index de colonne.
  - Exemple : tab[0, 1] donne ici 2.
- Nombre d'éléments (tab.size): donne la taille totale d'une array numpy (le produit des tailles des différentes dimensions).
  - np.size(tab) affiche 6 et np.size(np.array([2,5,6,8])) affiche 4
- Dimension de l'array (tab.shape): renvoie les dimensions de l'array (cad taille du tableau) , d'abord le nombre de lignes, puis le nombre de colonnes,
  - np.shape(np.array([2,5,6,8])) affiche (4,) et np.shape(tab) affiche *(2, 3)*
  - On distingue bien ici que les deux tableaux correspondent à des tableaux 1D et 2D, respectivement.
  - ndim renvoie (1 pour un vecteur et 2 pour une matrice)
- **Remarque** : Les fonctions shape , size , et ndim peuvent être évoquées de 2 manières :
  - tab.shape **ou** np.shape(tab)

Tableaux spécifiques

- Tableaux constants
  - zeros permet de créer des tableaux dont tous les coefficients sont nuls.
  - ones permet de créer des tableaux dont tous les coefficients valent 1.
  - zeros_like et ones_like : forment un tableau constant (valeurs 0 ou 1) ayant le même format que le tableau passé en argument.
  - La fonction fill remplit un tableau par une constante
    - a = np.zeros((2, 3), dtype = int) : a : création d'une array 2 x 3 avec que des zéros. Si type non précisé, c'est float.
    - b = np.zeros_like(a) : création d'une array de même taille et type que celle donnée, et avec que des zéros.
    - b = np.zeros_like(a, dtype = float) : l'array est de même taille, mais on impose un type.
  - zeros np.empty, np.empty_like: a = np.empty((2, 3), dtype = float) : création d'une array 2x 3 de floats non initialisée.
  - np.full_like: tab = np.array([[1, 2, 3], [4, 5, 6]]) a=np.full_like(tab,1) donne array([[1, 1, 1],[1, 1, 1]])

Création de tableaux

Function name	Type of array
np.array	Creates an array for which the elements are given by an array-like object, which, for example, can be a (nested) Python list, a tuple, an iterable sequence, or another ndarray instance.
np.zeros	Creates an array – with the specified dimensions and data type – that is filled with zeros.
np.ones	Creates an array – with the specified dimensions and data type – that is filled with ones.
np.diag	Creates a diagonal array with specified values along the diagonal, and zeros elsewhere.
np.arange	Creates an array with evenly spaced values between specified start, end, and increment values.
np.linspace	Creates an array with evenly spaced values between specified start and end values, using a specified number of elements.
np.logspace	Creates an array with values that are logarithmically spaced between the given start and end values.
np.meshgrid	Generate coordinate matrices (and higher-dimensional coordinate arrays) from one- dimensional coordinate vectors.
np.fromfunction	Create an array and fill it with values specified by a given function, which is evaluated for each combination of indices for the given array size.
np.fromfile	Create an array with the data from a binary (or text) file. NumPy also provides a corresponding function np.tofile with which NumPy arrays can be stored to disk, and later read back using np.fromfile.
np.genfromtxt, np.loadtxt	Creates an array from data read from a text file. For example, a comma-separated value (CSV) file. The function np.genfromtxt also supports data files with missing values.
np.random.rand	Generates an array with random numbers that are uniformly distributed between 0 and 1. Other types of distributions are also available in the np.random module.

Tableaux spécifiques (suite)

- Matrices communes :
  - np.identity(n) : la matrice identité d'ordre n (float par défaut, sinon préciser dtype : np.identity(3, dtype = int).
  - np.eye(3, 2) : matrice 3 x 2 avec des 1 sur la diagonale et des 0 ailleurs, en float par défaut (sinon, utiliser dtype pour le type) :

```
: np.eye(3, 2)
array([[1., 0.],
       [0., 1.],
       [0., 0.]])
```

```
: a = np.array([[1,5,3],[2,7,4],[9,8,0]]); a
array([[1, 5, 3],
       [2, 7, 4],
       [9, 8, 0]])
```

  - np.diag/np.diagonal renvoie la diagonale d'une matrice. Avec un deuxième argument (facultatif) k, on obtient une surdiagonale (si k > 0) ou une sous-diagonale (si k < 0). a.diagonal()

```
: np.diag(a,-2)
array([9])
```

```
: np.diag(a,-1)
array([2, 8])
```

```
: np.diag(a)
array([1, 7, 0])
```

```
: np.diag(a,1)
array([5, 4])
```

```
: np.diag(a
array([3])
```

Indexation et découpage (slicing)

Tableau unidimensionnels : syntaxe [ ], identique listes Python ; indexes négatifs : à partir de la fin (-1 dernier, -2 avant dernier, etc.). Possibilité utiliser t-uple.

Expression	Description
a[m]	Select element at index m, where m is an integer (start counting form 0).
a[-m]	Select the mth element from the end of the list, where m is an integer.
a[m:n]	The last element in the list is addressed as -1, the second-to-last element as -2, and so on.
a[:] or a[0:-1]	Select elements with index starting at m and ending at n – 1 (m and n are integers). Select all elements in the given axis.
a[:n]	Select elements starting with index 0 and going up to index n – 1 (integer).
a[m:] or a[m:-1]	Select elements starting with index m (integer) and going up to the last element in the array.
a[m:n:p]	Select elements with index m through n (exclusive), with increment p.
a[::-1]	Select all the elements, in reverse order.

Lecture/Ecriture dans un tableau

- Lecture de valeurs dans un vecteur, « slicing »
  - La lecture d'éléments procède par coupes « slices »
  - Une coupe : **début (inclus): fin(exclue) : pas**
  - v = np.array(range(0,103,10)); v :

    array([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
  - Dernier élément : v[-1] ou v[v.size-1]
  - Avant-dernier : v[-2]
- Lecture de valeurs dans une matrice
  - m = np.array([[10*i+j for j in range(3)] for i in range(4)]) #m[i, j] = 10 i+ j
  - array([[0, 1, 2],[10, 11, 12], [20, 21, 22], [30, 31, 32]])
- Vecteur des entiers de 0 à n: v = np.arange(n)
- Adresse mémoire de a : id(v)

```
]: v = np.arange(10);v,id(v)
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), 4758641520)
```


Vues

- Les sous-tableaux extraits avec des opérations de slicing peuvent être affectés à de nouvelles variables
- Dans ce cas, ce ne sont pas des variables indépendantes, mais ce sont des vues des tableaux
- Si l'on modifie le tableau initial, la vue et modifiée, et réciproquement
- La base du tableau b c'est le tableau a : **b.** `base` affiche le tab. a

```
Entrée [14]: import numpy as np
             a=np.ones((4,4))
             b=a[::2,::2]
             b[0,0]=0
             a[2,2]=3
             a
Out[14]: array([[0., 1., 1., 1.],
                [1., 1., 1., 1.],
                [1., 1., 3., 1.],
                [1., 1., 1., 1.]])

Entrée [15]: b
Out[15]: array([[0., 1.],
                [1., 3.]])
```

```
a = [1,2,3,4,5,6,7,8,9]
a[::3]  donne [1, 4, 7]
a[2::3] donne [3, 6, 9]
```


Fancy indexing en lecture

- Fancy indexing : accès dans un ordre quelconque
  - Possibilité d'indexer des arrays au moyen de listes

```
: a = np.arange(0,100,10);a
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])

: b = a[[5,2,1,3]]; b
array([50, 20, 10, 30])

: a = np.arange(24).reshape(4,6); a
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]])

: b = a[[3,1,0,1]]; b
array([[18, 19, 20, 21, 22, 23],
       [ 6,  7,  8,  9, 10, 11],
       [ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

```
]: indices = np.array([[5,2,1],[3,8,7]]
   b = a[indices]; b
: array([[50, 20, 10],
         [30, 80, 70]])
```

4ème ligne, 2ème ligne, 1ère ligne et encore 2ème ligne

Fancy indexing en écriture

```
: a = np.arange(0,80,10); a
array([ 0, 10, 20, 30, 40, 50, 60, 70])

: a[[3, 2, 5, 4]] = (3333, 2222, 5555, 4444); a |  # on modifie a[3], a[2], a[5] et a[4]
array([   0,   10, 2222, 3333, 4444, 5555,   60,   70])
```

- a.put(positions,source) : remplace, terme à terme, les valeurs de source dans les positions indiquées du tableau a
- a.take(positions) : lit, terme à terme, les valeurs de a dans les positions indiquées.

```
: a = np.arange(10); a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

: a.put([5,1,6,2],[55,11,66,22]); a
array([ 0, 11, 22,  3,  4, 55, 66,  7,  8,  9])
```

```
: np.take(a,[6, 1, 5, 0])
array([66, 11, 55,  0])
```

Fancy indexing (suite)

Pour une matrice, on utilisera un argument supplémentaire « axis=0/1 » pour spécifier dans quel sens on fait la lecture(ligne/colonne). En l'absence de cette précision, la lecture s'effectue comme si le tableau avait été « aplati ».

```
: a = np.arange(0,160,10).reshape(4,4); a
array([[  0,  10,  20,  30],
       [ 40,  50,  60,  70],
       [ 80,  90, 100, 110],
       [120, 130, 140, 150]])

: i = np.array([4,0,3,0])
  a.take(i)
array([40,  0, 30,  0])
```

```
: i=np.array([3,0,2])
  a.take(i,axis=0)
array([[120, 130, 140, 150],
       [  0,  10,  20,  30],
       [ 80,  90, 100, 110]])

: a.take(i,axis=1)
array([[ 30,   0,  20],
       [ 70,  40,  60],
       [110,  80, 100],
       [150, 120, 140]])
```


Fancy indexing (suite)

- place (a,conds,b) écrit dans a les valeurs de b aux positions de a spécifiées par le tableau de booléens conds
- copyto est presque synonyme de la fonction place
- Exemple : Remplacer toutes les valeurs multiples de 3 par des 0

```
: a= np.arange(10); a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

: np.place(a, np.mod(a,3) == 0, 0); a
array([0, 1, 2, 0, 4, 5, 0, 7, 8, 0])
```

```
: a= np.arange(10); a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

: np.copyto(a,0, where=np.mod(a,3) == 0);a
array([0, 1, 2, 0, 4, 5, 0, 7, 8, 0])
```


Dimensions d'un tableau

- Redimensionnement par reshape ou shape /resize
  - Contrainte : nombre total d'éléments doit rester constant

```
: a = np.array(range(6)); a
array([0, 1, 2, 3, 4, 5])

: a.reshape(2,3)
array([[0, 1, 2],
       [3, 4, 5]])
```

Avec rechape, le tab. a ne change pas

```
: a.reshape(3,2)
array([[0, 1],
       [2, 3],
       [4, 5]])

: a.reshape(6,1)
array([[0],
       [1],
       [2],
       [3],
       [4],
       [5]])
```

- Avec shape /resize le redimensionnement est effectif

```
: a = np.array(range(6)); a
array([0, 1, 2, 3, 4, 5])

: a.shape = (3,2);a
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
: a = np.array(range(6)); a
array([0, 1, 2, 3, 4, 5])

: a.resize(3,2); a
array([[0, 1],
       [2, 3],
       [4, 5]])
```

le tab. a change

Redimensionnement

- La fonction resize permet également de créer des tableaux **constants** contenant une autre valeur que 0 et 1 (sinon on utiliserait les fonctions zeros ou ones) :

```
np.resize(2.3, 4)
array([2.3, 2.3, 2.3, 2.3])
```

```
: np.resize(4, (3,6))
array([[4, 4, 4, 4, 4, 4],
       [4, 4, 4, 4, 4, 4],
       [4, 4, 4, 4, 4, 4]])
```

Aplatissement

- Aplatissement d'un tableau tab : flatten ou ravel
  - renvoie une copie « aplatie » de tab
  - tab.flatten() : parcours par défaut('C') (à droite d'abord)
  - tab.flatten('F'): parcours Fortran (en bas d'abord)

```
: a = np.array([[1,5,3],[2,7,4]])
: a.flatten()
array([1, 5, 3, 2, 7, 4])
: a.flatten('F')
array([1, 2, 5, 7, 3, 4])
```

```
: a = np.array([[1,5,3],[2,7,4]])
: a.ravel()
array([1, 5, 3, 2, 7, 4])
: a.ravel('F')
array([1, 2, 5, 7, 3, 4])
```


Accès séquentiel au tableau

- tab. flat permet d'accéder aux éléments du tableau

```
: a = np.array([[1,5,3],[2,7,4]]);a
array([[1, 5, 3],
       [2, 7, 4]])
: a.flat[3]
2
```

```
: b=a.flatten('C');b
array([1, 5, 3, 2, 7, 4])
: b.flat[3]
2
```

```
: b=a.flatten('F');b
array([1, 2, 5, 7, 3, 4])
: b.flat[3]
7
```


Lecture/Ecriture dans un tableau

- Lecture de valeurs dans un vecteur, « slicing »
  - La lecture d'éléments procède par coupes « slices »
  - Une coupe : **début (inclus): fin(exclue) : pas**
  - v = np.array(range(0,103,10)); v :
    array([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
  - Dernier élément : v[-1] ou v[v.size-1]
  - Avant-dernier : v[-2]
- Lecture de valeurs dans une matrice
  - m = np.array([[10*i+j for j in range(3)] for i in range(4)]) #m[i, j] = 10 i+ j
  - array([[0, 1, 2],[10, 11, 12], [20, 21, 22], [30, 31, 32]])
- Vecteur des entiers de 0 à n: v = np.arange(n)
- Adresse mémoire de a : id(v)

```
]: v = np.arange(10);v,id(v)
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), 4758641520)
```


Vues

- Les sous-tableaux extraits avec des opérations de slicing peuvent être affectés à de nouvelles variables
- Dans ce cas, ce ne sont pas des variables indépendantes, mais ce sont des vues des tableaux
- Si l'on modifie le tableau initial, la vue et modifiée, et réciproquement
- La base du tableau b c'est le tableau a : **b.** `base` affiche le tab. a

```
Entrée [14]: import numpy as np
             a=np.ones((4,4))
             b=a[::2,::2]
             b[0,0]=0
             a[2,2]=3
             a
Out[14]: array([[0., 1., 1., 1.],
                [1., 1., 1., 1.],
                [1., 1., 3., 1.],
                [1., 1., 1., 1.]])

Entrée [15]: b
Out[15]: array([[0., 1.],
                [1., 3.]])
```

```
a = [1,2,3,4,5,6,7,8,9]
a[::3]  donne [1, 4, 7]
a[2::3] donne [3, 6, 9]
```

Fancy indexing en lecture

- Fancy indexing : accès dans un ordre quelconque
  - Possibilité d'indexer des arrays au moyen de listes

```
In : a = np.arange(0,100,10);a
Out: array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])

In : b = a[[5,2,1,3]]; b
Out: array([50, 20, 10, 30])
```

```
In : a = np.arange(24).reshape(4,6); a
Out: 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]])

In : b = a[[3,1,0,1]]; b
Out: array([[18, 19, 20, 21, 22, 23],
            [ 6,  7,  8,  9, 10, 11],
            [ 0,  1,  2,  3,  4,  5],
            [ 6,  7,  8,  9, 10, 11]])
```

```
In : indices = np.array([[5,2,1],[3,8,7]]
     b = a[indices]; b
Out: array([[50, 20, 10],
            [30, 80, 70]])
```

4ème ligne, 2ème ligne, 1ère ligne et encore 2ème ligne

Fancy indexing en écriture

```
In : a = np.arange(0,80,10); a
Out: array([ 0, 10, 20, 30, 40, 50, 60, 70])

In : a[[3, 2, 5, 4]] = (3333, 2222, 5555, 4444); a   # on modifie a[3], a[2], a[5] et a[4]
Out: array([   0,   10, 2222, 3333, 4444, 5555,   60,   70])
```

- a.put(positions,source) : remplace, terme à terme, les valeurs de source dans les positions indiquées du tableau a
- a.take(positions) : lit, terme à terme, les valeurs de a dans les positions indiquées.

```
In : a = np.arange(10); a
Out: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In : a.put([5,1,6,2],[55,11,66,22]); a
Out: array([ 0, 11, 22,  3,  4, 55, 66,  7,  8,  9])
```

```
In : np.take(a,[6, 1, 5, 0])
Out: array([66, 11, 55,  0])
```


Fancy indexing (suite)

Pour une matrice, on utilisera un argument supplémentaire « axis=0/1 » pour spécifier dans quel sens on fait la lecture(ligne/colonne). En l'absence de cette précision, la lecture s'effectue comme si le tableau avait été « aplati ».

```
In : a = np.arange(0,160,10).reshape(4,4); a
Out: array([[  0,  10,  20,  30],
            [ 40,  50,  60,  70],
            [ 80,  90, 100, 110],
            [120, 130, 140, 150]])

In : i = np.array([4,0,3,0])
     a.take(i)
Out: array([40,  0, 30,  0])
```

```
In : i=np.array([3,0,2])
     a.take(i,axis=0)
Out: array([[120, 130, 140, 150],
            [  0,  10,  20,  30],
            [ 80,  90, 100, 110]])

In : a.take(i,axis=1)
Out: array([[ 30,   0,  20],
            [ 70,  40,  60],
            [110,  80, 100],
            [150, 120, 140]])
```


Fancy indexing (suite)

- place (a,conds,b) écrit dans a les valeurs de b aux positions de a spécifiées par le tableau de booléens conds
- copyto est presque synonyme de la fonction place
- Exemple : Remplacer toutes les valeurs multiples de 3 par des 0

```
In : a= np.arange(10); a
Out: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In : np.place(a, np.mod(a,3) == 0, 0); a
Out: array([0, 1, 2, 0, 4, 5, 0, 7, 8, 0])
```

```
In : a= np.arange(10); a
Out: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In : np.copyto(a,0, where=np.mod(a,3) == 0);a
Out: array([0, 1, 2, 0, 4, 5, 0, 7, 8, 0])
```

Dimensions d'un tableau

- Redimensionnement par reshape ou shape /resize
  - Contrainte : nombre total d'éléments doit rester constant

```
: a = np.array(range(6)); a
array([0, 1, 2, 3, 4, 5])

: a.reshape(2,3)
array([[0, 1, 2],
       [3, 4, 5]])
```

Avec rechape, le tab. a ne change pas

```
: a.reshape(3,2)
array([[0, 1],
       [2, 3],
       [4, 5]])

: a.reshape(6,1)
array([[0],
       [1],
       [2],
       [3],
       [4],
       [5]])
```

- Avec shape /resize le redimensionnement est effectif

```
: a = np.array(range(6)); a
array([0, 1, 2, 3, 4, 5])

: a.shape = (3,2);a
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
: a = np.array(range(6)); a
array([0, 1, 2, 3, 4, 5])

: a.resize(3,2); a
array([[0, 1],
       [2, 3],
       [4, 5]])
```

le tab. a change

Redimensionnement

- La fonction resize permet également de créer des tableaux **constants** contenant une autre valeur que 0 et 1 (sinon on utiliserait les fonctions zeros ou ones) :

```
np.resize(2.3, 4)
array([2.3, 2.3, 2.3, 2.3])
```

```
: np.resize(4, (3,6))
array([[4, 4, 4, 4, 4, 4],
       [4, 4, 4, 4, 4, 4],
       [4, 4, 4, 4, 4, 4]])
```


Aplatissement

- Aplatissement d'un tableau tab : flatten ou ravel
  - renvoie une copie « aplatie » de tab
  - tab.flatten() : parcours par défaut('C') (à droite d'abord)
  - tab.flatten('F'): parcours Fortran (en bas d'abord)

```
: a = np.array([[1,5,3],[2,7,4]])

: a.flatten()
array([1, 5, 3, 2, 7, 4])

: a.flatten('F')
array([1, 2, 5, 7, 3, 4])
```

```
: a = np.array([[1,5,3],[2,7,4]])

: a.ravel()
array([1, 5, 3, 2, 7, 4])

: a.ravel('F')
array([1, 2, 5, 7, 3, 4])
```


Accès séquentiel au tableau

- tab. flat permet d'accéder aux éléments du tableau

```
: a = np.array([[1,5,3],[2,7,4]]);a
array([[1, 5, 3],
       [2, 7, 4]])

: a.flat[3]
2

: b=a.flatten('C');b
array([1, 5, 3, 2, 7, 4])

: b.flat[3]
2
```

```
: b=a.flatten('F');b
array([1, 2, 5, 7, 3, 4])

: b.flat[3]
7
```

Transposition d'une matrice

- la fonction transpose ou T renvoie la transposée d'une matrice

```
: a = np.array(range(6)); a
array([0, 1, 2, 3, 4, 5])

: a.transpose()
array([[1, 2],
       [5, 7],
       [3, 4]])

: a.T
array([[1, 2],
       [5, 7],
       [3, 4]])
```


Suppression/Insertion de lignes et colonnes

- la fonction delete(tab,k,axis) supprime la k^{ième} ligne/colonne (axis=0/1)

```
: a = np.array([[1,5,3],[2,7,4],[9,8,0]]); a
array([[1, 5, 3],
       [2, 7, 4],
       [9, 8, 0]])

: b = np.delete(a,1,0);b
array([[1, 5, 3],
       [9, 8, 0]])
```

- la fonction insert(tab,k,v,axis) insert la valeur v avant k^{ième} ligne/colonne (axis=0/1)

```
: np.insert(a,2,-1,axis=1)
array([[ 1,  5, -1,  3],
       [ 2,  7, -1,  4],
       [ 9,  8, -1,  0]])
```


Suppression/Insertion (suite)

- la fonction append(tab,M,axis) ajoute une matrice M après la dernière ligne/colonne (axis=0/1)

```
: a = np.array([[1,5,3],[2,7,4],[9,8,0]]); a
array([[1, 5, 3],
       [2, 7, 4],
       [9, 8, 0]])

: np.append(a,[[77],[88],[99]],1)
array([[ 1,  5,  3, 77],
       [ 2,  7,  4, 88],
       [ 9,  8,  0, 99]])

: np.append(a,[[33,55,77]],0)
array([[ 1,  5,  3],
       [ 2,  7,  4],
       [ 9,  8,  0],
       [33, 55, 77]])
```


Permutation / rotations de lignes et de colonnes

- la fonction fliplr(tab) inverse l'ordre des colonnes de ta en lr : left right

```
: a = np.array([[1,5,3],[2,7,4],[9,8,0]]); a
array([[1, 5, 3],
       [2, 7, 4],
       [9, 8, 0]])

: np.fliplr(a)
array([[3, 5, 1],
       [4, 7, 2],
       [0, 8, 9]])
```

- la fonction flipud(tab) inverse l'ordre des lignes de tab en ud : up down

```
: a = np.arange(1,10); a
array([1, 2, 3, 4, 5, 6, 7, 8, 9])

: np.flipud(a)
array([9, 8, 7, 6, 5, 4, 3, 2, 1])

: np.flipud(a)
array([[9, 8, 0],
       [2, 7, 4],
       [1, 5, 3]])

: a[:7] =np.flipud(a[:7]); a
array([7, 6, 5, 4, 3, 2, 1, 8, 9])
```

Permutation / rotations (suite)

- la fonction rot90(tab,k) renvoie une copie de tab après k rotations d'angle $\pi/2$.

```
: a = np.array([[1,5,3],[2,7,4],[9,8,0]]); a
array([[1, 5, 3],
       [2, 7, 4],
       [9, 8, 0]])
```

```
: np.rot90(a,1)
array([[3, 4, 0],
       [5, 7, 8],
       [1, 2, 9]])
```

```
: np.rot90(a,-1)
array([[9, 2, 1],
       [8, 7, 5],
       [0, 4, 3]])
```


Permutation / rotations (suite)

- la fonction roll(tab,k,axis) renvoie une copie de tab après k rotations d'une position (vers la droite si k>0, vers la gauche si k<0). Les éléments qui sortent d'un coté rentrent de l'autre.
- Pour une matrice :
  - axis=0 (par défaut) : c'est une rotation sur les lignes ()).
  - axis=1: c'est une rotation sur les colonnes d'une matrice.

```
: a = np.arange(10); a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.roll(a,1)
array([9, 0, 1, 2, 3, 4, 5, 6, 7, 8])
```

```
np.roll(a,2)
array([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])
```

```
np.roll(a,-2)
array([2, 3, 4, 5, 6, 7, 8, 9, 0, 1])
```

```
m= np.arange(15).reshape(3,5);m
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
np.roll(m,1,axis=1)
array([[ 4,  0,  1,  2,  3],
       [ 9,  5,  6,  7,  8],
       [14, 10, 11, 12, 13]])
```

```
: np.roll(m,1,axis=0)
array([[10, 11, 12, 13, 14],
       [ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9]])
```

```
: np.roll(m,-2,axis=1)
array([[ 2,  3,  4,  0,  1],
       [ 7,  8,  9,  5,  6],
       [12, 13, 14, 10, 11]])
```


Opérations par blocs

- La fonction concatenate permet d'accoler deux ou plusieurs tableaux (horizontalement avec axis=1, verticalement avec axis=0).

```
a = np.array(range(8)).reshape(2,4);a
rray([[0, 1, 2, 3],
      [4, 5, 6, 7]])
```

```
b = np.zeros(a.shape,int); b
rray([[0, 0, 0, 0],
      [0, 0, 0, 0]])
```

```
c = np.ones(a.shape,int); c
rray([[1, 1, 1, 1],
      [1, 1, 1, 1]])
```

```
np.concatenate((a,b),axis=1)
rray([[0, 1, 2, 3, 0, 0, 0, 0],
      [4, 5, 6, 7, 0, 0, 0, 0]])
```

```
np.concatenate((a,c),axis=0)
rray([[0, 1, 2, 3],
      [4, 5, 6, 7],
      [1, 1, 1, 1],
      [1, 1, 1, 1]])
```

```
np.concatenate((a,b,c),axis=1)
rray([[0, 1, 2, 3, 0, 0, 0, 0, 1, 1, 1, 1],
      [4, 5, 6, 7, 0, 0, 0, 0, 1, 1, 1, 1]]
```


Opérations par blocs (suite)

- La fonction hstack/vstack permet une concaténation horizontale / verticale

```
a = np.array(range(8)).reshape(2,4);a
rray([[0, 1, 2, 3],
      [4, 5, 6, 7]])
```

```
b = np.zeros(a.shape,int); b
rray([[0, 0, 0, 0],
      [0, 0, 0, 0]])
```

```
: np.hstack((a,b))
array([[0, 1, 2, 3, 0, 0, 0, 0],
       [4, 5, 6, 7, 0, 0, 0, 0]])
```

```
: np.vstack((a,b))
array([[0, 1, 2, 3],
       [4, 5, 6, 7],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

Opérations par blocs (suite)

- La fonction column_stack combine deux tableaux-lignes 1D en colonnes d'un tableau 2D

```
a = np.array((1,2,3))
b = np.array((3,4,5))
```

```
np.column_stack((a,b))
```

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


Opérations par blocs (suite)

- La fonction tile (tab,[n,p]) construit un tableau en répétant **n** fois le tableau tab dans le sens horizontal et **p** fois dans le sens vertical.
  - [1,1] : tableau inchangé

```
a=np.array([[1,2,3],[4,5,6]]);a
```

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

```
np.tile(a,[1,2])
```

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

```
np.tile(a,2)
```

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

```
np.tile(a,[2,1])
```

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


Opérations par blocs (suite)

- La fonction vsplit(tab,k) (resp. hsplit(tab,k)) renvoie un tuple de k tableaux de n/ k lignes (resp. colonnes) représentant un découpage du tableau tab
  - Nombre n de lignes (resp. colonnes) de tab doit être un multiple de k.

```
a = np.array(range(8)).reshape(2,4);a
```

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

```
b, c = np.vsplit(a,2)
b
```

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

```
c
```

```
array([[4, 5, 6, 7]])
```

```
b, c = np.hsplit(a,2)
b
```

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

```
c
```

```
array([[2, 3],
       [6, 7]])
```


Tableaux de valeurs échelonnées

- La fonction arange (deb,fin,h,dtype=…) crée des valeurs régulièrement espacées de l'intervalle [deb,fin[avec un pas de h

```
np.arange(100,200,30)
```

```
array([100, 130, 160, 190])
```

```
np.arange(10.,0,-1)
```

```
array([10., 9., 8., 7., 6., 5., 4., 3., 2., 1.])
```

- linspace(a,b,n) : le résultat est un vecteur de n valeurs régulièrement échelonnées du segment [a, b](b inclus).
  - Si b < a , les valeurs sont obtenues dans l'ordre décroissant.

```
A = np.linspace(0, 1, 11);A # 7-11 valeurs de [0,1] dans l'ordre croissant
```

```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

```
A = np.linspace(1,0, 11);A # 7-11 valeurs de [0,1] dans l'ordre décroissant
```

```
array([1. , 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0. ])
```

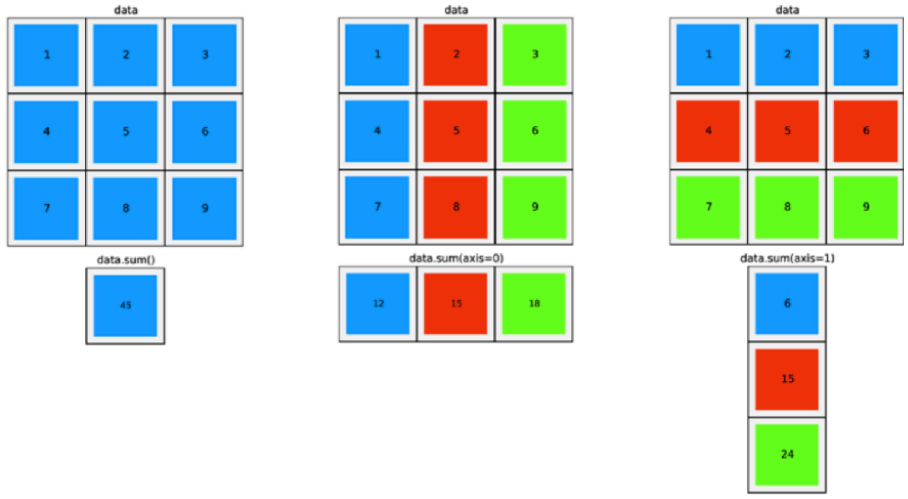
Tableaux répondant à une formule donnée

- La fonction fromfunction permet de construire un tableau dont le terme général obéit à une formule donnée.

```
: def f(i,j):
      return 10*i+j

: np.fromfunction(f,(3,4))

array([[ 0.,  1.,  2.,  3.],
       [10., 11., 12., 13.],
       [20., 21., 22., 23.]])
```

```
|: np.fromfunction(lambda i,j: 10*i+j,(3,4))

array([[ 0.,  1.,  2.,  3.],
       [10., 11., 12., 13.],
       [20., 21., 22., 23.]])
```


Choix des axes pour l'agrégation



Fonctions d'agrégation

- Ces fonctions condensent un array (ou une partie d'array) par un scalaire. Par défaut, agrégation sur le tableau entier

NumPy Function	Description
np.mean	The average of all values in the array.
np.std	Standard deviation.
np.var	Variance.
np.sum	Sum of all elements.
np.prod	Product of all elements.
np.cumsum	Cumulative sum of all elements.
np.cumprod	Cumulative product of all elements.
np.min, np.max	The minimum / maximum value in an array.
np.argmin, np.argmax	The index of the minimum / maximum value in an array.
np.all	Return True if all elements in the argument array are nonzero.
np.any	Return True if any of the elements in the argument array is nonzero.

Opérations terme à terme

- add (a,b) additionne les éléments de a et b (autre syntaxe possible : a + b)
- subtract (a,b) soustrait les éléments de b à ceux de a (autre syntaxe possible : a − b)
- multiply (a,b) multiplie les éléments de a et b (autre syntaxe possible : a ∗ b)
- divide (a,b) quotients (flottants) des éléments de a par b (autre syntaxe : a/b)
- floor_divide(a,b) quotients (entiers) des divisions de a par b (autre : a//b)
- power(a,b) élève les éléments de a à la puissance les éléments de b
- mod(a,b) donne les restes dans les divisions des éléments de a par ceux de b
- negative(a) (tableaux des opposés : autre syntaxe possible −a)
- Autres :
  - absolute(a) (modules), sign(a) (signes)
  - Arrondis : rint(a) (à l'entier), floor(a) , ceil(a) , trunc(a) (troncature), round_(a,n) (arrondi à n décimales)
  - Racine carrée et élévation au carré (toujours terme à terme) : sqrt et square .

Opérations terme à terme (suite)

NumPy function	Description
`np.add, np.subtract, np.multiply, np.divide`	Addition, subtraction, multiplication and division of two NumPy arrays.
`np.power`	Raise first input argument to the power of the second input argument (applied elementwise).
`np.remainder`	The remainder of division.
`np.reciprocal`	The reciprocal (inverse) of each element.
`np.real, np.imag, np.conj`	The real part, imaginary part, and the complex conjugate of the elements in the input arrays.
`np.sign, np.abs`	The sign and the absolute value.
`np.floor, np.ceil, np.rint`	Convert to integer values.
`np.round`	Round to a given number of decimals.

Fonctions mathématiques

- Fonctions trigonométriques : :sin cos tan arcsin arccos arctan sinh cosh tanh arcsinh arccosh arctanh
- hypot(a,b) renvoie les hypoténuses sqrt(x2 + y2)
- arctan2(a,b) renvoie les angles polaires des points (y, x)
- degrees(a) (ou encore rad2deg) convertit les angles x de radians en degrés
- radians(a) (ou encore deg2rad) convertit les angles x de degrés en radians
- Fonctions exponentielles et logarithmiques :
  - exp expm1 : expm1(x) signifie ex – 1 et est plus précis que exp(x)-1 pour x proche de 0.
  - log log10 log2 : logarithme népérien (resp. de base 10, de base 2)
  - log1p : log1p(x) signifie ln(1 + x) et est plus précis que log(1+x) pour x proche de 0).
  - exp2 : exp2(x) signifie 2**x, c'est-à-dire 2^x.

Expressions booléennes

- Les opérateurs <,>,==, etc. permettent de faire une comparaison terme à terme, et supportent le broadcasting
- Les résultats des opérateurs booléens peuvent être combinés avec les fonctions np.any ou np.all
- Cela permet d'éviter le recours à des branchements conditionnels et de tout écrire sous la forme d'expressions mathématiques :

```
]: x = np.array([-2, -1, 0, 1, 2])

]: x > 0
array([False, False, False,  True,  True])

]: 1 * (x > 0)
array([0, 0, 0, 1, 1])

]: x * (x > 0)
array([0, 0, 0, 1, 2])
```


Opérations sur tableaux

Function	Description
`np.where`	Choose values from two arrays depending on the value of a condition array.
`np.choose`	Choose values from a list of arrays depending on the values of a given index array.
`np.select`	Choose values from a list of arrays depending on a list of conditions.
`np.nonzero`	Return an array with indices of nonzero elements.
`np.logical_and`	Perform and elementwise AND operation.
`np.logical_or, np.logical_xor`	Elementwise OR/XOR operations.
`np.logical_not`	Elementwise NOT operation (inverting).

Function	Description
`np.unique`	Create a new array with unique elements, where each value only appears once.
`np.in1d`	Test for the existence of an array of elements in another array.
`np.intersect1d`	Return an array with elements that are contained in two given arrays.
`np.setdiff1d`	Return an array with elements that are contained in one but not the other, of two given arrays.
`np.union1d`	Return an array with elements that are contained in either, or both, of two given arrays.

Opérations sur tableaux (suite)

Function	Description
`np.transpose`, `np.ndarray.transpose`, `np.ndarray.T`	The transpose (reverse axes) of an array.
`np.fliplr / np.flipud`	Reverse the elements in each row / column.
`np.rot90`	Rotate the elements along the first two axes by 90 degrees.
`np.sort`, `np.ndarray.sort`	Sort the element of an array along a given specified axis (which default to the last axis of the array). The np.ndarray method sort performs the sorting in place, modifying the input array.

NumPy Function	Description
`np.dot`	Matrix multiplication (dot product) between two given arrays representing vectors, arrays, or tensors.
`np.inner`	Scalar multiplication (inner product) between two arrays representing vectors.
`np.cross`	The cross product between two arrays that represent vectors.
`np.tensordot`	Dot product along specified axes of multidimensional arrays.
`np.outer`	Outer product (tensor product of vectors) between two arrays representing vectors.
`np.kron`	Kronecker product (tensor product of matrices) between arrays representing matrices and higher-dimensional arrays.
`np.einsum`	Evaluates Einstein's summation convention for multidimensional arrays.

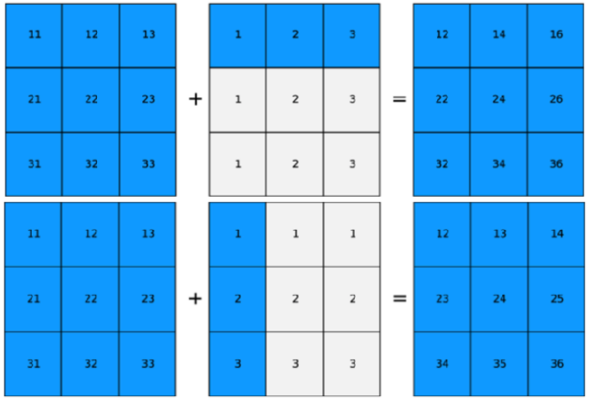

```
: a = np.arange(1,7);b = np.array([4,1,8,7,2,9])
: def g(x): return (x**2)
: vf = np.vectorize(g);vf(a)
array([ 1,  4,  9, 16, 25, 36])
: def f(x,y): return 0 if x < y else y
: vf = np.vectorize(f);vf(a,b)
array([0, 1, 0, 0, 2, 0])
: np.piecewise(a, [a < 3 , a >5 , a >= 5], [-1,0,1])
array([-1, -1,  0,  0,  1,  1])
: m = np.arange(15).reshape(3,5); m
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
: np.apply_along_axis(np.sum,0,m) # somme lignes
array([15, 18, 21, 24, 27])
: np.apply_along_axis(np.sum,1,m) # somme colonnes
array([10, 35, 60])
: np.apply_along_axis(np.mean,0,m) # moyenne lignes
array([5., 6., 7., 8., 9.])
: np.apply_along_axis(np.mean,1,m) # moyenne colonnes
array([ 2.,  7., 12.])
```


Vectorisation d'une fonction

- vectorize permet de vectoriser une fonction : application, terme à terme, à tous les éléments du tableau.
- piecewise (a,[conds],[images] permet de calculer l'image d'un tableau par une fonction f par morceaux.
  - Chaque f(x) est calculé en choisissant dans images ce qui correspond au premier test vérifié dans conds.
- apply_along_axis permet d'appliquer une même fonction suivant les lignes, ou suivant les colonnes.

Remarque : Broadcasting

- Le broadcasting consiste à rendre possibles des opérations sur les tableaux qui ne sont pas possible sinon.
  - Par exemple, ajout d'un tableau 3x3 avec un tableau 1x3
- Règle de base : si dim=1 les lignes/colonnes sont recopiées pour atteindre la taille de l'autre tableau



Tableaux pseudo-aléatoires

- random.seed réinitialise le générateur de nombres aléatoires (argument entier).
- random.rand renvoie un tableau de valeurs pseudo-aléatoires dans l'intervalle [0, 1]. random.randn : même syntaxe mais elle renvoie un échantillon de valeurs pseudo aléatoires au sens de la loi normale réduite (c'est-à-dire d'espérance 0 et d'écart-type 1).
- random.sample analogue à random.rand mais l'argument est un tuple.
- random.randint renvoie une valeur ou un tab. d'entiers pseudo-aléatoires au sens de la distribution uniforme dans un intervalle semi-ouvert [a, b[.
- random.choice renvoie un échantillon aléatoire de valeurs extraites d'un tableau a .
- random.shuffle rebat aléatoirement les éléments d'un vecteur.
- random.permutation : analogue à shuffle. Elle accepte un entier n(et aussi un intervalle comme argument. le résultat est alors une permutation de l'intervalle d'entiers [0, n[.

Tableaux pseudo-aléatoires (suite)

```
In : np.random.seed(0)
     np.random.rand(4)
Out: array([0.5488135 , 0.71518937, 0.60276338, 0.54488318])

In : np.random.rand(2,4)
Out: array([[0.4236548 , 0.64589411, 0.43758721, 0.891773  ],
            [0.96366276, 0.38344152, 0.79172504, 0.52889492]])

In : np.random.randint(100) # un seul entier pseudo-aléatoire dans [0,99[
Out: 88

In : np.random.randint(1,10,size=6)
Out: array([4, 6, 1, 3, 4, 9])

In : a = np.arange(10);a
Out: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In : np.random.choice(a)# une valeur choisie dans a
Out: 3

In : np.random.choice(a,(2,6)) # un tableau 2 × 6 de valeurs choisies dans a
Out: array([[3, 3, 7, 0, 1, 9],
            [9, 0, 4, 7, 3, 2]])

In : np.random.shuffle(a); a
Out: array([3, 8, 1, 5, 6, 4, 9, 0, 2, 7])

In : np.random.permutation(6) # une permutation de [0,6[
Out: array([0, 5, 3, 2, 1, 4])

In : np.random.permutation(a) # une permutation des valeurs de a
Out: array([4, 3, 6, 1, 5, 9, 0, 7, 8, 2])
```


Probabilités

- random.binomial (n,p) - **loi binomiale** : Nb de succès à l'issue de n tentatives avec une probabilité p de succès à chaque fois.
- random.negative_binomial(n,p) - **loi binomiale négative**. Cette loi mesure le nombre d'échecs avant d'atteindre le n-ième succès dans une répétition de tentatives indépendantes ayant chacune une probabilité p de succès.
- random.geometric (p) : **loi géométrique** de paramètre p, c'est-à-dire le temps d'attente du premier succès (ou encore le nombre d'échecs avant celui-ci) dans une répétition de tentatives indépendantes ayant chacune une probabilité p de succès.
- random.hypergeometric(n_good,n_bad,N) **loi hypergéométrique**. Si on considère le modèle classique d'une boîte contenant n_good jetons gagnants et n_bad jetons perdants, cette loi mesure le nombre de jetons gagnants obtenus après N tirages sans remise (on suppose donc que N ≤ n_good + n_bad).

Probabilités(bis)

- multinomial(n,probs) : **loi multinomiale**. Ici p représente un vecteur $[p_1, \ldots, p_k]$ de probabilités (p_i sont dans [0, 1] et leur somme vaut 1). Il s'agit ici de répéter n fois (indépendamment) une même expérience possédant k résultats possibles $r_1, \ldots, r_k$ (avec p_i la probabilité du résultat r_i) et de lire le vecteur $[n_1, \ldots, n_k]$ des occurences de chacun de ces résultats
- random.poisson(λ) : **loi géométrique**.

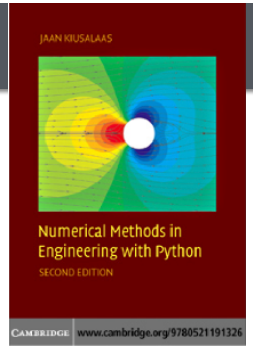
  Cette loi, d'image $X(\Omega) = \mathbb{N}$ est définie par $p_\lambda(X = k) = e^{-\lambda}\frac{\lambda^k}{k!}$ pour tout k de $\mathbb{N}$.

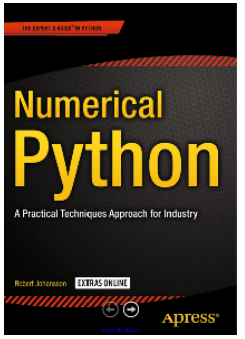
  D'espérance λ, elle est une bonne approximation de la loi binomiale $\mathcal{B}\left(n, p = \frac{\lambda}{n}\right)$ avec « p petit et n grand ».

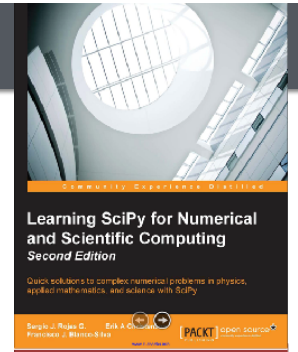
  Si un événement rare est susceptible de se présenter avec une probabilité $p \ll 1$ (ou encore λ fois) dans un grand intervalle de temps, alors $p_\lambda(X, k)$ mesure la probabilité qu'il survienne k fois dans un tel intervalle de temps.
- random.uniform ([low, high, size]) renvoie une / des réalisation(s) de la loi uniforme sur [low, high] (défaut [0, 1]).
- random.exponential ([beta, size]) renvoie une / des réalisation(s) de la loi exponentielle de paramètre $\lambda = 1/\beta$

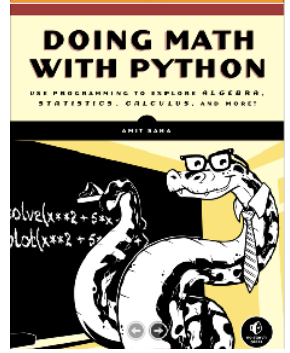
  C'est une loi continue, définie sur $\mathbb{R}^+$, dont la densité s'écrit $f(x) = \lambda e^{-\lambda x} = \frac{1}{\beta}\exp\left(-\frac{x}{\beta}\right)$. Son espérance est β.
- random.normal ([m, σ , size]) renvoie une / des réalisation(s) de la loi normale d'espérance m, d'écart-type σ.

```
np.random.binomial(100,1/2)# nombre de succès après 100 essais avec p=1/2
7

np.random.binomial(200,1/3,size=10) # répéter 10 fois une série de 200 essais avec p=1/3
rray([67, 70, 61, 67, 65, 59, 57, 66, 64, 65])

e=np.random.geometric(1/10,size=5);e #5 fois de suite, on a attendu le premier succès, avec p = 1/10 à chaque tenta
rray([3, 9, 9, 9, 3])

e.mean(), e.var()#moyenne et variance des observations
6.6, 8.64)

np.random.negative_binomial(100,1/2) # avec p=1/2, on a connu 105 échecs avec le 100ème succès
05

e = np.random.hypergeometric(30,70,50); e # 30 jetons ok, 70 "pas ok", 50 tirages sans remise
                                          # ici on a obtenu 15 jetons "ok"
5

p.random.multinomial(1000,[1/2,1/3,1/6]) #Ici on effectue 1000 fois la même expérience aléatoire comportant
                                         # trois résultats possibles avec les probabilités respectivesl/2, 1/3, 1/6.
                                         # On obtient resp. 4064 fois, 355 fois, 181 fois les trois résultats possib
rray([464, 355, 181])

np.random.poisson(lam=2,size=25)#  n considère un évènement susceptible de se présenter deux fois dans un intervall
# de longueur n (avec n grand). On répète 25 fois l'expérience qui consiste à observer combien de fois cet évènemen
# effectivement produit dans un intervalle de temps [t0, t0 + n].
rray([0, 2, 2, 2, 3, 3, 2, 3, 2, 0, 4, 6, 2, 1, 1, 2, 2, 1, 2, 4, 1, 2,
      1, 1, 2])

np.random.uniform(0,10) # une réalisation de la loi uniforme sur [0,10]
.777518392924809

np.random.exponential(10,5) # 5 réalisations, loi exponentielle de paramètre 1/10
rray([ 1.41642031, 12.61812678,  5.04279932,  8.3337825 ,  2.0245876 ])

np.random.normal(6,1,5) # 5 fois la loi normale d'espérance 6 d'écart-type 1
rray([6.60415971, 5.96071718, 4.8319065 , 6.52327666, 5.82845367])

np.random.standard_normal(5) # 5 réalisations de la loi normale centrée réduite
rray([ 0.77179055,  0.82350415,  2.16323595,  1.33652795, -0.36918184])
```







Références