NumPy

Pour l'analyse de données :
Guide complet et exercices résolus

Par
Bernard MUTUALA Nkota
Analyste de données
Excel | SQL | Power BI | Python

Kinshasa - 2025

Partie I : LES BASES FONDAMENTALES DE NUMPY

1.1. Introduction à NumPy

1.1.1. Qu'est-ce que NumPy ?

NumPy (Numerical Python) est une bibliothèque Python spécialement conçue pour le calcul numérique et scientifique.

Elle fournit une structure de données appelée **ndarray** (*n-dimensional array*), beaucoup plus rapide et efficace que les listes Python classiques.

NumPy permet de :

- ✓ *Manipuler de grands ensembles de données numériques ;*
- ✓ *Faire des calculs mathématiques (somme, moyenne, variance, racine carrée, etc.) ;*
- ✓ *Effectuer des opérations sur des matrices ou des vecteurs ;*
- ✓ *Servir de base à d'autres bibliothèques comme Pandas, Matplotlib, Seaborn, et Scikit-learn.*

En résumé, NumPy est le cœur de la Data Science en Python.

1.1.2. Pourquoi NumPy est essentiel pour un analyste de données ?

Problème avec les listes Python	***Solution avec NumPy***
Lent pour les calculs massifs	Très rapide grâce à des opérations vectorisées
Pas optimisé pour le calcul scientifique	Optimisé en langage C en arrière-plan
Difficile à manipuler en tableau	Manipule facilement des matrices multidimensionnelles

Exemple :

```
import numpy as np
# Liste Python
liste = [1, 2, 3, 4, 5]
# Tableau NumPy
tableau = np.array([1, 2, 3, 4, 5])
# Calculons la somme
print(sum(liste))       # Méthode Python
print(np.sum(tableau))  # Méthode NumPy
```

En effet, le résultat pour le code précédent va bien-sûr être identique, mais NumPy est jusqu'à 100 fois plus rapide sur de gros volumes de données.

1.2. Le tableau NumPy (ndarray)

1.2.1. Définition

Un ndarray (n-dimensional array) est une structure de données capable de stocker plusieurs valeurs numériques du même type dans une seule variable.

Exemple simple:

```
import numpy as np
a = np.array([10, 20, 30, 40])
print(a)
```

Résultat :

```
[10 20 30 40]
```

1.2.2. Créer un tableau NumPy

a) À partir d'une liste

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

b) Tableaux de zéros et de uns

```
np.zeros((2, 3))  # 2 lignes, 3 colonnes, remplis de 0
np.ones((3, 2))   # 3 lignes, 2 colonnes, remplis de 1
```

c) Tableau vide

```
np.empty((2, 2))  # valeurs aléatoires selon la mémoire
```

d) Séquences automatiques

```
np.arange(0, 10, 2)   # de 0 à 10, pas de 2
np.linspace(0, 1, 5)  # 5 valeurs entre 0 et 1
```

e) Aléatoires

```
np.random.randint(1, 10, size=(3, 3))
```

1.3. Les principales propriétés d'un tableau NumPy

```
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a.ndim)   # nombre de dimensions
print(a.shape)  # (lignes, colonnes)
print(a.size)   # nombre total d'éléments
print(a.dtype)  # type de données
```

Le tableau suivant décrit chacune de ces précédentes propriétés et en donne un exemple pour chacune.

Propriété	*Description*	*Exemple*
ndim	Nombre de dimensions	2
shape	Forme du tableau (lignes, colonnes)	(2, 3)
Size	Nombre total d'éléments	6
dtype	Type de données	int64 ou float64

1.4. Indexation et sélection des éléments

1.4.1. Indexation simple

```
a = np.array([10, 20, 30, 40, 50])
print(a[0])  # premier élément
print(a[-1]) # dernier élément
```

1.4.2. Indexation 2D

```
b = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
print(b[0, 1])  # ligne 0, colonne 1
print(b[2, 2])  # ligne 2, colonne 2
```

1.4.3. Slicing (tranchage)

```
a = np.array([10, 20, 30, 40, 50])
print(a[1:4])    # du 2e au 4e élément
print(a[:3])     # du début au 3e
print(a[::2])    # un élément sur deux
```

1.4.4. Slicing multidimensionnel

```
b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(b[:2, 1:])  # les 2 premières lignes, colonnes 1 à fin
```

1.5. Opérations sur les tableaux

1.5.1. Opérations arithmétiques

```
a = np.array([1, 2, 3])
b = np.array([10, 20, 30])
print(a + b)
print(a * b)
print(a ** 2)
```

Important : NumPy permet le calcul vectorisé, c'est-à-dire sans boucle for.

1.5.2. Fonctions mathématiques intégrées

```
a = np.array([1, 4, 9, 16])
print(np.sqrt(a))   # racine carrée
print(np.exp(a))    # exponentielle
print(np.log(a))    # logarithme
```

1.5.3. Statistiques de base

```
data = np.array([5, 10, 15, 20])
print(np.mean(data))   # moyenne
print(np.median(data)) # médiane
print(np.std(data))    # écart-type
print(np.var(data))    # variance
print(np.min(data))    # minimum
print(np.max(data))    # maximum
```

1.6. Formes et dimensions

1.6.1. Modifier la forme d'un tableau

```
a = np.arange(12)
b = a.reshape(3, 4)
print(b)
```

1.6.2. Aplatir un tableau

```
b.flatten()  # transforme en 1D
```

1.6.3. Ajouter

```
a = np.array([1, 2, 3])
a_2d = a[:, np.newaxis]  # ajoute une colonne
print(a_2d.shape)        # (3, 1)
```

1.6.4. Fusionner des tableaux

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
print(np.vstack((a, b)))  # empiler verticalement
print(np.hstack((a, b.T)))  # empiler horizontalement
```

Application réelle (*mini-exemple*)

Contexte : Comparer la performance de deux agences :

```
ventes = np.array([12000, 18000, 21000])
depenses = np.array([8000, 11000, 15000])
profit = ventes - depenses
print("Profit moyen :", np.mean(profit))
```

Résultat :

```
Profit moyen : 4666.67
```

Partie II : LES FONCTIONS AVANCÉES DE NUMPY

2.1. Génération de données aléatoires

2.1.1. Le sous-module np.random

Le module np.random permet de générer des nombres aléatoires, très utiles pour :

- ✓ *Créer des échantillons de test ;*
- ✓ *Simuler des données réelles (ventes, scores, revenus, etc.) ;*
- ✓ *Tester des modèles statistiques.*

2.1.2. Les fonctions principales

a) np.random.randint()

Crée des entiers aléatoires dans une plage donnée.

```
np.random.randint(low=10, high=100, size=(3, 4))
```

Le tableau suivant décrit chaque paramètre ce cette fonction :

Paramètre	***Description***
Low	Valeur minimale incluse
High	Valeur maximale incluse
Size	Taille du tableau (lignes, colonnes)

Exemple :

```
ventes = np.random.randint(1000, 10000, size=(5, 3))
print(ventes)
```

Avec cet exemple, on simule les ventes de 3 produits pendant 5 jours.

b) np.random.rand()

Génère des nombres aléatoires réels entre 0 et 1 (distribution uniforme).

```
notes = np.random.rand(3, 4)
```

c) np.random.randn()

Génère des nombres selon une distribution normale (gaussienne) centrée sur 0.

```
donnees = np.random.randn(1000)
```

d) np.random.seed()

Permet de reproduire les mêmes nombres aléatoires à chaque exécution (utile pour la cohérence en data science).

```
np.random.seed(42)
print(np.random.randint(1, 10, 5))
```

Mini-exercice

Question : Simule les revenus mensuels (en $) de 12 mois pour 3 magasins et calcule :

- ✓ La moyenne par magasin,
- ✓ Le mois où le revenu moyen a été le plus élevé.

Résolution :

```
np.random.seed(1)
revenus = np.random.randint(8000, 15000, size=(12, 3))
print("Revenus :\n", revenus)
print("Moyenne par magasin :", np.mean(revenus, axis=0))
print("Moyenne par mois :", np.mean(revenus, axis=1))
print("Mois le plus performant :", np.argmax(np.mean(revenus, axis=1)) + 1)
```

Résultat attendu :

```
Moyenne par magasin : [11200, 10400, 11800]
Mois le plus performant : 8
```

2.2. Trier, rechercher et filtrer des données

2.2.1. np.sort()

Trie un tableau sans modifier l'original.

```
a = np.array([15, 3, 8, 1])
print(np.sort(a))
```

2.2.2. a.sort()

Trie directement le tableau existant (modifie l'original).

2.2.3. np.argsort()

Retourne les indices du tri (très utile pour trier des colonnes associées).

```
prix = np.array([120, 80, 100])
produits = np.array(["A", "B", "C"])
indices = np.argsort(prix)
print(produits[indices])
```

Résultat : ['B' 'C' 'A'] (trié selon les prix croissants).

2.2.4. Filtrer des valeurs

```
revenus = np.array([5000, 8000, 12000, 4000, 15000])
print(revenus[revenus > 7000])
Résultat : [8000 12000 15000]
```

2.2.5. Rechercher des indices

```
a = np.array([2, 5, 8, 5, 10])
print(np.where(a == 5))
```

Résultat : (array([1, 3]),)

2.3. Gestion des valeurs manquantes

2.3.1. Identifier les valeurs manquantes

```
a = np.array([10, np.nan, 20, np.nan, 30])
print(np.isnan(a))
```

2.3.2. Supprimer les NaN

```
a_clean = a[~np.isnan(a)]
```

2.3.3. Remplacer les NaN par la moyenne

```
a[np.isnan(a)] = np.nanmean(a)
print(a)
```

2.4. Statistiques et agrégations avancées

2.4.1. Agrégations

```
data = np.random.randint(10, 100, (5, 3))
print("Somme :", np.sum(data))
print("Moyenne :", np.mean(data))
print("Min :", np.min(data, axis=0))  # par colonne
print("Max :", np.max(data, axis=1))  # par ligne
```

2.4.2. Corrélation et covariance

```
x = np.array([10, 20, 30, 40, 50])
y = np.array([5, 10, 15, 20, 25])
print("Corrélation :", np.corrcoef(x, y))
print("Covariance :", np.cov(x, y))
```

2.5. Sauvegarder et charger des tableaux

2.5.1. Sauvegarder avec **np.save()**

```
data = np.array([1, 2, 3])
np.save("mon_fichier.npy", data)
```

2.5.2. Charger avec np.load()

```
reloaded = np.load("mon_fichier.npy")
```

2.5.3. Sauvegarder plusieurs tableaux

```
a = np.arange(5)
b = np.arange(10, 15)
np.savez("multi_tableaux.npz", tableau1=a, tableau2=b)
data = np.load("multi_tableaux.npz")
print(data["tableau1"])
print(data["tableau2"])
```

2.6. Mini-projet pratique : analyse d'une entreprise fictive

Contexte :

Une entreprise veut connaître :

- ✓ Son chiffre d'affaires mensuel moyen,
- ✓ Son taux de croissance et
- ✓ Le mois le plus rentable.

Résolution :

```
np.random.seed(42)
ventes = np.random.randint(10000, 25000, size=12)
depenses = np.random.randint(5000, 15000, size=12)
profits = ventes - depenses
print("Ventes :", ventes)
print("Dépenses :", depenses)
print("Profits :", profits)
print("Moyenne profit :", np.mean(profits))
print("Mois le plus rentable :", np.argmax(profits) + 1)
croissance = np.diff(profits)
print("Croissance mensuelle :", croissance)
```

Résultats possibles :

```
Moyenne profit : 7850.33
Mois le plus rentable : 7
Croissance positive sur 8 mois.
```

2.7. Exercices d'application (avec corrections)

Exercice 1 :

Créer un tableau des âges [22, 25, 28, 35, 40, 29, 31, 23, 30]
Trouver la moyenne, la médiane, et les âges supérieurs à la moyenne.

Résolution :

```
ages = np.array([22, 25, 28, 35, 40, 29, 31, 23, 30])
print(np.mean(ages))
print(np.median(ages))
print(ages[ages > np.mean(ages)])
```

Exercice 2 :

Simuler les ventes hebdomadaires (en $) de 4 produits pendant 6 semaines. Calculer la moyenne par produit et trouver le produit le plus performant.

Résolution :

```
np.random.seed(0)
ventes = np.random.randint(500, 1500, size=(6, 4))
print(ventes)
moyennes = np.mean(ventes, axis=0)
print("Moyennes :", moyennes)
print("Produit le plus performant :", np.argmax(moyennes) + 1)
```

Exercice 3 :

Créer un tableau 4x4 avec des valeurs de 10 à 25. Extraire la diagonale et calculer la somme.

Résolution :

```
a = np.arange(10, 26).reshape(4, 4)
print(a)
print("Diagonale :", np.diag(a))
print("Somme :", np.sum(np.diag(a)))
```

RECUEIL D'EXERCICES, leurs RESOLUTIONS et EXPLICATIONS

1. Niveau facile

Objectif : Maîtriser les bases de NumPy

Exercice 1 : Créer un tableau simple

Énoncé :

Crée un tableau NumPy contenant les nombres 1 à 10, puis affiche :

1. Sa forme (shape)
2. Son type de données (dtype)
3. Sa dimension (ndim)

Résolution :

```
import numpy as np
# Création du tableau
arr = np.arange(1, 11)
# Affichages
print(arr)
print("Shape :", arr.shape)
print("Type :", arr.dtype)
print("Dimension :", arr.ndim)
```

Explications :

- ✓ **np.arange**(1, 11) : génère les nombres de 1 à 10.
- ✓ **shape** : indique la taille du tableau (10 éléments).
- ✓ **dtype** : type des valeurs (par défaut int64).
- ✓ **ndim** : nombre de dimensions (ici 1D).

_____ o _____

Exercice 2 : Créer un tableau de zéros et de uns

Énoncé :

Crée un tableau de 5 lignes et 3 colonnes rempli de zéros, puis un autre rempli de uns.

Résolution :

```
zeros = np.zeros((5, 3))
ones = np.ones((5, 3))
print("Tableau de zéros:\n", zeros)
print("Tableau de uns:\n", ones)
```

Explications :

- ✓ **np.zeros**((5, 3)) : crée une matrice 5x3 avec des 0.
- ✓ **np.ones**((5, 3)) : même chose mais avec des 1.

_____ o _____

Exercice 3 : Créer un tableau de valeurs identiques
Énoncé :

Crée un tableau 4x4 rempli de la valeur 7.

Résolution :

```
arr = np.full((4, 4), 7)
print(arr)
```

Explication :

np.full((4, 4), 7) : crée une matrice de 4x4 où chaque cellule vaut 7.

_____ o _____

Exercice 4 : Créer un tableau de nombres aléatoires
Énoncé :

Crée un tableau 3x3 de nombres aléatoires compris entre 0 et 1.

Résolution :

```
arr = np.random.rand(3, 3)
print(arr)
```

Explications :
np.random.rand(3, 3) : génère 9 nombres aléatoires uniformes entre 0 et 1. C'est très utile pour tester des algorithmes avant d'avoir de vraies données.

_____ o _____

Exercice 5 : Créer un tableau de nombres entiers aléatoires
Énoncé :

Crée un tableau 5x5 de nombres aléatoires entiers compris entre 10 et 50.

Résolution :

```
arr = np.random.randint(10, 51, (5, 5))
print(arr)
```

Explications :

np.random.randint(10, 51, (5, 5)) : les bornes sont [10, 51[, donc 10 inclus et 51 exclu. C'est très utilisé pour simuler des IDs, âges, prix, etc.

_____ o _____

Exercice 6 : Reshaper un tableau
Énoncé :

Crée un tableau contenant les nombres de 1 à 12 puis redimensionne-le en 3 lignes et 4 colonnes.

Résolution :

```
arr = np.arange(1, 13)
reshaped = arr.reshape(3, 4)
print(reshaped)
```

Explications :

reshape(3, 4) : change la forme sans modifier les valeurs. C'est très utile pour restructurer des données.

_____ o _____

Exercice 7 : Sélectionner un élément spécifique
Énoncé :

À partir du tableau suivant :
arr = np.array([[10, 20, 30], [40, 50, 60]])
Affiche l'élément 50.

Résolution :

```
arr = np.array([[10, 20, 30], [40, 50, 60]])
print(arr[1, 1])  # ligne 1, colonne 1 (indices commencent à 0)
```

Explication :

Les indices commencent à 0 : [ligne, colonne].

_____ o _____

Exercice 8 : Extraire une colonne ou une ligne
Énoncé :

À partir du tableau :
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
1. Affiche la 2e ligne
2. Affiche la 3e colonne

Résolution :

```
print("Deuxième ligne :", arr[1, :])
print("Troisième colonne :", arr[:, 2])
```

Explication :

arr[1, :] : toute la 2e ligne
arr[:, 2] : toute la 3e colonne.

_____ o _____

Exercice 9 : Trouver les statistiques de base
Énoncé :

Crée un tableau [10, 20, 30, 40, 50] et affiche : la somme, la moyenne, la valeur max et min.

Résolution :

```
arr = np.array([10, 20, 30, 40, 50])
print("Somme :", arr.sum())
print("Moyenne :", arr.mean())
print("Max :", arr.max())
print("Min :", arr.min())
```

Explication :

Ces méthodes sont utilisées dans toutes les analyses de données (pour des KPI, des indicateurs, etc.).

_____ o _____

Exercice 10 : Opérations entre tableaux
Énoncé :

Soit : a = np.array([1, 2, 3]) et b = np.array([4, 5, 6])

Calcule :

La somme a + b ; Le produit a * b et La différence a - b

Résolution :

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print("Somme :", a + b)
print("Produit :", a * b)
print("Différence :", a - b)
```

Explication :

Ces opérations sont vectorisées, donc il n'y a pas besoin de boucles. C'est la vraie puissance de NumPy !

2. Niveau moyen

Objectif : Manipuler et préparer les données efficacement

Exercice 11 : Filtrer les valeurs selon une condition

Énoncé :

Soit le tableau suivant : arr = np.array([12, 5, 7, 18, 3, 10])
Affiche uniquement les valeurs supérieures à 8.

Résolution :

```
arr = np.array([12, 5, 7, 18, 3, 10])
result = arr[arr > 8]
print(result)
```

Explication :

Le filtrage booléen est fondamental dans Numpy :

arr > 8 : crée un tableau [True, False, False, True, False, True]
arr[arr > 8] : garde uniquement les valeurs correspondantes.

_____ O _____

Exercice 12 : Remplacer des valeurs spécifiques

Énoncé :

À partir de : arr = np.array([10, 0, 5, 0, 20, 0])
Remplace toutes les valeurs égales à 0 par np.nan.

Résolution :

```
arr = np.array([10, 0, 5, 0, 20, 0], dtype=float)
arr[arr == 0] = np.nan
print(arr)
```

Explication :

On force le type float, car np.nan n'existe pas en int. C'est vraiment très utilisé lors du nettoyage des données (zéros, valeurs manquantes).

_____ O _____

Exercice 13 : Supprimer les valeurs manquantes

Énoncé :

À partir du tableau précédent (avec des NaN), supprime toutes les valeurs manquantes.

Résolution :

```
arr = arr[~np.isnan(arr)]
print(arr)
```

Explication :

np.isnan(arr) : repère les NaN (True/False).

~ : inverse la condition. Et donc, on ne garde que les valeurs valides.

_____ o _____

Exercice 14 : Trier un tableau

Énoncé :

Trie le tableau suivant dans l'ordre croissant : arr = np.array([15, 3, 9, 1, 20])

Résolution :

```
sorted_arr = np.sort(arr)
print(sorted_arr)
```

Explication :

np.sort() : renvoie un nouveau tableau trié.
Pour trier sur place **: arr.sort()**.

_____ o _____

Exercice 15 : Rechercher une valeur dans un tableau

Énoncé :

Vérifie si le nombre 50 existe dans le tableau suivant :

```
arr = Résolution :
exists = 50 in arr
print("Présent :", exists)
np.array([10, 20, 30, 40, 50])
```

Explication :

L'opérateur **in** fonctionne directement sur les tableaux NumPy.
Très pratique pour vérifier la présence d'un identifiant ou d'un code.

_____ o _____

Exercice 16 : Trouver la position du maximum et du minimum

Énoncé :

Soit : arr = np.array([8, 3, 15, 6, 12]). Trouve les indices des valeurs max et min.

Résolution :

```
print("Index du max :", np.argmax(arr))
print("Index du min :", np.argmin(arr))
```

Explication :

np.argmax() : index du maximum.
np.argmin() : index du minimum. C'est très utile pour repérer les meilleures ou pires performances.

_____ o _____

Exercice 17 : Appliquer une fonction mathématique
Énoncé :

Soit : arr = np.array([0, np.pi/2, np.pi])

Calcule le sinus, le cosinus et la tangente de chaque valeur.

Résolution :

```
print("Sinus :", np.sin(arr))
print("Cosinus :", np.cos(arr))
print("Tangente :", np.tan(arr))
```

Explication :

Les fonctions trigonométriques de Numpy travaillent sur des tableaux entiers. C'est beaucoup plus rapide que de boucler avec **math.sin()**.

_____ o _____

Exercice 18 : Calcul matriciel
Énoncé :

Soient : A = np.array([[1, 2], [3, 4]]) et B = np.array([[5, 6], [7, 8]])
Calcule le produit matriciel entre A et B.

Résolution :

```
C = np.dot(A, B)
print(C)
```

Explication :

np.dot(A, B) ou **A @ B** : multiplication matricielle. C'est utilisé dans la modélisation linéaire, les réseaux de neurones, etc.

_____ o _____

Exercice 19 : Normaliser un tableau

Énoncé :

Normalise le tableau suivant (ramène toutes les valeurs entre 0 et 1) :

arr = np.array([10, 20, 30, 40, 50])

Résolution :

```
normalized = (arr - arr.min()) / (arr.max() - arr.min())
print(normalized)
```

Explication :

Très utilisé avant les analyses statistiques ou machine learning (mise à l'échelle).

_____ o _____

Exercice 20 : Créer un tableau 2D à partir de plusieurs 1D

Énoncé :

Fusionne les tableaux suivants pour créer une matrice :
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = np.array([7, 8, 9])

Résolution :

```
M = np.vstack([a, b, c])
print(M)
```

Explication :

np.vstack() : empile les vecteurs verticalement.
np.hstack() : empile horizontalement.
C'est très utile pour combiner plusieurs séries de données.

3. Niveau difficile

Objectif : Appliquer NumPy à de vraies situations d'analyse

Exercice 21 : Remplacer les valeurs aberrantes

Énoncé :

Tu disposes d'un tableau de salaires :
salaries = np.array([500, 800, 1200, 950, 30000, 1000, 850])
Remplace toute valeur supérieure à 2000 par la moyenne des autres valeurs.

Résolution :

```
import numpy as np
salaries = np.array([500, 800, 1200, 950, 30000, 1000, 850])
# Détection des salaires normaux
mask = salaries <= 2000
# Calcul de la moyenne des valeurs normales
mean_normal = salaries[mask].mean()
# Remplacement
salaries[~mask] = mean_normal
print(salaries)
```

Explication :

mask : tableau booléen identifiant les valeurs normales.
~mask : inverse (les valeurs anormales).
C'est très utile dans les nettoyages de données réelles (revenus, prix, etc.).

_____ o _____

Exercice 22 : Identifier les doublons dans un tableau

Énoncé :

Soit : ids = np.array([101, 103, 101, 104, 105, 103, 106])

Affiche uniquement les identifiants dupliqués.

Résolution :

```
ids, counts = np.unique(ids, return_counts=True)
duplicates = ids[counts > 1]
print("Doublons :", duplicates)
```

Explication :

np.unique(..., return_counts=True) : renvoie les valeurs uniques et leur fréquence.
counts > 1 : repère les répétitions.

_____ o _____

Exercice 23 : Détecter les valeurs manquantes simulées

Énoncé :

Soit : data = np.array([12, np.nan, 15, 9, np.nan, 20, 18])

Affiche le nombre total de valeurs manquantes.

Résolution :

```
missing_count = np.isnan(data).sum()
print("Nombre de valeurs manquantes :", missing_count)
```

Explication :

np.isnan(data) : True pour les NaN.
.sum() additionne les True (1) : total des NaN.

_____ o _____

Exercice 24 : Remplacer les NaN par la médiane

Énoncé :

Toujours à partir du tableau précédent, remplace les valeurs manquantes par la médiane.

Résolution :

```
median = np.nanmedian(data)
data = np.where(np.isnan(data), median, data)
print(data)
```

Explication :

np.nanmedian() : calcule la médiane en ignorant les NaN.
np.where(condition, valeur_si_vrai, valeur_si_faux) : remplace conditionnellement.

_____ o _____

Exercice 25 : Agréger des données mensuelles

Énoncé :

Tu as 12 valeurs représentant les ventes de chaque mois :
sales = np.array([1200, 1500, 900, 1000, 1700, 2000, 2200, 2100, 1900, 2500, 2700, 3000])
Calcule les ventes totales et la moyenne trimestrielle (4 trimestres)

Résolution :

```
total = sales.sum()
quarterly_avg = sales.reshape(4, 3).mean(axis=1)
print("Total annuel :", total)
print("Moyennes trimestrielles :", quarterly_avg)
```

Explication :

reshape(4,3) : 4 trimestres × 3 mois.
mean(axis=1) : moyenne par ligne (trimestre).

_____ o _____

Exercice 26 : Détection d'anomalies statistiques
Énoncé :

Détecte les valeurs considérées comme anomalies selon l'écart-type :
data = np.array([12, 14, 13, 15, 100, 16, 14, 15])
Une anomalie est une valeur à plus de 2 écarts-types de la moyenne.

Résolution :

```
mean = data.mean()
std = data.std()
anomalies = data[np.abs(data - mean) > 2 * std]
print("Anomalies :", anomalies)
```

Explication :

np.abs(data - mean) : écart absolu.
> 2 * std : seuil statistique de détection.
C'est la méthode courante en détection de fraudes ou qualité de données.

_____ o _____

Exercice 27 : Combiner deux sources de données
Énoncé :

Deux fichiers ont fourni deux tableaux de ventes :
sales_A = np.array([100, 200, 300]) et
sales_B = np.array([400, 500, 600])
Combine-les pour obtenir un tableau unique avec les ventes totales.

Résolution :

```
total_sales = np.concatenate([sales_A, sales_B])
print(total_sales)
```

Explication :

np.concatenate() : fusionne plusieurs tableaux le long d'un axe.

C'est très utile quand on fusionne plusieurs fichiers CSV.

_____ O _____

Exercice 28 : Créer un masque de conditions multiples
Énoncé :

Soit un tableau de températures :
temps = np.array([18, 25, 32, 40, 15, 28, 22])
Affiche uniquement les valeurs entre 20 et 30 inclus.

Résolution :

```
mask = (temps >= 20) & (temps <= 30)
print(temps[mask])
```

Explication :

& : "et logique" entre conditions.
(a >= 20) & (a <= 30) : filtre une plage de valeurs.

_____ O _____

Exercice 29 : Créer un tableau simulant des données d'âge
Énoncé :

Crée un tableau simulant l'âge de 1000 personnes, compris entre 18 et 60 ans, puis calcule l'âge moyen et le nombre de personnes de plus de 50 ans.

Résolution :

```
ages = np.random.randint(18, 61, 1000)
mean_age = ages.mean()
count_over_50 = np.sum(ages > 50)
print("Âge moyen :", mean_age)
print("Nombre de +50 ans :", count_over_50)
```

Explication :

Cet exercice simule une vraie base de données démographique.

_____ O _____

Exercice 30 : Simulation d'un mini dataset et résumé statistique

Énoncé :

Simule un petit jeu de données sur les ventes :
100 produits, prix entre 10 et 500, quantités entre 1 et 20.
Calcule le chiffre d'affaires total, le prix moyen et le produit le plus cher.

Résolution :

```
prices = np.random.randint(10, 501, 100)
quantities = np.random.randint(1, 21, 100)
# Calcul du chiffre d'affaires
revenue = prices * quantities
print("Chiffre d'affaires total :", revenue.sum())
print("Prix moyen :", prices.mean())
print("Produit le plus cher :", prices.max())
```

Explication :

prices * quantities : opération vectorisée (sans boucle).
Ce genre de calcul est typique des tableaux financiers ou commerciaux.

TABLE DES MATIERES