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 fixée à la création.

Traditionnellement on charge la librairie **numpy** avec la ligne
`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 `(2,3)` : 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 !
Ligne **i**	`A[i]`	
Colonne **j**	`A[:,j]`	
Sous-tableau	`A[1:5,2:4]`	Ici on fabrique une sous-matrice avec les coefficients des lignes 1 à 4 et des colonnes 2 et 3.

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

1.4 — Extraction d'une sous-matrice

Il est très simple d'extraire une partie d'une matrice. Considérons par exemple la matrice suivante

```
A = np.array([[1,0,1,0,1,0],
              [0,1,0,0,0,0],
              [1,0,1,0,1,0],
              [1,0,1,0,1,0],
              [1,0,0,1,0,0],
              [1,0,1,0,1,0]])
```

On peut facilement extraire le quart en haut à gauche de cette matrice avec `A[0:4,0:4]`. Cette commande extrait les lignes d'indice `0` (inclus) à `4` (exclu) et de même pour les lignes.

Quelles sont les commandes pour : extraire un carré de 3 lignes et 3 colonnes au centre de la matrice ? les 3 premières lignes ? les deux premières colonnes ?

1.5 — Les tests booléens

On reprend la matrice précédente, et on veut tester si le quart supérieur droit est égal au quart inférieur droit. Quelle commande vous vient spontanément à l'esprit ?

Il faut comprendre que pour `Python`, une opération booléenne... est une opération ! Avec `numpy`, cette opération est faite élément par élément. Ainsi

```
A[0:4, 0:4]  == A[4:6,4:6]
```

va renvoyer le tableau de booléen `C`

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

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

2 — Exercices

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`. Commentez.

Ex. 2 — 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.

Ex. 3 — 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.
3. Écrire une fonction **`est_inversible(M)`** qui renvoient **`True`** si la matrice **`M`** est inversible, **`False`** sinon.

Ex. 4 — Méthode de Strassen 1. Combien d'addition et de multiplication sont nécessaires pour effectuer le produit d'une matrice M de taille (n,p) par une matrice N de taille (p,q), si on suit la définition du cours ?

2. La méthode de Strassen est une méthode qui réduit le nombre d'opérations nécessaires. Supposons que n, p et q soient pairs. On coupe les matrices en quatre blocs de tailles $(n/2, p/2)$ et $(p/2, q/2)$

$$M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} \quad \text{et} \quad N = \begin{pmatrix} N_{11} & N_{12} \\ N_{21} & N_{22} \end{pmatrix}$$

et on calcule les 7 sous produits matriciels

$$\begin{cases} X_1 = (M_{11} + M_{22}) \times (N_{11} + N_{22}) \\ X_2 = (M_{21} + M_{22}) \times N_{11} \\ X_3 = M_{11} \times (N_{12} - N_{22}) \\ X_4 = M_{22} \times (N_{21} - N_{11}) \\ X_5 = (M_{11} + M_{12}) \times N_{22} \\ X_6 = (M_{21} - M_{11}) \times (N_{11} + N_{12}) \\ X_7 = (M_{12} - M_{22}) \times (N_{21} + N_{22}) \end{cases}$$

Puis on les recompose de la manière suivante

$$\begin{cases} P_{11} = X_1 + X_4 - X_5 + X_7 \\ P_{12} = X_3 + X_5 \\ P_{21} = X_2 + X_4 \\ P_{22} = X_1 - X_2 + X_3 + X_6 \end{cases}$$

On vérifie ainsi qu'on a bien

$$M \times N = \begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix}$$

a) Implémenter cet algorithme.
b) Dans le cas où m, n ou p est impair, on se ramène au cas précédent en ajoutant une ligne ou une colonne en bas ou à droite de M ou N. Modifier votre programme en conséquence.

On pourra utiliser la fonction **`concatenate`** de la librairie **`numpy`**.

c) Modifier enfin votre fonction en tirant partie de la récursivité.

Ex. 5 — Pivot de Gauss On résoudra cet exercice à l'aide la librairie **`numpy`**.

1. Programmer une fonction **`remontee(T,b)`** qui résout le système $TX = b$, où T est une matrice triangulaire supérieure. Si le système est de Cramer, **`remontee`** renvoie l'unique solution, sinon elle renvoie **`False`**.
2. Programmer une fonction pour chacune des 3 opérations élémentaires de la méthode du Pivot de Gauss.
3. Programmer l'algorithme du pivot de Gauss.