LES BASES DE NUMPY

INTRODUCTION

Dans ce cours, nous allons nous intéresser à la bibliothèque NumPy de Python. Les scientifiques, les ingénieurs et les analystes de données sont confrontés à de nombreux défis de nos jours. Les scientifiques des données veulent pouvoir effectuer des analyses numériques sur de grands ensembles de données avec un effort de programmation minimal. Ils souhaitent également écrire un code lisible, efficace et rapide, aussi proche que possible du langage mathématique auquel ils sont habitués. Un certain nombre de solutions sont disponibles dans le monde du calcul scientifique. Python est un langage de programmation à usage général populaire qui est largement utilisé dans la communauté scientifique. Il est orienté objet et considéré comme étant d'un niveau supérieur à C ou Fortran. Il vous permet d'écrire du code lisible et propre avec un minimum de tracas. Cependant, il lui manque un équivalent MATLAB (langage de script émulé par un environnement de développement du même nom utilisé à des fins de calcul numérique) prêt à l'emploi. C'est là qu'intervient NumPy.

I. Introduction à NumPy

a) Présentation de NumPy et son rôle dans le calcul scientifique en Python

Numpy est une extension du langage de programmation python qui prend en charge de grands tableaux multidimensionnels, ainsi qu'une bibliothèque étendue de fonctions mathématiques de haut niveau. La bibliothèque contient une longue liste de fonctions mathématiques utiles, y compris certaines fonctions pour l'algèbre linéaire, la transformation de Fourier et les routines de génération de nombres aléatoires.

Le code NumPy est beaucoup plus propre que le code Python pur et il essaie d'accomplir les mêmes tâches. Il y a moins de boucles nécessaires car les opérations fonctionnent directement sur les tableaux et matrices. Les nombreuses fonctions pratiques et mathématiques facilitent également la vie.

Les tableaux de NumPy sont stockés plus efficacement qu'une structure de données équivalente en Python de base, comme une liste de listes. L'amélioration des performances s'adapte au nombre d'éléments du tableau. Pour les grands tableaux, il est vraiment rentable d'utiliser NumPy. Des fichiers pouvant atteindre plusieurs téraoctets peuvent être mappés en mémoire, ce qui entraîne lecture et écriture optimales des données.

b) Installation de NumPy

Pour installer NumPy sur Windows, vous pouvez suivre les étapes suivantes :

1. Assurez-vous d'avoir Python installé sur votre système. Vous pouvez le télécharger depuis le site officiel de Python (https://www.python.org) et suivre les instructions d'installation.
2. Ouvrez une invitation de commandes en appuyant sur la touche Windows + R, puis en tapant "cmd" suivi de la touche Entrée.
3. Dans l'invitation de commandes, saisissez la commande suivante pour installer NumPy à l'aide de l'outil de gestion de packages de Python appelé pip :
4. Copier le code

pip install numpy

Si vous avez plusieurs versions de Python installées, vous devez utiliser la commande pip correspondante à la version de Python que vous utiliserez ultérieurement.

5. Appuyez sur Entrée pour exécuter la commande. Cela téléchargera et installera NumPy sur votre système.
6. Une fois l'installation terminée, vous devriez pouvoir importer NumPy dans vos scripts Python sans erreur. Assurez-vous d'avoir une connexion Internet active lors de l'installation de NumPy, car pip télécharge les fichiers nécessaires depuis le dépôt Python Package Index (PyPI).

C'est ainsi que vous pouvez installer NumPy sur Windows en utilisant pip. Si vous rencontrez des problèmes lors de l'installation, vous vous êtes assuré que votre environnement Python est correctement configuré et à jour.

c) Création et manipulation de tableaux NumPy

Les tableaux ou les ndarrays n-dimensions sont l'objet principal de numpy utilisés pour stocker les éléments du même type de données. Le tableau NumPy est en général homogène (il existe un type de tableau spécial qui est hétérogène) les éléments du tableau doivent être du même type. L'avantage est que, si l'on sait que les éléments dans le tableau sont du même type, il est facile de déterminer la taille de stockage requise pour le tableau. Les tableaux NumPy sont indexés à partir de 0, comme en Python. Les types de données sont représentés par objets spéciaux. Nous aborderons ces objets en détail dans ce cours.

Voici différentes façons de créer un tableau NumPy :

- À partir d'une liste :

```
Entrée [1]: import numpy as np #importation de numpy

Entrée [2]: my_list = [1, 2, 3, 4, 5]
            my_array = np.array(my_list)
```

- À partir d'une séquence de nombres :

Entrée [3]:
```
my_range = np.arange(0, 10, 2)  # Créer un tableau de 0 à 10 (exclus) avec un pas de 2
```

- À partir d'une liste de listes (tableau multidimensionnel) :

Entrée [4]:
```
my_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
my_array = np.array(my_lists)
```

- En spécifiant une forme (nombre de lignes et de colonnes) :

Entrée [6]:
```
my_zeros = np.zeros((3, 4))  # Crée un tableau de zéros avec 3 lignes et 4 colonnes
my_ones = np.ones((2, 2))  # Crée un tableau de uns avec 2 lignes et 2 colonnes
my_random = np.random.random((2, 3))  # Crée un tableau de nombres aléatoires avec 2 lignes et 3 colonnes
```

Une fois que vous avez créé un tableau NumPy, vous pouvez effectuer diverses opérations de manipulation, telles que :

- Accéder aux éléments du tableau :

Entrée [7]:
```
print(my_array[0])  # Accède au premier élément du tableau
print(my_array[1, 2])  # Accède à l'élément à la deuxième ligne et troisième colonne du tableau multidimensionnel
```

```
[1 2 3]
6
```

- Effectuer des opérations mathématiques sur les tableaux :

Entrée [8]:
```
result = my_array + 2  # Ajoute 2 à chaque élément du tableau
result = np.sin(my_array)  # Calcule le sinus de chaque élément du tableau
```

- Appliquer des fonctions statistiques sur les tableaux :

Entrée [9]:
```
print(np.mean(my_array))  # Calcule la moyenne des éléments du tableau
print(np.max(my_array))  # Trouve la valeur maximale dans le tableau
```

```
5.0
9
```

d) Accès aux éléments d'un tableau et utilisation des index

Nous utilisons les index pour accéder aux éléments spécifiques du tableau. L'index 0 correspond au premier élément, l'index 1 correspond au deuxième élément, et ainsi de suite. Les index négatifs permettent d'accéder aux éléments à partir de la fin du tableau, où -1 correspond au dernier élément, -2 à l'avant-dernier élément, et ainsi de suite.
Nous pouvons également utiliser les index pour modifier les éléments du tableau. Dans l'exemple, nous avons modifié le troisième élément en lui donnant la valeur 35 et l'avant-dernier élément en lui donnant la valeur 45.

Pour accéder aux éléments d'un tableau et utiliser les index, vous pouvez utiliser la syntaxe suivante :

Entrée [1]:
```
# Création d'un tableau
mon_tableau = [10, 20, 30, 40, 50]

# Accès à un élément à l'aide de l'index
premier_element = mon_tableau[0]  # Accès au premier élément (index 0)
deuxieme_element = mon_tableau[1]  # Accès au deuxième élément (index 1)
troisieme_element = mon_tableau[2]  # Accès au troisième élément (index 2)

# Modification d'un élément à l'aide de l'index
mon_tableau[3] = 42  # Modification du quatrième élément (index 3)

# Affichage des éléments
print(premier_element)  # Affiche 10
print(deuxieme_element)  # Affiche 20
print(troisieme_element)  # Affiche 30
print(mon_tableau)  # Affiche [10, 20, 30, 42, 50]
```

```
10
20
30
[10, 20, 30, 42, 50]
```

En Python, les indices de tableau commencent à partir de zéro, ce qui signifie que le premier élément a un index de 0, le deuxième élément a un index de 1, et ainsi de suite. Vous pouvez utiliser ces indices pour accéder aux éléments spécifiques du tableau.
Notez également que vous pouvez modifier les éléments du tableau en utilisant l'index. Dans l'exemple ci-dessus, nous avons modifié le quatrième élément (index 3) pour lui donner la valeur 42.
Assurez-vous que l'index que vous utilisez est valide pour éviter les erreurs. Si vous utilisez un index en dehors de la plage valide, vous obtiendrez une erreur d'index hors limite (IndexError).

II. Opérations mathématiques et statistiques

a) Utilisation des fonctions mathématiques et statistiques de base de NumPy

NumPy est une bibliothèque très populaire en Python pour le calcul scientifique et numérique. Elle offre de nombreuses fonctions mathématiques et statistiques de base. Voici quelques exemples d'utilisation de ces fonctions avec NumPy :

Entrée [5]:
```
import numpy as np
```

Entrée [6]:
```
# Création d'un tableau NumPy
tableau = np.array([1, 2, 3, 4, 5])
```

Entrée [7]:
```
# Fonctions mathématiques de base
print(np.mean(tableau))  # Calcul de la moyenne : affiche 3.0
print(np.sum(tableau))  # Calcul de la somme : affiche 15
print(np.min(tableau))  # Trouver la valeur minimale : affiche 1
print(np.max(tableau))  # Trouver la valeur maximale : affiche 5
print(np.sqrt(tableau))  # Calcul de la racine carrée de chaque élément : affiche [1. 1.41421356 1.73205081 2. 2.23606798]
print(np.exp(tableau))  # Calcul de l'exponentielle de chaque élément : affiche [2.71828183 7.3890561 20.08553692 54.59815003
```

```
3.0
15
1
5
[1.         1.41421356 1.73205081 2.         2.23606798]
[  2.71828183   7.3890561   20.08553692  54.59815003 148.4131591 ]
```

Entrée [8]:
```
# Fonctions statistiques
print(np.median(tableau))  # Calcul de la médiane : affiche 3.0
print(np.std(tableau))  # Calcul de l'écart-type : affiche 1.4142135623730951
print(np.var(tableau))  # Calcul de la variance : affiche 2.0
print(np.percentile(tableau, 25))  # Calcul du 1er quartile : affiche 2.0
```

```
3.0
1.4142135623730951
2.0
2.0
```

Nous avons utilisé NumPy pour créer un tableau NumPy à partir d'une liste. Ensuite, nous avons appliqué plusieurs fonctions mathématiques et statistiques de base sur ce tableau :

- **np.mean()** : Calcule la moyenne des éléments du tableau.
- **np.sum()** : Calcule la somme des éléments du tableau.
- **np.min()** : Trouve la valeur minimale du tableau.
- **np.max()** : Trouve la valeur maximale du tableau.
- **np.sqrt()** : Calcule la racine carrée de chaque élément du tableau.
- **np.exp()** : Calcule l'exponentielle de chaque élément du tableau.
- **np.median()** : Calcule la médiane du tableau.
- **np.std()** : Calcule l'écart-type du tableau.
- **np.var()** : Calcule la variance du tableau.
- **np.percentile()** : Calcule le percentile spécifié du tableau (dans cet exemple, le 1er quartile).
  Les percentiles sont les valeurs de la variable qui divisent la population ou la variable continue en 100 groupes égaux en nombre

Ces fonctions sont un échantillon des fonctionnalités offertes par NumPy pour le traitement mathématique et statistique des tableaux. NumPy propose bien d'autres fonctions pour répondre à des besoins spécifiques. Vous pouvez consulter la documentation officielle de NumPy pour en savoir plus sur les possibilités offertes par cette bibliothèque : https://numpy.org/doc/

b) Utilisation des fonctions de trigonométrie et d'algèbre linéaire

NumPy (Numerical Python) est une bibliothèque très populaire en Python utilisée pour le calcul scientifique et numérique. Elle fournit de nombreuses fonctionnalités mathématiques puissantes pour effectuer des opérations sur des tableaux multidimensionnels. NumPy offre également des fonctions pour la trigonométrie et l'algèbre linéaire. Voici quelques exemples d'utilisation de ces fonctions :

- **Fonctions trigonométriques**

Entrée [1]:
```
import numpy as np
```

Entrée [2]:
```
angle = np.pi / 4  # Un angle de pi/4 radians (45 degrés)

# Fonctions trigonométriques
print(np.sin(angle))  # Calcule le sinus de l'angle : affiche 0.7071067811865476
print(np.cos(angle))  # Calcule le cosinus de l'angle : affiche 0.7071067811865476
print(np.tan(angle))  # Calcule la tangente de l'angle : affiche 0.9999999999999999
print(np.arcsin(0.5))  # Calcule l'arc sinus de 0.5 : affiche 0.5235987755982989 (en radians)
print(np.arccos(0.5))  # Calcule l'arc cosinus de 0.5 : affiche 1.0471975511965979 (en radians)
print(np.arctan(1))  # Calcule l'arc tangente de 1 : affiche 0.7853981633974483 (en radians)
```

```
0.7071067811865476
0.7071067811865476
0.9999999999999999
0.5235987755982989
1.0471975511965979
0.7853981633974483
```

Nous avons utilisé différentes fonctions trigonométriques disponibles dans NumPy. Nous avons calculé le sinus, le cosinus et la tangente d'un angle spécifique (pi/4 radians). Nous avons également calculé les arcsinus, arccosinus et arctangente de certaines valeurs.

- **Fonctions pour l'algèbre linéaire**

Entrée [4]:
```
# Création de matrices
matrice1 = np.array([[1, 2], [3, 4]])
matrice2 = np.array([[5, 6], [7, 8]])

# Produit matriciel
produit = np.dot(matrice1, matrice2)
print(produit)  # Affiche [[19 22] [43 50]]

# Transposée d'une matrice
transposee = np.transpose(matrice1)
print(transposee)  # Affiche [[1 3] [2 4]]

# Déterminant d'une matrice
determinant = np.linalg.det(matrice1)
print(determinant)  # Affiche -2.0

# Résolution d'un système d'équations linéaires
coefficients = np.array([[2, 1], [1, -1]])
valeurs = np.array([5, 1])
solution = np.linalg.solve(coefficients, valeurs)
print(solution)  # Affiche [2. -1.]
```

```
[[19 22]
 [43 50]]
[[1 3]
 [2 4]]
-2.0000000000000004
[2. 1.]
```

Nous avons utilisé quelques fonctionnalités d'algèbre linéaire fournies par NumPy :

- **np.dot()** : Calcule le produit matriciel de deux matrices.
- **np.transpose()** : Calcule la transposée d'une matrice.
- **np.linalg.det()** : Calcule le déterminant d'une matrice.
- **np.linalg.solve()** : Résout un système d'équations linéaires.

Ces fonctions vous permettent de réaliser diverses opérations mathématiques avancées dans le domaine de la trigonométrie et de l'algèbre linéaire en utilisant NumPy. Consultez la documentation officielle de NumPy pour découvrir davantage de fonctionnalités dans ces domaines.

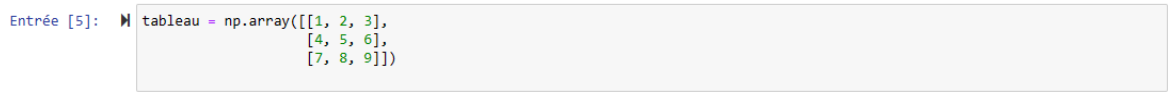
III. Indexation et troncature

a) Utilisation de l'indexation avancée pour extraire des éléments d'un tableau

NumPy offre diverses fonctionnalités d'indexation et de troncature pour manipuler des tableaux multidimensionnels de manière flexible.

ndarraysx[obj]peut être indexé en utilisant la syntaxe Python standard , où *x* est le tableau et *obj* la sélection. Il existe différents types d'indexation disponibles selon *obj* : indexation de base, indexation avancée et accès aux champs.

1. **Indexation simple** :
    - Accès aux éléments : Vous pouvez accéder à un élément spécifique en spécifiant les indices correspondants pour chaque dimension du tableau. Par exemple, **tableau[i, j]** accède à l'élément à la position (i, j) dans un tableau à deux dimensions.
    - Accès à une ligne ou une colonne : Vous pouvez accéder à une ligne ou une colonne spécifique en utilisant une indexation partielle. Par exemple, **tableau[i, :]** renvoie la ième ligne d'un tableau à deux dimensions.
    - Accès à une plage d'indices : Vous pouvez utiliser la notation de tranche (slicing) pour accéder à une plage d'indices. Par exemple, **tableau[i:j]** renvoie les éléments du tableau de l'index i inclus à l'index j exclus.



- Accès à un élément spécifique :
    - **tableau[0, 0]** renvoie le premier élément du tableau, soit 1.
    - **tableau[1, 2]** renvoie l'élément à la deuxième ligne et troisième colonne du tableau, soit 6.
- Accès à une ligne ou une colonne :
    - **tableau[1, :]** renvoie la deuxième ligne du tableau : **[4, 5, 6]**.
    - **tableau[:, 2]** renvoie la troisième colonne du tableau : **[3, 6, 9]**.
- Accès à une plage d'indices :
    - **tableau[0:2, :]** renvoie les deux premières lignes du tableau : **[[1, 2, 3], [4, 5, 6]]**.
    - **tableau[:, 1:3]** renvoie les deux dernières colonnes du tableau : **[[2, 3], [5, 6], [8, 9]]**.

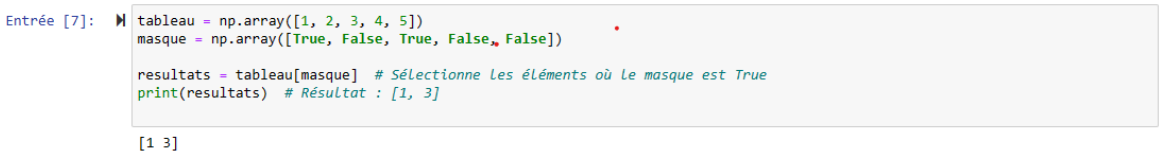
Il est important de noter que l'indexation en NumPy commence à 0, c'est-à-dire que le premier élément a l'indice 0. De plus, la notation de tranche (slicing) utilise l'indice de fin exclusif, ce qui signifie que l'élément à l'indice de fin n'est pas inclus dans la tranche.

L'indexation simple en NumPy vous permet de sélectionner des éléments individuels, des lignes, des colonnes ou des sous-tableaux en spécifiant les indices appropriés pour chaque dimension.

2. **Indexation avancée** :

L'indexation avancée en NumPy offre des fonctionnalités plus puissantes pour accéder et manipuler les données dans un tableau multidimensionnel. Elle permet notamment l'utilisation d'indexation booléenne et d'indexation entière. Voici un aperçu de ces techniques :

Indexation booléenne : L'indexation booléenne permet de sélectionner des éléments d'un tableau en utilisant un tableau de booléens de la même forme que le tableau d'origine. Les éléments correspondants aux indices où la valeur booléenne est **True** sont sélectionnés.



Indexation entière : L'indexation entière permet de sélectionner des éléments spécifiques d'un tableau en utilisant un tableau d'indices entiers. Les éléments correspondant aux indices spécifiés sont sélectionnés dans l'ordre spécifié par le tableau d'indices.

Entrée [8]:
```
tableau = np.array([10, 20, 30, 40, 50])
indices = np.array([1, 3])

resultats = tableau[indices]  # Sélectionne les éléments aux indices 1 et 3
print(resultats)  # Résultat : [20, 40]
```

```
[20 40]
```

L'indexation avancée peut également être utilisée en combinaison avec d'autres techniques d'indexation pour effectuer des sélections plus complexes. Par exemple, vous pouvez utiliser des tableaux booléens et entiers simultanément pour sélectionner des sous-ensembles spécifiques de données.

L'indexation avancée offre une grande flexibilité pour accéder et manipuler les données dans les tableaux NumPy. Elle est largement utilisée dans les domaines de la data science et du calcul scientifique pour filtrer, extraire et manipuler des données selon des conditions et des critères spécifiques.

b) Manipulation des dimensions et troncature de tableaux

NumPy offre des fonctionnalités pour manipuler les dimensions des tableaux ainsi que pour tronquer les valeurs d'un tableau. Voici un aperçu de ces fonctionnalités :

Manipulation des dimensions des tableaux :

Remodelage (Reshaping) : Vous pouvez changer la forme d'un tableau en utilisant la méthode **reshape()** ou la fonction **np.reshape().** Cela permet de modifier le nombre de dimensions et la taille de chaque dimension du tableau.

Entrée [9]:
```
tableau = np.array([1, 2, 3, 4, 5, 6])
nouveau_tableau = np.reshape(tableau, (2, 3))

print(nouveau_tableau)
# Résultat :
# [[1 2 3]
#  [4 5 6]]
```

```
[[1 2 3]
 [4 5 6]]
```

Aplatir (Flattening) : Vous pouvez aplatir un tableau multidimensionnel en utilisant la méthode **flatten()** ou la fonction **np.flatten()**. Cela transforme le tableau en une seule dimension.

Entrée [10]:
```
tableau = np.array([[1, 2, 3],
                    [4, 5, 6]])

aplatir_tableau = tableau.flatten()
print(aplatir_tableau)  # Résultat : [1 2 3 4 5 6]
```

```
[1 2 3 4 5 6]
```

Transposition : Vous pouvez permuter les dimensions d'un tableau en utilisant la méthode **transpose()** ou l'attribut **T**. Cela permet d'échanger les lignes avec les colonnes.

Troncature de tableaux :

Troncature de valeurs : Vous pouvez tronquer les valeurs d'un tableau en utilisant les fonctions **np.floor()**, **np.ceil()** et **np.round()**. Ces fonctions renvoient respectivement les valeurs tronquées à l'entier inférieur, à l'entier supérieur et à l'entier le plus proche.

Entrée [11]:
```
tableau = np.array([1.3, 2.7, 4.1])

tronc_inf = np.floor(tableau)
tronc_sup = np.ceil(tableau)
tronc_arrondi = np.round(tableau)

print(tronc_inf)  # Résultat : [1. 2. 4.]
print(tronc_sup)  # Résultat : [2. 3. 5.]
print(tronc_arrondi)  # Résultat : [1. 3. 4.]
```

```
[1. 2. 4.]
[2. 3. 5.]
[1. 3. 4.]
```

Troncature à une précision spécifiée : Vous pouvez tronquer les valeurs d'un tableau à une précision spécifiée en utilisant la fonction **np.trunc()**. Cette fonction renvoie les valeurs tronquées au nombre de décimales spécifié par l'argument **decimals**

Entrée [12]:
```
tableau = np.array([1.234, 2.567, 3.891])

tronc_precision = np.trunc(tableau * 100) / 100
print(tronc_precision)  # Résultat : [1.23 2.56 3.89]
```

```
[1.23 2.56 3.89]
```

La manipulation des dimensions et la troncature des tableaux en NumPy permettent de transformer les données selon les besoins spécifiques de l'analyse et du traitement des données. Ces fonctionnalités sont largement utilisées dans les domaines de la data science et du calcul scientifique pour préparer, nettoyer et normaliser les données en vue de leur analyse et de leur modélisation.

c) Utilisation des masques pour filtrer les données

Les masques sont une technique puissante en NumPy pour filtrer les données en utilisant des conditions booléennes. Un masque est un tableau de booléens ayant la même forme que le tableau de données d'origine. Les valeurs True dans le masque indiquent les positions où la condition est satisfaite, tandis que les valeurs False indiquent les positions où la condition n'est pas satisfaite.

Entrée [13]:
```
# Créer un tableau de données
donnees = np.array([1, 2, 3, 4, 5, 6])

# Créer un masque basé sur une condition
masque = donnees > 3

# Utiliser le masque pour filtrer les données
resultats = donnees[masque]

print(resultats)  # Résultat : [4, 5, 6]
```

```
[4 5 6]
```

Nous avons créé un masque **masque** en utilisant la condition **donnees > 3**. Cela crée un tableau de booléens avec **True** là où les valeurs de **donnees** sont supérieures à 3 et **False** sinon. Ensuite, nous utilisons ce masque pour filtrer les données en utilisant **donnees[masque]**. Cela renvoie uniquement les valeurs de **donnees** où le masque est **True**, c'est-à-dire les valeurs supérieures à 3.

Vous pouvez également combiner plusieurs conditions en utilisant les opérateurs logiques tels que **&** (et), **|** (ou) et **~** (non).

Entrée [14]:
```
# Créer un tableau de données
donnees = np.array([1, 2, 3, 4, 5, 6])

# Créer un masque basé sur plusieurs conditions
masque = (donnees > 2) & (donnees < 6)

# Utiliser le masque pour filtrer les données
resultats = donnees[masque]

print(resultats)  # Résultat : [3, 4, 5]
```

```
[3 4 5]
```

Dans cet exemple, le masque est créé en combinant les conditions **donnees > 2** et **donnees < 6** à l'aide de l'opérateur **&**. Cela crée un masque qui est **True** pour les valeurs de **donnees** qui satisfont à ces deux conditions. Ensuite, nous utilisons ce masque pour filtrer les données et obtenir uniquement les valeurs qui satisfont les conditions.

Les masques sont extrêmement utiles pour effectuer des opérations de filtrage et de sélection sur les tableaux NumPy, que ce soit pour extraire des sous-ensembles de données ou pour effectuer des calculs spécifiques sur des valeurs qui satisfont certaines conditions.

IV. Diffusion et vectorisation

a) Compréhension du broadcast et son utilisation pour les opérations entre tableaux de formes différentes

Le broadcasting (diffusion) est une fonctionnalité puissante de NumPy qui permet d'effectuer des opérations entre des tableaux de formes différentes, en ajustant automatiquement les dimensions des tableaux pour les rendre compatibles. Lorsque vous effectuez une opération entre deux tableaux avec des formes différentes, NumPy applique les règles de broadcasting pour rendre les tableaux compatibles en ajustant automatiquement leurs dimensions. Les règles de broadcasting sont les suivantes :

- Si les tableaux n'ont pas le même nombre de dimensions, le tableau avec moins de dimensions est étendu avec des dimensions de taille 1 à gauche jusqu'à ce qu'il ait le même nombre de dimensions que l'autre tableau.

- Si les tableaux ont le même nombre de dimensions, mais leurs tailles de dimensions ne correspondent pas, une dimension de taille 1 est étendue dans le tableau ayant une taille de dimension inférieure pour qu'il corresponde à la taille de dimension du tableau avec la taille de dimension supérieure.

- Si après l'application des règles précédentes, une dimension d'un tableau a une taille de 1, et l'autre tableau a une taille de dimension supérieure à 1 dans cette dimension, le tableau avec la taille de dimension égale à 1 est étendu dans cette dimension pour correspondre à la taille de dimension du tableau avec la taille de dimension supérieure.

Entrée [15]:
```
# Tableau A de forme (3, 1)
A = np.array([[1], [2], [3]])

# Tableau B de forme (3, 2)
B = np.array([[4, 5], [6, 7], [8, 9]])

# Opération de broadcasting entre A et B
resultat = A + B

print(resultat)
# Résultat :
# [[ 5  6]
#  [ 8  9]
#  [11 12]]
```

```
[[ 5  6]
 [ 8  9]
 [11 12]]
```

Nous avons deux tableaux **A** et **B** avec des formes différentes. Le tableau **A** a une forme de (3, 1) et le tableau **B** a une forme de (3, 2). Lorsque nous effectuons l'opération **A + B**, le broadcasting est appliqué automatiquement. Le tableau **A** est étendu en ajoutant une dimension de taille 1 à droite, de sorte que sa forme devienne (3, 2) et il correspond à la forme de **B**. Ensuite, l'addition est effectuée entre les tableaux étendus pour obtenir le résultat final.

Le broadcasting est une fonctionnalité très utile qui permet d'effectuer des opérations vectorisées entre des tableaux de formes différentes, évitant ainsi d'avoir à effectuer des boucles manuelles. Cela simplifie le code et améliore les performances en utilisant les fonctionnalités vectorisées de NumPy. Il est largement utilisé dans le domaine du calcul scientifique et de la data science pour effectuer des calculs efficaces sur des ensembles de données multidimensionnels de différentes formes.

b) Vectorisation des opérations pour améliorer les performances

La vectorisation des opérations est une technique essentielle en NumPy pour améliorer les performances des calculs en utilisant des opérations vectorisées plutôt que des boucles explicites. La vectorisation fait la même opération sur élément par élément d'un tableau dans un code compilé. Les types des éléments n'ont pas besoin d'être déterminés à chaque fois et c'est ce qui accélère les opérations.

Voici quelques conseils pour tirer pleinement parti de la vectorisation des opérations en NumPy :

1. **Évitez les boucles explicites** : Au lieu d'utiliser des boucles **for** pour effectuer des calculs élément par élément, essayez d'exprimer les opérations sous forme de manipulations sur des tableaux entiers. Les opérations vectorisées sont généralement plus rapides car elles exploitent les capacités de calcul parallèle des processeurs modernes.
2. **Utilisez les opérations vectorisées de NumPy** : NumPy fournit de nombreuses fonctions et opérations vectorisées, telles que l'addition, la soustraction, la multiplication, la division, les fonctions trigonométriques, les fonctions exponentielles, les fonctions logarithmiques, etc. Utilisez ces fonctions plutôt que d'implémenter des versions personnalisées.
3. **Évitez les copies inutiles de tableaux** : Lorsque vous effectuez des opérations sur des tableaux, essayez de minimiser les copies de tableaux en utilisant des vues (**view**) ou des références (**reference**) si possible. Les copies de tableaux peuvent être coûteuses en termes de mémoire et de temps de calcul.
4. **Utilisez les fonctions universelles (ufuncs) de NumPy** : Les fonctions universelles de NumPy sont des fonctions optimisées pour les opérations élémentaires sur les tableaux. Elles peuvent être utilisées pour effectuer des opérations mathématiques et des réductions (somme, produit, maximum, minimum, etc.) sur des tableaux de manière efficace et vectorisée.
5. **Exploitez les diffusions (broadcasting)** : Comme expliqué précédemment, utilisez les règles de broadcasting de NumPy pour effectuer des opérations entre des tableaux de formes différentes de manière transparente et efficace.
6. **Utilisez les fonctionnalités de parallélisme** : NumPy utilise des bibliothèques sous-jacentes telles que BLAS, qui sont optimisées pour les calculs vectorisés et parallèles. Vous pouvez tirer parti de ces fonctionnalités en utilisant des implémentations optimisées de NumPy, telles que NumPy avec OpenBLAS ou NumPy avec Intel MKL (Math Kernel Library).

En suivant ces bonnes pratiques, vous pouvez améliorer considérablement les performances de vos calculs en NumPy. La vectorisation des opérations permet de tirer parti de la puissance des processeurs modernes et d'exécuter des calculs efficaces sur des ensembles de données de grande taille dans le domaine du calcul scientifique et de la data science.

V. Fonctionnalités avancées

a) Utilisation de NumPy pour effectuer des opérations d'algèbre linéaire (produit matriciel, résolution de systèmes linéaires, valeurs propres, etc.)

NumPy fournit un ensemble complet de fonctions pour effectuer des opérations d'algèbre linéaire. Voici quelques exemples d'opérations d'algèbre linéaire couramment utilisées avec NumPy :

- **Produit matriciel** :

Vous pouvez utiliser la fonction numpy.dot ou l'opérateur @ pour effectuer le produit matriciel entre deux tableaux. Par exemple :

Entrée [16]:
```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

produit = np.dot(A, B)
# Ou : produit = A @ B

print(produit)
# Résultat :
# [[19 22]
#  [43 50]]
```

```
[[19 22]
 [43 50]]
```

- **Résolution de systèmes linéaires** :

NumPy fournit la fonction **numpy.linalg.solve** pour résoudre des systèmes linéaires. Elle prend en entrée une matrice A et un vecteur b, et renvoie le vecteur x qui satisfait l'équation linéaire Ax = b. Voici un exemple :

Entrée [17]:
```
A = np.array([[2, 1], [4, -1]])
b = np.array([5, 10])

solution = np.linalg.solve(A, b)

print(solution)
# Résultat : [2. 1.]
```

```
[2.5 0. ]
```

- **Valeurs propres et vecteurs propres** :

NumPy propose la fonction **numpy.linalg.eig** pour calculer les valeurs propres et les vecteurs propres d'une matrice. Elle prend en entrée une matrice carrée et renvoie deux tableaux : un tableau contenant les valeurs propres et un tableau contenant les vecteurs propres correspondants. Voici un exemple :

Entrée [18]:
```
A = np.array([[2, 1], [1, 3]])

valeurs_propres, vecteurs_propres = np.linalg.eig(A)

print(valeurs_propres)
# Résultat : [1.38196601 3.61803399]

print(vecteurs_propres)
# Résultat :
# [[-0.85065081 -0.52573111]
#  [ 0.52573111 -0.85065081]]
```

```
[1.38196601 3.61803399]
[[-0.85065081 -0.52573111]
 [ 0.52573111 -0.85065081]]
```

Ces exemples ne couvrent qu'une petite partie des opérations d'algèbre linéaire que vous pouvez effectuer avec NumPy. NumPy offre également des fonctions pour le calcul du déterminant, de l'inverse d'une matrice, de la factorisation LU, de la factorisation QR, etc. Vous pouvez explorer la documentation officielle de NumPy pour en savoir plus sur les fonctionnalités d'algèbre linéaire disponibles.

b) Lecture et écriture de données à partir de fichiers en utilisant NumPy

NumPy offre des fonctionnalités pour lire et écrire des données à partir de fichiers. Voici quelques exemples d'utilisation de NumPy pour la lecture et l'écriture de données à partir de fichiers :

- **Lecture de données à partir d'un fichier :**

Pour lire des données à partir d'un fichier, vous pouvez utiliser la fonction numpy.loadtxt. Cette fonction prend en charge différents formats de fichiers tels que CSV, texte délimité par des espaces, etc. Voici un exemple :

Entrée [21]:
```
# Lecture des données à partir d'un fichier CSV
donnees = np.loadtxt('donnees.csv', delimiter=',')
```

La fonction **numpy.loadtxt** est utilisée pour lire les données à partir d'un fichier CSV appelé 'donnees.csv' où les valeurs sont séparées par des virgules. Les données sont stockées dans le tableau **donnees** qui peut être utilisé pour effectuer d'autres opérations avec NumPy.

- **Écriture de données dans un fichier** :

Pour écrire des données dans un fichier, vous pouvez utiliser la fonction **numpy.savetxt**. Cette fonction permet d'écrire les données dans un format spécifié dans un fichier.

Entrée [22]:
```
# Données à écrire dans un fichier
donnees = np.array([[1, 2, 3], [4, 5, 6]])

# Écriture des données dans un fichier CSV
np.savetxt('donnees.csv', donnees, delimiter=',')
```

La fonction **numpy.savetxt** est utilisée pour écrire les données du tableau **donnees** dans un fichier CSV appelé 'donnees.csv'. Les valeurs sont séparées par des virgules en utilisant le paramètre **delimiter=','**. Vous pouvez spécifier d'autres délimiteurs en fonction de vos besoins.

Ces exemples montrent comment utiliser NumPy pour lire et écrire des données à partir de fichiers. NumPy offre également des fonctions avancées pour la lecture et l'écriture de fichiers binaires, la gestion des en-têtes de fichiers, la gestion des valeurs manquantes, etc. Vous pouvez consulter la documentation officielle de NumPy pour en savoir plus sur ces fonctionnalités.

VI. Exercices pratiques

Exercice : Manipulation de données météorologiques

Supposons que vous disposez des données météorologiques sous forme de tableaux NumPy. Les données sont stockées dans les tableaux suivants :

jours = np.array(['Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi', 'Samedi', 'Dimanche'])

temperatures_max = np.array([25, 28, 30, 26, 27, 29, 24])

temperatures_min = np.array([15, 18, 20, 16, 17, 19, 14])

Votre objectif est de réaliser les opérations suivantes :

1. Trouver le jour le plus chaud et le jour le plus froid en termes de température maximale. Afficher le jour correspondant ainsi que la température maximale.
2. Calculer la différence entre la température maximale et la température minimale de chaque jour. Stocker les résultats dans un nouveau tableau appelé "ecarts_temperatures".
3. Filtrer les jours où la température maximale dépasse 27 degrés Celsius en utilisant un masque booléen. Afficher les jours correspondants.
4. Calculer la moyenne des températures maximales et minimales et les afficher.

Solution de l'exercice

Entrée [1]:
```
import numpy as np
```

Entrée [5]:
```
# Données météorologiques
jours = np.array(['Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi', 'Samedi', 'Dimanche'])
temperatures_max = np.array([25, 28, 30, 26, 27, 29, 24])
temperatures_min = np.array([15, 18, 20, 16, 17, 19, 14])
```

Entrée [6]:
```
# Jour le plus chaud
jour_plus_chaud = jours[np.argmax(temperatures_max)]
temperature_max_plus_chaud = np.max(temperatures_max)
print("Jour le plus chaud :", jour_plus_chaud)
print("Température maximale :", temperature_max_plus_chaud)
```

```
Jour le plus chaud : Mercredi
Température maximale : 30
```

Entrée [7]:
```
# Jour le plus froid
jour_plus_froid = jours[np.argmin(temperatures_max)]
temperature_max_plus_froid = np.min(temperatures_max)
print("Jour le plus froid :", jour_plus_froid)
print("Température maximale :", temperature_max_plus_froid)
```

```
Jour le plus froid : Dimanche
Température maximale : 24
```

Entrée [8]:
```
# Différence entre température maximale et minimale
ecarts_temperatures = temperatures_max - temperatures_min
print("Écarts de températures :", ecarts_temperatures)
```

```
Écarts de températures : [10 10 10 10 10 10 10]
```

Entrée [9]:
```
# Jours où la température maximale dépasse 27 degrés Celsius
jours_chauds = jours[temperatures_max > 27]
print("Jours où la température maximale dépasse 27 degrés Celsius :", jours_chauds)
```

```
Jours où la température maximale dépasse 27 degrés Celsius : ['Mardi' 'Mercredi' 'Samedi']
```

Entrée [10]:
```
# Moyenne des températures maximales et minimales
moyenne_temperature_max = np.mean(temperatures_max)
moyenne_temperature_min = np.mean(temperatures_min)
print("Moyenne des températures maximales :", moyenne_temperature_max)
print("Moyenne des températures minimales :", moyenne_temperature_min)
```

```
Moyenne des températures maximales : 27.0
Moyenne des températures minimales : 17.0
```

Références

[1] NumPy Beginner's Guide Third Edition

[2] Apprenez NumPy ebook gratuit non affilié crée à partir des contributeurs de Stack overflow

[3] Zestedesavoir Les bases de NumPy et Matplotlib 21 Mai 2023

[4] https://www.data-transitionnumerique.com/

[5] https://numpy.org/doc/

[6] https://courspython.com/index.html