Librairie numpy

1 — Présentation de numpy

La librairie **numpy** est consacrée entièrement au calcul numérique en **Python**. Elle utilise essentiellement des variables de type **ndarray** (en abrégé **array**), qui l'on peut voir comme des tableaux à plusieurs dimensions. Les calculs avec **numpy** sont particulièrement optimisés car les **array** sont homogènes (ils ne contiennent que des valeurs d'un même type) et de taille fixé à la création.
Traditionnellement on charge la librairie **numpy** avec `import numpy as np`.

1.1 — Les bases

On définit un tableau avec la fonction **np.array** :

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

Cet **array A** a plusieurs attributs à connaître :

- **A.ndim** : sa dimension. C'est le nombre d'indices nécessaire pour accéder à une valeur numérique. Un tableau à une seule ligne comme `[1,2,3]` est de dimension 1, un tableau avec deux lignes comme le précédent, est de dimension 2, etc.
- **A.shape** : c'est une liste qui contient la longueur du tableau pour chaque dimension. Avec l'exemple donné, **A.shape** vaut `(3,3)` : il y a donc deux lignes, chacune avec trois colonnes.
- **A.dtype** : le type des données contenues dans **A**. Ici c'est **float64** (flottants stockés sur 64 bits).

1.2 — Fonctions élémentaires

Opérations	Commande	Commentaire
Création	`A = np.array(...)`	
Matrice nulle	`np.zeros((n,p))`	Deux paires de parenthèses !
Matrice identité	`np.identity(n)`	
Matrice diagonale	`np.diag(L)`	La liste **L** contient les valeurs sur la diagonale
Valeurs dans $[a\,;b[$ avec un pas **p**	`np.arange(a,b,p)`	On ne sait pas toujours combien de valeurs on va obtenir, à cause des erreurs d'arrondis
k valeurs régulièrement réparties dans $[a\,;b]$	`np.linspace(a,b,k)`	en général, préférable à **arange**
Matrice au hasard	`np.random.random((n,p))`	Deux paires de parenthèses et deux **random** !
Copie	`B = A.copy()`	Important pour pouvoir modifier **B** sans modifier **A**
Coefficients	`A[i,j]` ou `A[i][j]`	Les indices commencent à 0 !
Lignes	`A[i]`	
Colonnes	`A[:,j]`	

1.3 — Les opérations

On peut effectuer un grand nombre d'opérations directement sur les **array** : elles sont effectuées *élément par élément* !
Ainsi `A**2` va élever au carré chaque coefficient de **A** (j'insiste : ce n'est pas l'opération matricielle !).
La plupart des fonctions mathématiques sont redéfinies par **numpy** : par exemple `np.sin` pour prendre le sin de chaque élément de **A**, etc. On trouve de même exp, $\sqrt{}$, etc.
Quelques opérations se font globalement sur un tableau

Opérations	Commande	Commentaire
Somme de tous les éléments	`np.sum(A)`	
Produits de tous les éléments	`np.prod(A)`	
Tranposée	`np.tranpose(A)`	

1.4 — Les tests booléens

Les tests booléens nécessitent un petit effort de compréhension, amplement récompensés par leur efficacité. Tout d'abord il faut comprendre que pour **Python**, une opération booléenne... est une opération ! Avec **numpy**, cette opération est faite élément par élément. Par exemple

```
A = np.array([[1, -1 , 0], [-2, 4, -5]])
C = (A > 0)
```

va renvoyer le tableau de booléen **C**

```
array([[ True, False, False],
       [False,  True, False]], dtype=bool)
```

Il faut savoir utiliser ces tableaux de booléens selon ce que vous désirez en faire. Quelques exemples :

Comparer si deux tableaux sont égaux Le test `A == B` renvoie un tableau de booléens. Pour savoir si ce tableau ne contient que des `True`, on utilise la méthode `all()`. Donc ici le test se fait par `(A == B).all()`.
Avec la méthode `any()` on teste si au moins une des valeurs est vraie.

Sélectionner les valeurs d'un tableau (1) Pour extraire les valeurs positives du tableau précédent, on utilise tout simplement le tableau de booléens **C** : avec la commande `A[C]`, **Python** va parcourir le tableau **A** et à chaque fois que la valeur correspondante dans **C** est `True`, il va conserver cette valeur. Les résultats sont mis dans un tableau à une dimension.

Sélectionner les valeurs d'un tableau (2) Cette fois on cherche à transformer le tableau **A** en mettant à zéro les valeurs négatives. Très simple ! La commande `A * C` suffit. Pourquoi ? Un indice : dans les opérations algébriques, **Python** interprète `True` comme 1 et `False` comme 0. Je vous laisse finir l'explication...

2 — Exercices de base

Ex. 1 — Répondre à chacune des questions avec une seule ligne de code.

1. Créer un tableau **A** qui contient tous les multiples de 3 entre 0 et 100.
2. Créer un tableau **B** qui contient les sinus des carrés des éléments de **A** multipliés par $\pi/12$.
3. Déterminer le maximum du tableau **B** et le garder dans une variable.
4. Compter le nombre de fois que ce maximum est présent dans **B**. Commentaire.

Corrigé de l'exercice 1:

```
A = np.arange(0,100,3)
B = np.sin(A**2 * math.pi/12)
m = np.max(B)
np.sum(B == m)
```

Théoriquement, le maximum est atteint plusieurs fois. Les erreurs d'arrondis empêchent **Python** de s'en rendre compte.

Ex. 2 — Les fonctions demandées peuvent ne faire qu'une ligne de code ! On pourra utiliser les fonctions suivantes :

np.triu(A) Renvoie le tableau *A* dont les éléments *au-dessous* de la diagonale ont été annulés.

np.tril(A) Renvoie le tableau *A* dont les éléments *au-dessus* de la diagonale ont été annulés.

1. Écrire une fonction **est_triangulaire_sup(M)** qui renvoient **True** si la matrice **M** est triangulaire supérieure, **False** sinon.
2. Écrire une fonction **est_diagonale(M)** qui renvoient **True** si la matrice **M** est diagonale, **False** sinon.

Corrigé de l'exercice 2:

```
import numpy as np

def est_diagonale(M):
    """ Teste si la matrice M est diagonale"""
    n,p = M.shape
    if n != p:
        return False

    for i in range(n):
        for j in range(p):
            if i!=j and M[i,j]!= 0:
                return False

    return True

def est_diagonale2(M):
    """ Teste si la matrice M est diagonale"""
    n,p = M.shape
    if n != p:
        return False

    return (M == np.diag(np.diag(M))).all()

def est_triangulaire_sup(M):
    """ Teste si la matrice M est triangulaire supérieure""
    n,p = M.shape
    if n!= p:
        return False

    for i in range(n):
        for j in range(i):
            if M[i,j]!= 0:
                return False

    return True

def est_triangulaire_sup2(M):
    """ Teste si la matrice M est triangulaire supérieure""

    return (M == np.triu(M)).all()
```

Ex. 3 — LE JEU DE LA VIE Le jeu de la vie est un automate cellulaire inventé par John Conway en 1970. Malgré des règles très simples, il présente des comportements étonnamment sophistiqués.
Le principe est le suivant : sur un damier, les cases peuvent être blanches (« mortes ») ou noires (« vivantes »). À chaque génération, l'état d'une case est déterminée par celui de ses huit voisines de la façon suivante :
– une cellule morte qui possède exactement trois voisines vivantes devient vivante (« elle naît ») ;
– une cellule vivante possédant deux ou trois voisines vivantes reste vivante, sinon elle meurt.

Au bord du damier, il y a plusieurs conventions possibles. La plus simple à programmer consiste à coller les bords du damier entre eux. Ainsi les cellules au bord du damier ont comme voisines celles qui sont au bord du damier du côté opposé.
Télécharger le fichier **jeudelavie.py** et compléter-le avec les deux fonctions suivantes :

1. une fonction **CompteVoisines**(A) qui renvoie un **array** avec pour chaque case le nombre de voisines vivnates (vous pouvez faire des boucles...)
2. une fonction **NextGeneration**(A) (sans boucles !) qui, partant d'une situation, donne la situation à l'étape suivante.

Corrigé de l'exercice 3:

```
# Paul Pichaureau - 2015-09-07
# Le jeu de la vie

import numpy as np

#  Au hasard
hasard = np.floor(np.random.random((20,20))*2)

# Quelques configurations intéressantes
glisseur = np.array( [[0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,1,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,1,0,0,0,0,0,0,0,0,0],
               [0,0,1,1,1,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [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]])

canon = np.array( [ [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,
                    [1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,
                    [1,1,0,0,0,0,0,0,0,0,1,0,0,0,1,0,1,1,0,
                    [0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,


def comptevoisin(A):
    """Renvoie la matrice du nb de voisins vivants,
    pour chaque case"""
```

```
    return ( np.roll(A,-1,0)
             + np.roll(A, 1,0)
             + np.roll(A,-1,1)
             + np.roll(A, 1,1)
             + np.roll(np.roll(A,-1,0),-1,1)
             + np.roll(np.roll(A, 1,0), 1,1)
             + np.roll(np.roll(A, 1,0),-1,1)
             + np.roll(np.roll(A,-1,0), 1,1))

def nextgeneration(A):
    V = comptevoisin(A)
    return ( A * (V==2) +  (V==3))

import matplotlib.pyplot as plt
from matplotlib import animation

A = canon

nx, ny  = A.shape
fig = plt.figure()
im = plt.imshow(A, cmap='gist_gray_r', interpolation='none')

def init():
    im.set_data(np.zeros((nx, ny)))

def animate(i):
    global A
    A = nextgeneration(A)
    im.set_data(A)
    return im

anim = animation.FuncAnimation(fig, animate, init_func=init, frames=240, interval=200)
plt.show()
```