



Programmation Python

# Séance 6 — NumPy

**Dr. El Hadji Bassirou TOURÉ**

Département de Mathématiques et Informatique, Faculté des Sciences et  
Techniques, Université Cheikh Anta Diop de Dakar

---

<https://ebtoure.sn/cours/programmation-python/chapitres/06-numpy.html>

Version du 11 octobre 2026



Version en ligne

# Programmation Python

## — Séance 6

NumPy

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et Techniques, Université Cheikh Anta Diop de Dakar

# Objectifs de la séance

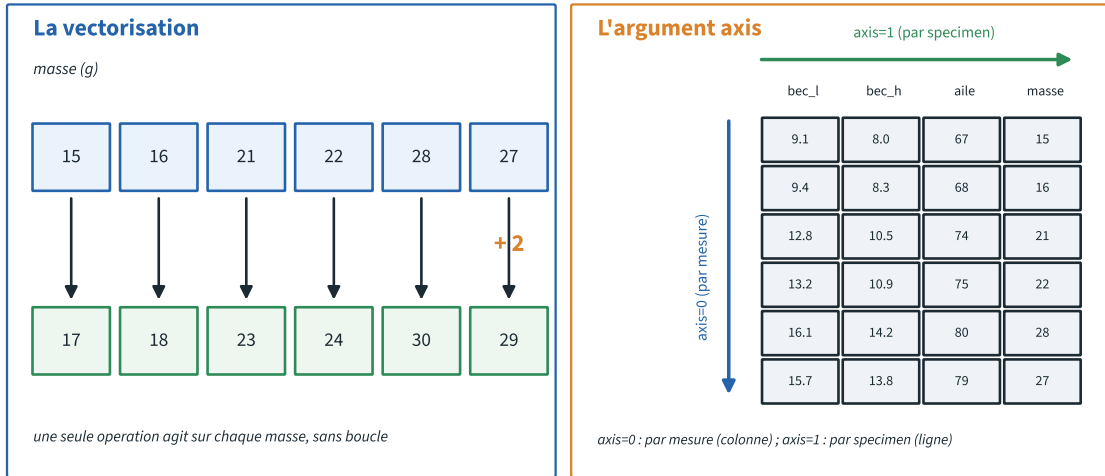
## Contenu

On entre dans la **science des données** avec un exemple de **morphométrie** : des mesures relevées sur des spécimens de pinsons (bec, aile, masse). **numpy** fournit le **tableau** (**array**), sur lequel on calcule **sans boucle**. C'est l'outil sur lequel repose **pandas** (Séance 7) et, plus loin, le Machine Learning.

- **Le tableau** : **np.array**, types homogènes.
- **Vectorisation, agrégations, filtrage** sur une mesure (1D).
- **Tableaux 2D** et **axis** : la table complète spécimens  $\times$  mesures.
- **Broadcasting** et **normalisation (z-score)** : le pont vers le ML.

L'entraînement se fait dans le **notebook**, avec une cellule « À vous » après chaque notion.

# Deux idées centrales



La vectorisation (une opération agit sur chaque valeur, sans boucle) et l'argument `axis` (résumer la table par mesure ou par spécimen).

# Le tableau NumPy

# L'idée : un tableau pour le calcul

## L'idée en une phrase

Un **tableau** NumPy est une collection de nombres **de même type**, conçue pour le calcul. On importe NumPy sous le nom `np`, et on crée un tableau avec `np.array(...)`. Ici, la **masse** (en g) de 6 spécimens de pinsons.

Le tableau ressemble à une liste, mais il autorise des opérations **globales** (sur tous les éléments à la fois) qui n'existent pas sur les listes.

# Créer un tableau

```
import numpy as np

masse = np.array([15, 16, 21, 22, 28, 27])
print(masse)
```

RÉSULTAT →

```
[15 16 21 22 28 27]
```

`np` est le nom conventionnel de NumPy. Le tableau s'affiche sans virgules.

# Des éléments de même type

```
bec = np.array([9.1, 8.0, 12.8])  
print(bec)
```

RÉSULTAT →

```
[ 9.1  8.  12.8]
```

Un tableau ne contient qu'un seul type. Comme les mesures de bec sont décimales, tout le tableau est **flottant** (le point après chaque nombre le signale).

# La vectorisation

# L'idée : calculer sans boucle

## L'idée en une phrase

Une opération sur un tableau (+, \*, une comparaison) s'applique à **chaque élément d'un seul coup**, sans boucle. C'est la **vectorisation** : un code plus court, plus lisible, et bien plus rapide.

# Opérer sur tout le tableau

```
print(masse + 2)
print(masse * 2)
```

RÉSULTAT →

```
[17 18 23 24 30 29]
[30 32 42 44 56 54]
```

`masse + 2` ajoute 2 à chaque masse ; `masse * 2` double chaque masse. Aucune boucle.

# La boucle de la Séance 3, en une ligne

```
# Facon Seance 3 : une boucle
total = 0
for m in masse:
    total = total + m
print(total)

# Facon NumPy : vectorise
print(masse.sum())
```

RÉSULTAT →

```
129
129
```

Même résultat : NumPy remplace la boucle d'accumulation par un simple `.sum()`.

# Piège — \* agit élément par élément

Ce n'est pas un produit matriciel

`masse * masse` élève **chaque** masse au carré, case par case.

```
print(masse * masse)
```

RÉSULTAT →

```
[225 256 441 484 784 729]
```

$15^2 = 225$ ,  $16^2 = 256$ , etc. Le produit matriciel, lui, s'écrit avec `@` — hors de notre programme.

# Les agrégations

# Résumer un tableau en un nombre

```
print(masse.sum())  
print(masse.mean())  
print(masse.min())  
print(masse.max())
```

RÉSULTAT →

```
129  
21.5  
15  
28
```

La masse **moyenne** vaut 21.5 g. L'**écart-type** mesure la dispersion ; il tombe rarement rond, on l'arrondit :

```
print(round(masse.std(), 2))
```

```
4.92
```

# Le filtrage booléen

# Sélectionner des spécimens : un masque

```
print(masse >= 20)
print(masse[masse >= 20])
print((masse >= 20).sum())
```

RÉSULTAT →

```
[False False  True  True  True  True]
[21 22 28 27]
4
```

La comparaison produit un **masque** de `True/False` ; placé entre crochets, il **sélectionne** les spécimens d'au moins 20 g ; `.sum()` sur un masque les **compte** (4, sans doute les plus grandes espèces).

# Les tableaux 2D

# L'idée : la table complète

## L'idée en une phrase

On a mesuré **4 caractères** par spécimen. On les range dans un tableau **2D** : une **ligne** par spécimen, une **colonne** par mesure (longueur de bec, hauteur de bec, aile, masse). `shape` donne le format (`spécimens, mesures`).

# Créer un tableau 2D

```
# colonnes : bec_long, bec_haut, aile, masse
X = np.array([[ 9.1,  8.0, 67, 15],
              [ 9.4,  8.3, 68, 16],
              [12.8, 10.5, 74, 21],
              [13.2, 10.9, 75, 22],
              [16.1, 14.2, 80, 28],
              [15.7, 13.8, 79, 27]])

print(X)
print(X.shape)
```

RÉSULTAT →

```
[[ 9.1  8.  67. 15. ]
 [ 9.4  8.3 68. 16. ]
 [12.8 10.5 74. 21. ]
 [13.2 10.9 75. 22. ]
 [16.1 14.2 80. 28. ]
 [15.7 13.8 79. 27. ]]
(6, 4)
```

# Accéder aux spécimens et aux mesures

```
print(X[0, 0])  
print(X[0])  
print(X[:, 0])  
print(X[:, 3])
```

RÉSULTAT →

```
9.1  
[ 9.1  8.  67. 15. ]  
[ 9.1  9.4 12.8 13.2 16.1 15.7]  
[15. 16. 21. 22. 28. 27.]
```

[spécimen, mesure] pour une valeur ; `X[0]` un spécimen entier ; `X[:, 0]` la colonne des becs ; `X[:, 3]` celle des masses (le `:` veut dire « tous les spécimens »).

# L'argument axis

# L'idée : dans quel sens résumer

## L'idée en une phrase

Sur un tableau 2D, une agrégation se fait **par colonne** ou **par ligne**, selon `axis`. `axis=0` descend les **colonnes** : une valeur par **mesure**. `axis=1` parcourt les **lignes** : une valeur par **spécimen**.

# Moyenne par mesure, moyenne par spécimen

```
print(X.mean(axis=0).round(2))  
print(X.mean(axis=1).round(2))
```

RÉSULTAT →

```
[12.72 10.95 73.83 21.5 ]  
[24.78 25.42 29.58 30.28 34.58 33.88]
```

`axis=0` : moyenne de **chaque mesure** (bec 12.72 mm, aile 73.83 mm, masse 21.5 g) — ce qui a un sens. `axis=1` mélange des unités (mm et g) : correct, mais **peu parlant** — d'où la normalisation, juste après.

# Broadcasting et normalisation

# L'idée : opérer entre un tableau et un vecteur

## L'idée en une phrase

Le **broadcasting** permet d'opérer entre un tableau 2D et un vecteur : le vecteur est appliqué à chaque ligne. Ici, on retire à **chaque mesure** sa propre moyenne (calculée par `axis=0`).

# Retirer à chaque mesure sa moyenne

```
print((X - X.mean(axis=0)).round(2))
```

RÉSULTAT →

```
[[-3.62 -2.95 -6.83 -6.5 ]  
 [-3.32 -2.65 -5.83 -5.5 ]  
 [ 0.08 -0.45  0.17 -0.5 ]  
 [ 0.48 -0.05  1.17  0.5 ]  
 [ 3.38  3.25  6.17  6.5 ]  
 [ 2.98  2.85  5.17  5.5 ]]
```

Le vecteur des moyennes par colonne est soustrait à **chaque ligne** : NumPy l'étend automatiquement à toute la hauteur du tableau.

# Normaliser (z-score) : le pont vers le ML

## Des unités différentes

Bec et aile en mm, masse en g : pour comparer les mesures, on **normalise** chaque colonne — retirer la moyenne, diviser par l'écart-type.

```
X_reduit = (X - X.mean(axis=0)) / X.std(axis=0)
print(X_reduit.std(axis=0).round(2))
```

RÉSULTAT →

```
[1.  1.  1.  1.]
```

Après normalisation, **chaque mesure** a une moyenne de 0 et un écart-type de 1 : toutes sont ramenées à une échelle commune, quelle que soit leur unité. Ce geste est **systématique** en morphométrie comme avant d'entraîner un modèle.

# Synthèse

# Pièges fréquents

- `*` n'est **pas** le produit matriciel : il multiplie **élément par élément**.
- Le sens de `axis` : `axis=0` **par mesure** (colonne), `axis=1` **par spécimen** (ligne).
- **Mélanger des unités** : moyenner des mesures d'unités différentes (mm et g) n'a pas de sens ; on **normalise** d'abord.
- Un tableau a des **types homogènes** : mélanger entiers et flottants donne des flottants.

# À retenir — Séance 6

## L'essentiel

**Créer :** `import numpy as np` puis `np.array(...)` ; éléments de **même type**.

**Vectorisation :** une opération s'applique à **tout le tableau**, sans boucle.

**Agrégations :** `.sum()`, `.mean()`, `.std()`, `.min()`, `.max()`.

**Filtrage :** masque `tableau >= v`, puis `tableau[masque]`.

**2D :** `shape` donne (`spécimens`, `mesures`) ; accès `[i, j]`, `[:, j]` (mesure), `[i, :]` (spécimen).

**axis :** `axis=0` par mesure, `axis=1` par spécimen.

**Normalisation (z-score) :** `(X - X.mean(axis=0)) / X.std(axis=0)` ramène chaque mesure à une moyenne de 0 et un écart-type de 1.

# Prochaine séance

## Séance 7 — pandas

NumPy calcule sur des tableaux de nombres. `pandas`, bâti sur NumPy, ajoute des **colonnes nommées** et une colonne `espece` : le **DataFrame**, la table de données de la bio-informatique. On y manipulera la table complète des pinsons — tris, groupes par espèce, valeurs manquantes.

À faire d'ici là : refaire les cellules « À vous » du notebook de la séance.

# Références

- Harris, C. et al. — *Array programming with NumPy*, Nature, 2020.
- VanderPlas, J. — *Python Data Science Handbook*, 2e éd., O'Reilly, 2023.
- McKinney, W. — *Python for Data Analysis*, 3e éd., O'Reilly, 2022.
- NumPy — *NumPy: the absolute basics for beginners* (documentation), 2024.

# Ressources de la séance

- **Notebook** · [TP 6 — NumPy](#) (16 Ko)