Chapitre 4
Maîtriser la librairie NumPy

1. Introduction à NumPy

NumPy est la librairie Python dédiée au calcul scientifique fournissant des fonctions très performantes de calcul, mais aussi des structures de données, tout aussi performantes.

En Data Science, il est essentiel d'avoir des structures adaptées pour stocker et manipuler de grandes quantités de données. C'est là qu'intervient NumPy, qui intègre une nouvelle structure de données en Python, les `ndarrays` (tableaux à N dimensions, en français, N représentant un chiffre), qui sont des tableaux multidimensionnels ou matrices. Il est important de noter que cette structure de données permet de stocker des données uniquement de même type (uniquement des nombres entiers par exemple).

Les `ndarrays` sont optimisés pour le stockage de données, mais aussi pour leur manipulation et plus encore pour les calculs, car ceux-ci peuvent être vectorisés. NumPy peut gérer de très gros tableaux et est très performante en temps de calcul sur ces tableaux : les `ndarrays` prennent en effet moins de place mémoire que d'autres objets Python, comme par exemple les listes. C'est pour cela que cette librairie a été développée et qu'elle est si utilisée : elle est très performante. C'est également pour cette raison que de nombreuses librairies ont été développées au-dessus de celle-ci, telle que Pandas, que nous verrons plus tard dans ce livre.

Les `ndarrays` de NumPy peuvent être unidimensionnels (aussi appelés 1D array, ce qu'on peut voir comme une liste), bidimensionnels (2D array) donc un tableau avec des lignes et des colonnes, ou encore des tableaux à plus de deux dimensions (3D array, 4D array, 5D array...), que nous ne verrons pas dans ce livre. Nous travaillerons exclusivement avec les `ndarrays` à deux dimensions dans ce chapitre, qui est la structure de données la plus utilisées en Data Science pour manipuler de grands jeux de données.

Lorsque vous souhaitez effectuer des opérations mathématiques ou logiques rapides sur un grand jeu de données, NumPy est votre allié. Pour pouvoir utiliser NumPy sous Python, il suffit de charger la librairie sous Jupyter, celle-ci étant déjà installée dans la distribution Anaconda.

Syntaxe

```
import numpy as np
```

Remarque

Il est courant d'utiliser l'alias "np" pour la librairie NumPy.

Remarque

Pour la suite de ce chapitre, n'hésitez pas à tester les codes que nous proposerons directement dans le notebook lié à ce chapitre et disponible en téléchargement.

2. Les tableaux NumPy

2.1 Créer un ndarray

2.1.1 Créer un ndarray à partir de listes

On peut créer un tableau NumPy à partir d'une liste en utilisant la fonction `array()` afin de convertir cette liste en tableau.

Syntaxe

```
import numpy as np
notre_tableau=np.array([élément1,élément2,élément3,élément4])
```

Exemple de code

```
import numpy as np
notre_tableau=np.array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
18,19,20])
print(notre_tableau)
type(notre_tableau)
```

Résultat

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
numpy.ndarray
```

Ici, on importe la librairie NumPy, on crée un tableau à partir d'une liste, on affiche ce tableau avec `print()` et on affiche le type de l'objet `notre_tableau`. En résultat, on a notre premier array NumPy, qui est un tableau à une dimension ici, qu'on pourrait considérer comme une liste, mais qui est bien de type `ndarray`.

Pour créer un tableau bidimensionnel, il faut créer une liste contenant des listes, ce qui permet de représenter un tableau à deux entrées : les lignes et les colonnes. Pour cela, il suffit de considérer que la liste qu'on donne à la fonction `array()` est une liste de lignes et que les listes contenues dans cette liste représentent les colonnes. Ainsi, chaque ligne contient une liste qui correspond aux colonnes de cette ligne.

Syntaxe

```
mon_array_bidimensionnel=np.array([[élément ligne 1 colonne 1,
élément ligne 1 colonne 2,élément ligne 1 colonne 3],
[élément ligne 2 colonne 1, élément ligne 2 colonne 2,
élément ligne 2 colonne 3]])
```

Code

```
import numpy as np
mon_array_bidimensionnel=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(mon_array_bidimensionnel)
```

Résultat

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

On remarque tout de suite que, visuellement, cela ressemble à un tableau à deux dimensions, dont la première ligne contient trois colonnes avec les chiffres 1,2,3, la deuxième ligne (la deuxième liste de la liste principale) également trois colonnes avec les chiffres 4, 5, 6, etc.

Il s'agit d'un tableau d'entiers ici, et on ne pourrait pas changer une valeur par un type caractère par exemple, cela générerait une erreur.

Code

```
mon_array_bidimensionnel[0,0]="texte"
```

Résultat

```
ValueError: invalid literal for int() with base 10: 'texte'
```

Ici, nous disons à Python que nous souhaitons modifier la valeur de la première ligne à la première colonne par la chaîne de caractères "texte". Python nous retourne une erreur en nous disant que "texte" n'est pas de type int (entier), donc ça ne fonctionne pas. Comme nous le disions, un ndarray ne peut contenir qu'un seul type de données. Pour connaître le type des données contenues dans un ndarray, il suffit d'utiliser l'attribut dtype.

Syntaxe

```
mon_array.dtype
```

Regardons le type de données de notre tableau d'exemple.

Exemple de code

```
mon_array_bidimensionnel.dtype
```

Résultat

```
dtype('int32')
```

Ici, le type est int32, notre tableau ne peut donc contenir que des nombres entiers, d'où l'erreur précédente.

Remarque

Il existe deux types `int` : `int32` et `int64`. La différence entre `int32` et `int64`, c'est globalement la capacité de stockage du type. `int32` peut stocker des nombres entiers entre -2 147 483 648 et 2 147 483 647 et `int64` peut stocker des nombres entiers entre -9 223 372 036 854 775 808 et +9 223 372 036 854 775 808.

Il est possible de spécifier le type des données du tableau qu'on souhaite créer, avec l'option `dtype` dans la fonction `array()` de NumPy.

Code

```
mon_array_bidimensionnel=np.array([[1.5,2,3],[4,5.2,6],[7,8,9.1]],
dtype = float)
print(mon_array_bidimensionnel)
mon_array_bidimensionnel.dtype
```

Ici, on crée un tableau en spécifiant le type, `float`, puis on affiche le contenu de ce tableau. Enfin, on affiche le type des données contenues dans ce tableau.

Résultat

```
[[1.5 2.  3. ]
 [4.  5.2 6. ]
 [7.  8.  9.1]]
Out[9]:
dtype('float64')
```

Il s'agit bien d'un tableau contenant des données de type `float`, donc des nombres à virgule. Tous les entiers qu'on a donnés lors de la création du tableau (par exemple 2, 3, 4...) sont transformés en réels, `float`, et sont donc suivis de ".0".

2.1.2 Créer un ndarray grâce à des fonctions NumPy

Il arrive que vous sachiez d'avance la taille de votre `ndarray` mais pas encore son contenu, car vous souhaitez remplir ce tableau au fur et à mesure de votre code. NumPy fournit des fonctions permettant de créer et initialiser des `ndarrays` de taille connue et de contenu inconnu.

La première fonction est `np.zeros()`. Cette fonction permet de créer un `ndarray` dont l'ensemble des éléments correspondent à la valeur zéro.

Syntaxe

```
np.zeros((nombre de lignes, nombre de colonnes))
```

Il suffit de donner le nombre de lignes et de colonnes que vous souhaitez voir apparaître dans votre tableau.

Code

```
import numpy as np
mon_array_zeros=np.zeros((4,6))
print(mon_array_zeros)
```

Résultat

```
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]]
```

On obtient un tableau de type `float`, par défaut, rempli de zéros et de dimensions quatre lignes et six colonnes. Si on veut, on peut spécifier que le tableau est de type `int`.

Code

```
import numpy as np
mon_array_zeros=np.zeros((4,6), dtype=int)
print(mon_array_zeros)
```

Résultat

```
[[0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]]
```

Remarque

Il est aussi possible de créer un ndarray à une dimension contenant uniquement des zéros. Pour cela, plutôt que de donner un tuple contenant le nombre de lignes et de colonnes, il suffit de donner un seul nombre à la fonction `np.zeros` : `np.zeros(5)` créera un tableau à une dimension contenant 5 valeurs 0.

Il existe ensuite la fonction `np.ones()`. Cette fonction permet de créer un `ndarray` dont l'ensemble des éléments correspondent au chiffre 1.

Syntaxe

```
np.ones((nombre de lignes, nombre de colonnes))
```

Exemple de code

```
import numpy as np
mon_array_ones=np.ones((4,6), dtype=int)
print(mon_array_ones)
```

Résultat

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

Enfin, il existe la fonction `np.emptys()`. Cette fonction permet de créer un `ndarray` dont les valeurs des éléments sont aléatoires.

Syntaxe

```
np.empty((nombre de lignes, nombre de colonnes))
```

Exemple de code

```
import numpy as np
mon_array_empty=np.empty((4,6), dtype=int)
print(mon_array_empty)
```

Résultat

```
[[-1805877096         611          64           0           0           0]
 [          0           0           0           0   929446194   912471142]
 [ 1667510839   909654373   909193776  1664104498  1630941233   929260599]
 [  959657314   878982705  1650615603   929457204  1697985846   842152292]]
```