

Maîtrisez l'analyse des données avec NumPy Python



Vous souhaitez devenir **Data Analyst** ou Data Scientist ? Vous voulez faire des analyses de données avec python ? Ou vous souhaitez utiliser Python pour trouver des solutions à quelques problèmes du monde réel ? Alors, sachez que NumPy de Python est l'une des bibliothèques que vous devez connaître et utiliser si vous voulez faire de la science des données.

La **programmation informatique** fait intégralement partie du monde du Big Data. En effet, plusieurs aspects de la Data Science nécessitent du développement afin d'obtenir la solution parfaite adaptée aux besoins d'une entreprise. Scala et **Python** font partie des langages les plus utilisés dans ce domaine.

Python met à disposition plusieurs bibliothèques pour la manipulation de données telle que **Panda**, SciPy et **Scikit-Learn**. Toutes ces bibliothèques s'appuient sur NumPy. Donc vous voyez que NumPy est incontournable pour un Data scientist.

Dans ce tutoriel exhaustif, nous allons vous apprendre à maîtriser l'analyse des données avec NumPy Python.

Qu'est-ce que NumPy Python ?

NumPy (*Numerical Python*) est une bibliothèque de python qui comporte des fonctions permettant de manipuler des matrices ou tableaux multidimensionnels.

NumPy est la base de **SciPy**, qui n'est rien d'autre qu'un ensemble de bibliothèques Python pour des calculs scientifiques. Il est beaucoup plus adapté pour les problématiques qui requièrent l'usage des matrices ou des tableaux multidimensionnels, comme la Data Science, l'ingénierie, les mathématiques ou encore les simulations.

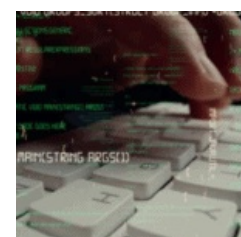
Rechercher sur le site



Catégories d'articles

- Actualité & News (2)
- Base de données (15)
- Big Data Streaming (4)
- Carrière dans le Big Data (25)
- Data Science (5)
- DataOps/DevOps (13)
- Programmation informatique (26)
- Projet Big Data (38)
- Tutoriels Big Data (19)

Les articles les plus consultés



Programmation
informatique
For-

**each/forEach() en Java :
comment utiliser les**

C'est en 2005 que NumPy fut créé par Travis Oliphant. C'est un logiciel open source et compte de nombreux contributeur. NumPy est un projet parrainé financièrement par NumFOCUS.

Pourquoi utiliser NumPy ?

Lors des calculs logiques et mathématiques sur des matrices et tableaux, c'est NumPy qui est très sollicité. Il permet d'effectuer rapidement et efficacement les opérations par rapport aux listes Python.

Les tableaux NumPy utilisent d'abord moins de mémoire et d'espace de stockage, ce qui le rend plus avantageux que les tableaux traditionnels de python.

En effet, un tableau NumPy est de petite taille et ne dépasse pas les 4MB. Mais une liste peut atteindre les 20MB. De plus, les tableaux NumPy sont faciles à manipuler.

Différence entre tableau NumPy et liste Python

Tout le long de cet article, nous allons voir des codes dans le langage Python, vu que NumPy est l'une de ses bibliothèques. Donc, il serait plus pratique que vous ayez les bases nécessaires afin de pouvoir tout suivre. Notre article sur la [programmation Python](#) vous aidera à acquérir cette base.

La question à se poser est : ***pourquoi ne pas utiliser les listes Python qui agissent comme de tableau au lieu d'utiliser les tableaux NumPy ?*** C'est la manière dont Python stocke un objet dans la mémoire qui répondra au mieux à cette question.

Un objet Python n'est rien d'autre qu'un pointeur, il pointe vers l'emplacement mémoire où vous pouvez trouver toutes les informations sur l'objet comme les octets et la valeur.

Les listes Python sont des tableaux pointeur, qui pointent chacun vers un emplacement qui contient les détails relatifs à l'élément. Cela augmente la surcharge de mémoire et parfois, ces informations se répètent lorsque les objets stockés sont de même type.

C'est pour éviter ce problème qu'il faut utiliser les tableaux NumPy. Ceux-ci prennent des éléments homogènes (des éléments qui ont les mêmes types de données), ce qui facilite la manipulation du tableau.

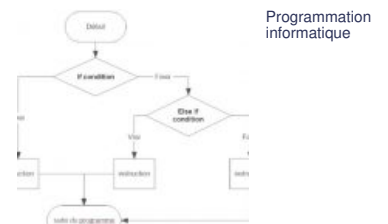
Cette différence est remarquable quand vous voulez traiter avec des tableaux ayant des milliers d'éléments. De plus, avec les tableaux NumPy, vous serez en mesure d'effectuer des opérations par éléments, ce qui est impossible avec les listes Python.

Voilà la vraie raison pour laquelle les tableaux NumPy sont les plus utilisés par rapport aux listes, pendant les opérations d'une grande quantité de données.

Différence entre NumPy et Pandas

boucles améliorées ?

Dans toutes applications, surtout en Big Data,
on sera toujours amené à eff...



Switch/Case : gérer les expressions conditionnelles en Java

En programmation informatique, les structures conditionnelles sont fondamen...



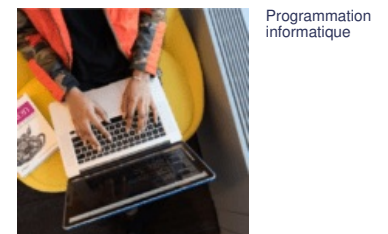
Scraping : Comment collecter les données du web avec Python ?

Vous travaillez avec Python ? Vus vous intéressez au Big Data et vous souha...



Maîtrisez la manipulation des dates en SQL

Il est essentiel de connaître la date et l'heure
surtout quand vous traitez...



Maîtrisez l'analyse des données avec Pandas Python

Vous souhaitez mener des analyses de données avec Python ? Panda Python vous...

Panda est sous licence BSD et open source. Il est sollicité lorsqu'il faut travailler sur des données tabulaires. Ce langage code dans des langues comme Python, Cython et C. Avec Pandas, les données sont au format JSON, SQL etc. Il propose des objets de table 2D qui sont appelés **DataFrame**, l'indexation de la série Pandas est lente et consomme plus mémoire.

Voyons cela avec cet exemple :

```
import pandas as pd
age = [['Aman', 95.5, "Male"], ['Sunny', 65.7, "Female"],
       ['Monty', 85.1, "Male"], ['toni', 75.4, "Male"]]

df = pd.DataFrame(age, columns=['Name', 'Marks', 'Gender'])
df
```

```
Entrée [3]: import pandas as pd

Entrée [4]: age = [['Aman', 95.5, "Male"], ['Sunny', 65.7, "Female"],
                  ['Monty', 85.1, "Male"], ['toni', 75.4, "Male"]]

            df = pd.DataFrame(age, columns=['Name', 'Marks', 'Gender'])
            df

Out[4]:
   Name Marks Gender
0  Aman   95.5   Male
1 Sunny   65.7  Female
2 Monty   85.1   Male
3  toni   75.4   Male

Entrée [ ]:
```

Quant à NumPy, c'est une bibliothèque Python utilisée pour travailler sur des données numériques. NumPy permet de créer et de manipuler des tableaux multidimensionnels. Les tableaux NumPy sont rapides, simples à comprendre et les utilisateurs peuvent facilement effectuer des calculs sur ses tableaux. Il occupe peu de mémoire et son indexation des tableaux est très rapide.

Prenons également un exemple pour illustrer cela :

```
import numpy as np
org_array = np.array([[23, 46, 85],
                      [43, 56, 99],
                      [11, 34, 55]])

print(org_array)
```

```
Entrée [1]: import numpy as np

Entrée [2]: org_array = np.array([[23, 46, 85],
                                  [43, 56, 99],
                                  [11, 34, 55]])

            print(org_array)

[[23 46 85]
 [43 56 99]
 [11 34 55]]

Entrée [ ]:
```

Démarrer avec NumPy Python

Installation de NumPy

Si vous avez installé Anaconda, alors NumPy est déjà installé sur votre système. Mais dans le cas contraire, tapez cette ligne de commande dans votre terminal :

```
pip install numpy
```

Pour utiliser la bibliothèque, il faut l'importer :

```
import numpy as np
```

np est juste une abréviation de numpy utiliser par les data scientists.

Manipulation de tableau NumPy

Comment créer un tableau de base NumPy ?

Il est très facile de créer un tableau NumPy, il faut juste utiliser la fonction **np.array()** pour cela :

```
np.array([ 1 , 2 , 3 , 4 ])
```

Vous aurez comme sortie :

```
Entrée [6]: import numpy as np
Entrée [9]: np.array([ 1 , 2 , 3 , 4 ])
Out[9]: array([1, 2, 3, 4])
Entrée [ ]:
```

Ce tableau n'a que des valeurs entières. Vous avez la possibilité de préciser le type de données dans l'argument :

```
np.array ([ 1 , 2 , 3 , 4 ], dtype = np . float32 )
```

Vous aurez comme sortie :

```
Entrée [6]: import numpy as np
Entrée [8]: np.array ([ 1 , 2 , 3 , 4 ], dtype = np . float32 )
Out[8]: array([1., 2., 3., 4.], dtype=float32)
Entrée [ ]:
```

Mais comme dans les tableaux NumPy, les données sont toutes de même type, alors, NumPy va automatiquement trans-typer les types qui ne correspondent pas.

```
np.array([ 1 , 2.0 , 3 , 4 ])
```

Ici, ce tableau est constitué des valeurs entières (1,3,4) et décimales (2.0). À la sortie, toutes les valeurs entières passeront à une valeur flottante pour que tout soit homogène.

```
Entrée [3]: import numpy as np
Entrée [5]: np.array([1,2.0,3,4])
Out[5]: array([1., 2., 3., 4.])
```

Les tableaux multidimensionnels NumPy

Tableau de Zéros et de Un

Avec NumPy, vous avez la possibilité de créer des tableaux dont toutes les valeurs sont 0. C'est avec la méthode **np.zeros()** que vous pouvez effectuer cette opération.

```
Entrée [6]: import numpy as np
Entrée [7]: np.zeros(5)
Out[7]: array([0., 0., 0., 0., 0.])
Entrée [ ]:
```

Pour créer le tableau de tous les 1, il faut faire :

```
np.ones(5, dtype = np . int32 )
```

```
Entrée [6]: import numpy as np
Entrée [10]: np.ones(5, dtype = np . int32 )
Out[10]: array([1, 1, 1, 1, 1])
Entrée [ ]:
```

Les nombres aléatoires

La méthode la plus utilisée pour créer des tableaux est la méthode **np.random.rand()**. Cette méthode crée des tableaux avec des valeurs aléatoires comprises entre 0 et 1 :

```
Entrée [6]: import numpy as np
Entrée [11]: np.random.rand(2,3)
Out[11]: array([[0.39047165, 0.71153639, 0.14080608],
                [0.43452607, 0.76776219, 0.71496644]])
Entrée [ ]:
```

Un tableau de votre choix

Vous avez la possibilité de remplir le tableau avec n'importe quelle donnée grâce à la méthode **np.full()** :

```
Entrée [6]: import numpy as np
Entrée [12]: np.full((2, 2), 7)
Out[12]: array([[7, 7],
                [7, 7]])
Entrée [ ]:
```

Imatrix dans NumPy

Il est possible de créer une matrice identité grâce à la méthode **np.eye()**. Une matrice identité est une matrice carrée qui a que des 1 sur sa diagonale et 0 partout ailleurs, comme ceci :

```
Entrée [6]: import numpy as np
Entrée [13]: np.eye(4)
Out[13]: array([[1., 0., 0., 0.],
                [0., 1., 0., 0.],
                [0., 0., 1., 0.],
                [0., 0., 0., 1.]])
Entrée [ ]:
```

Tableaux régulièrement espacés

Les tableaux régulièrement espacés s'obtiennent grâce à la méthode **np.arange()**. Le but est d'obtenir un tableau avec des valeurs espacées d'un pas régulier. Il faut savoir que l'intervalle est comme ceci : **[début, fin]** ou le dernier nombre ne sera pas pris.

```
Entrée [16]: import numpy as np
Entrée [18]: np.arange(5)
Out[18]: array([0, 1, 2, 3, 4])
Entrée [ ]:
```

Si l'on exécute *np.arange(2, 12, 3)*, le tableau aura que des valeurs comprises entre 2 et 12 exclu avec un pas régulier de 3.

```
Entrée [16]: import numpy as np
Entrée [19]: np.arange(2,12,3)
Out[19]: array([ 2,  5,  8, 11])
Entrée [ ]:
```

La fonction **np.linspace()** fait aussi presque la même chose, mais au lieu de la taille du pas, elle prend en compte le nombre d'échantillons qui doivent être extraits de l'intervalle. Un point à noter ici est que le dernier nombre est inclus dans les valeurs renvoyées, contrairement au cas de **np.arange()**.

```
Entrée [16]: import numpy as np
Entrée [20]: np.linspace(0, 1, 5)
Out[20]: array([0. , 0.25, 0.5 , 0.75, 1. ])
Entrée [ ]:
```

Maintenant, vous savez comment créer un tableau à l'aide de la bibliothèque NumPy mais il est aussi important d'avoir une idée sur la forme du réseau.

La forme et le remodelage des tableaux NumPy

Après avoir créé votre tableau, la tâche suivante est de vérifier le nombre d'axes, la taille et la forme du tableau.

Dimension des tableaux

Pour déterminer le nombre d'axes ou la dimension d'un tableau NumPy, on passe par l'attribut **ndims** comme suit :

```
# nombre d'axe
a = np.array([[5,10,15],[20,25,20]])
print('Array :','\n',a)
print('Dimensions :','\n',a.ndim)
```

Le résultat que l'on obtient à partir de ce bout de code est le suivant :

```
Entrée [2]: import numpy as np

Entrée [6]: # nombre d'axe
a=np.array([[5,10,15],[20,25,20]])
print('Array :', '\n', a)
print('Dimensions :', a.ndim)

Array :
[[ 5 10 15]
 [20 25 20]]
Dimensions : 2

Entrée [ ]:
```

Vous remarquez que le tableau a 2 dimensions et 3 colonnes.

Forme du tableau NumPy

La forme du tableau NumPy n'est rien d'autre que le nombre de lignes d'éléments sur le long de chaque dimension. Il est possible d'indexer chaque dimension pour obtenir le nombre d'éléments.

```
a = np . array ([[ 1 , 2 , 3 ],[ 4 , 5 , 6 ]])
print ( 'Array :' , '\n' , a )
print ( 'Forme :' , '\n' , a . shape )
print ( 'Ligne = ' , a . shape [ 0 ])
print ( 'Clonne = ' , a . shape [ 1])
```

Le résultat sera :

```
Entrée [16]: import numpy as np

Entrée [22]: a=np.array([[ 1 , 2 , 3 ],[ 4 , 5 , 6 ]])
print ( 'Array :' , '\n' , a )
print('Forme :',a.shape )
print('Ligne=',a.shape [ 0 ])
print('Clonne=',a.shape [ 1])

Array :
[[1 2 3]
 [4 5 6]]
Forme : (2, 3)
Ligne= 2
Clonne= 3

Entrée [ ]:
```

Taille du tableau Numpy

La taille du tableau est la multiplication du nombre de lignes par le nombre de colonnes. Et c'est l'attribut **size** qui permet de faire cette opération :

```
# taille du tableau
a = np . array ([[ 5 , 10 , 15 ],[ 20 , 25 , 20 ]])
print('Taille du tableau :',a.size)
print('Détermination manuelle de la taille du tableau :',a.shape[0]*a.shape[1])
```

Voici ce que l'on obtient :

```
Entrée [16]: import numpy as np

Entrée [26]: # taille du tableau
a = np . array ([[ 5 , 10 , 15 ],[ 20 , 25 , 20 ]])
print('Taille du tableau :',a.size)
print('Détermination manuelle de la taille du tableau :',a.shape[0]*a.shape[1])

Taille du tableau : 6
Détermination manuelle de la taille du tableau : 6

Entrée [ ]:
```

Comment changer la forme du tableau NumPy ?

Vous pouvez donner une autre forme à votre tableau sans modifier les données qu'il comporte et cela grâce à la méthode **np.reshape()**.

```
# remodeler
a=np.array([ 3 , 6 , 9 , 12 ])
print ( 'Forme initiale : ' , '\n' , a )
np.reshape(a ,( 2 , 2 ))
```

```
Entrée [32]: import numpy as np

Entrée [36]: # remodeler
a=np.array([ 3 , 6 , 9 , 12 ])
print ( 'Forme initiale : ' , '\n' , a )
np.reshape(a ,( 2 , 2 ))

Forme initiale :
[ 3 6 9 12]

Out[36]: array([[ 3,  6],
               [ 9, 12]])

Entrée [ ]:
```

Comme vous pouvez le constater, la forme initiale était 1 ligne et 3 colonnes et elle est passée de 2 lignes 2 colonnes.

Quand vous voulez changer la forme et que vous n'êtes pas sûr de l'un des axes, mettez simplement -1. NumPy va se charger du reste dès qu'il voit -1 :

```
a = np . array ([ 3 , 6 , 9 , 12 , 18 , 24 ])
print ( 'Trois lignes : ' , '\n' , np . reshape ( a ,( 3 , - 1 )))
print ( 'Trois colonnes : ' , '\n' , np . reshape ( a ,( - 1 , 3 )))
```

```
Entrée [37]: import numpy as np

Entrée [39]: a = np . array ([ 3 , 6 , 9 , 12 , 18 , 24 ])
print ( 'Trois lignes : ' , '\n' , np . reshape ( a ,( 3 , - 1 )))
print ( 'Trois colonnes : ' , '\n' , np . reshape ( a ,( - 1 , 3 )))

Trois lignes :
[[ 3  6]
 [ 9 12]
 [18 24]]
Trois colonnes :
[[ 3  6  9]
 [12 18 24]]

Entrée [ ]:
```

Aplatir un tableau NumPy

Il arrive parfois que vous alliez vouloir réduire un tableau multidimensionnel à un tableau à une seule dimension. Pour cela, vous pouvez utiliser les méthodes **flatten()** et **ravel()**.


```

a = np.ones((2,2))
b = a.flatten()
c = a.ravel()
print('Forme Original :', a.shape)
print('Array :','\n', a)
print('Forme après aplatissement :',b.shape)
print('Array :','\n', b)
print('Forme après ravel :',c.shape)
print('Array :','\n', c)

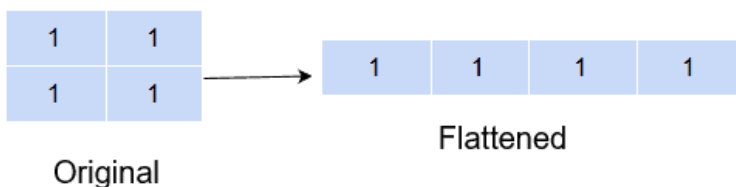
```

```

Entrée [3]: a = np.ones((2,2))
b = a.flatten()
c = a.ravel()
print('Forme Original :', a.shape)
print('Array :','\n', a)
print('Forme après aplatissement :',b.shape)
print('Array :','\n', b)
print('Forme après ravel :',c.shape)
print('Array :','\n', c)

Forme Original : (2, 2)
Array :
[[1. 1.]
 [1. 1.]]
Forme après aplatissement : (4,)
Array :
[1. 1. 1. 1.]
Forme après ravel : (4,)
Array :
[1. 1. 1. 1.]

```



Il existe une grande différence entre **flatten()** et **ravel()**. Avec **flatten()**, on obtient une copie du tableau original alors qu'avec **ravel()**, on obtient plutôt une référence du tableau original. Cela signifie que toute modification apportée au tableau renvoyé par **ravel()** sera également reflétée dans le tableau d'origine alors que ce ne sera pas le cas avec **flatten()**.

```

b[0] = 0
print(a)

```

```

Entrée [4]: b[0] = 0
print(a)

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

Entrée [ ]:

```

Ici, la modification faite n'a pas été reflétée dans le tableau d'origine.

```

c[0] = 0
print(a)

```

```

Entrée [5]: c[0] = 0
print(a)

[[0. 1.]
 [1. 1.]]

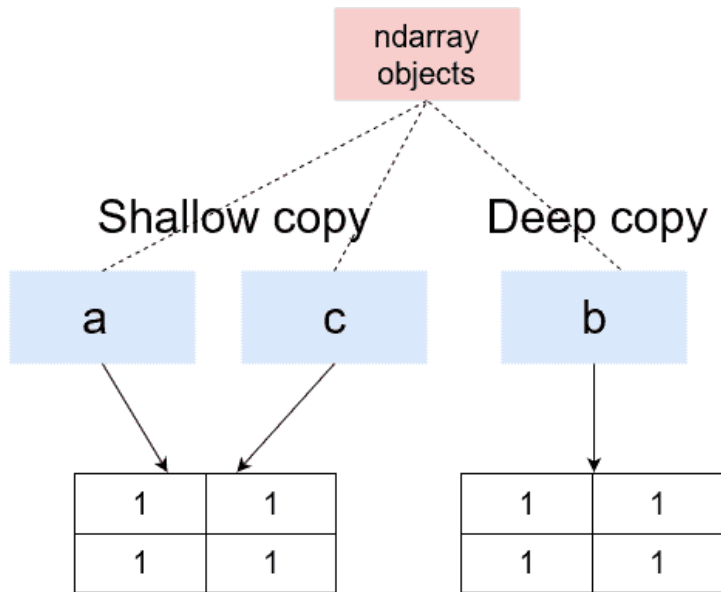
```

Quant à ce dernier, la valeur modifiée est apparue dans le tableau original.

L'action produite ici est que **ravel** crée une copie peu profonde alors que **flatten()** crée une copie profonde.

On parle de copie profonde lorsqu'un nouveau tableau est créé en mémoire et **flatten()** envoie un objet tableau qui pointe vers le nouvel emplacement. Ainsi les modifications apportées ne vont pas toucher l'ancien tableau.

Une copie peu profonde ou superficielle renvoie une référence à la position de la mémoire d'origine. Ce qui veut dire que l'objet renvoyé par **ravel()** pointe vers le même que l'objet tableau d'origine. Donc toute modification du nouveau tableau sera visible sur l'ancien tableau.



Transposition d'un tableau NumPy

Une autre manière de modifier la forme d'un tableau avec NumPy est la méthode **transpose()**. Cette méthode renvoie la transpose d'un tableau qui est le changement des valeurs des lignes par les colonnes.

```
a = np.array([[1,2,3],
[4,5,6]])
b = np.transpose(a)
print('Original','\n','Shape',a.shape,'\n',a)
print('Développer le long des colonnes :','\n','Forme',b.shape,'\n',b)
```

```
Entrée [6]: import numpy as np

Entrée [8]: a = np.array([[1,2,3],
[4,5,6]])
b = np.transpose(a)
print('Original','\n','Shape',a.shape,'\n',a)
print('Développer le long des colonnes :','\n','Forme',b.shape,'\n',b)

Original
Shape (2, 3)
[[1 2 3]
 [4 5 6]]
Développer le long des colonnes :
Forme (3, 2)
[[1 4]
 [2 5]
 [3 6]]

Entrée [ ]: |
```

On passe d'un tableau 2X3 à un tableau 3X2. Cette opération est très fréquente en algèbre linéaire.

Extension et compression d'un tableau NumPy Python

Développer un tableau NumPy

Avec la méthode **expand_dims()**, vous pouvez ajouter un nouvel axe à votre tableau. Il faut juste préciser l'axe :

```
# étendre la dimensions
a = np.array([1,2,3])
b = np.expand_dims(a,axis=0)
c = np.expand_dims(a,axis=1)
print('Original:', '\n', 'Shape', a.shape, '\n', a)
print('Etendre le long des colonnes:', '\n', 'Shape', b.shape, '\n', b)
print('étendre le long des lignes:', '\n', 'Shape', c.shape, '\n', c)
```

```
Entrée [6]: import numpy as np

Entrée [9]: # étendre la dimensions
a = np.array([1,2,3])
b = np.expand_dims(a,axis=0)
c = np.expand_dims(a,axis=1)
print('Original:', '\n', 'Shape', a.shape, '\n', a)
print('Etendre le long des colonnes:', '\n', 'Shape', b.shape, '\n', b)
print('étendre le long des lignes:', '\n', 'Shape', c.shape, '\n', c)

Original:
Shape (3,)
[[1 2 3]]
Etendre le long des colonnes:
Shape (1, 3)
[[[1 2 3]]]
étendre le long des lignes:
Shape (3, 1)
[[1]
 [2]
 [3]]

Entrée [ ]:
```

Compression d'un tableau NumPy

Pour réduire l'axe du tableau, vous pouvez utiliser la méthode **squeeze()**. Cette méthode effacera l'axe qui a une seule entrée. Par exemple, si vous créez une matrice 2X2X1, **squeeze()** effacera la troisième dimension de la matrice.

```
# cOMPRESSER
a = np.array([[[1,2,3],
[4,5,6]]])
b = np.squeeze(a, axis=0)
print('Original', '\n', 'Shape', a.shape, '\n', a)
print('Tableau compressé:', '\n', 'Shape', b.shape, '\n', b)
```

```
Entrée [6]: import numpy as np

Entrée [11]: # cOMPRESSER
a = np.array([[[1,2,3],
[4,5,6]]])
b = np.squeeze(a, axis=0)
print('Original', '\n', 'Shape', a.shape, '\n', a)
print('Tableau compressé:', '\n', 'Shape', b.shape, '\n', b)

Original
Shape (1, 2, 3)
[[[1 2 3]
 [4 5 6]]]
Tableau compressé:
Shape (2, 3)
[[1 2 3]
 [4 5 6]]

Entrée [ ]:
```

Par contre, si vous utilisez **squeeze** sur une matrice 2x2 vous obtiendrez une erreur.

```
Entrée [6]: import numpy as np

Entrée [14]: # COMPRESSER
a = np.array([[1,2,3],
[4,5,6]])
b = np.squeeze(a, axis=0)
print('Original','\n','Shape',a.shape,'\n',a)
print('Tableau compressé:', '\n','Shape',b.shape,'\n',b)

-----
ValueError                                Traceback (most recent call last)
<ipython-input-14-cf6bf7915edf> in <module>
      2 a = np.array([[1,2,3],
      3 [4,5,6]])
----> 4 b = np.squeeze(a, axis=0)
      5 print('Original','\n','Shape',a.shape,'\n',a)
      6 print('Tableau compressé:', '\n','Shape',b.shape,'\n',b)

<__array_function__ internals> in squeeze(*args, **kwargs)

~\anaconda3\lib\site-packages\numpy\core\fromnumeric.py in squeeze(a, axis)
    1493     return squeeze()
    1494     else:
-> 1495         return squeeze(axis=axis)
    1496
    1497
ValueError: cannot select an axis to squeeze out which has size not equal to one
```

Indexation et découpage du tableau NumPy

Jusqu'à présent, vous avez créé un tableau NumPy et puis jouez avec sa forme. Maintenant, vous découvrirez comment extraire une valeur spécifique du tableau grâce au découpage et à l'indexation.

Découpage de tableaux NumPy 1-D

Ici, le découpage veut dire récupérer des éléments d'un index à un autre. Il faut juste donner le point de départ et d'arrivée de l'index comme ceci : **[début : fin]**.

Vous pouvez même faire mieux en donnant le pas de la taille. Qu'est-ce que c'est ? Comme exemple, disons que vous voulez afficher les éléments d'un tableau avec pas 2. On a **[start:end:step-size]** :

```
Entrée [6]: import numpy as np

Entrée [15]: a = np.array([1,2,3,4,5,6])
print(a[1:5:2])

[2 4]

Entrée [ ]:
```

Vous l'aurez remarqué, le dernier élément n'est pas pris en compte, le découpage ne prend pas le dernier élément de l'index.

Pour éviter ce problème, il faut juste prendre une valeur supérieure à la valeur de l'index final :

```
Entrée [6]: import numpy as np

Entrée [17]: a = np.array([1,2,3,4,5,6])
print(a[1:7:2])

[2 4 6]

Entrée [ ]:
```

S'il arrive que vous ne spécifiez pas l'index de début ou de fin, la valeur par défaut est 0 et 1 pour le pas.

```
Entrée [6]: import numpy as np

Entrée [18]: a = np.array([1,2,3,4,5,6])
              print(a[0:2])
              print(a[1:2])
              print(a[1:6])

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

Découpage de tableaux NumPy 2D

Il sera un peu difficile de découper un tableau 2D, car il est composé de lignes et de colonnes. Mais le principe est facile à comprendre et peut être appliqué à n'importe quels autres tableaux de dimension.

Avant de découper un tableau 2D, voyons comment obtenir un élément d'un tableau 2D :

```
Entrée [6]: import numpy as np

Entrée [19]: a = np.array([[1,2,3],
                          [4,5,6]])
              print(a[0,0])
              print(a[1,2])
              print(a[1,0])

              1
              6
              4
```

Dans le cas d'un tableau 2-D, vous allez fournir la valeur de la ligne et de la colonne pour retrouver l'élément à extraire. Mais dans le cas d'un tableau 1-D, c'est juste la valeur de la colonne qui est à préciser, car il n'y a qu'une seule ligne.

Voici comment découper un tableau 2-D :

```
a = np.array([[1,2,3],[4,5,6]])
# affiche les valeurs de la première ligne
print('Valeurs de la première ligne :','\n',a[0:1,:])
# avec un pas pour les colonnes
print('Valeurs alternatives de la première ligne :','\n',a[0:1,:2])
#
print('Valeurs de la deuxième colonne :','\n',a[:,1:2])
print('Valeurs arbitraires :','\n',a[0:1,1:3])
```

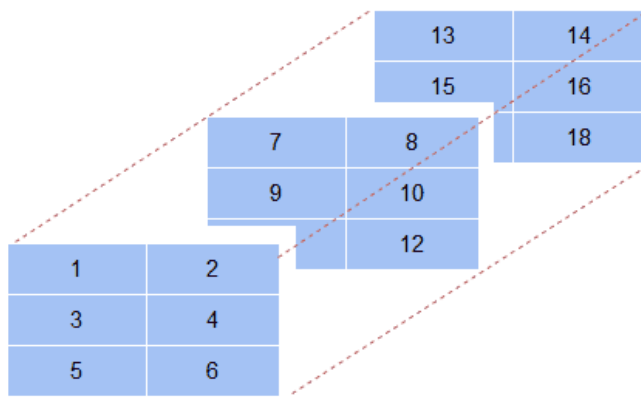
```
Entrée [3]: import numpy as np

Entrée [4]: a = np.array([[1,2,3],[4,5,6]])
              # affiche les valeurs de la première ligne
              print('Valeurs de la première ligne :','\n',a[0:1,:])
              # avec un pas pour les colonnes
              print('Valeurs alternatives de la première ligne :','\n',a[0:1,:2])
              #
              print('Valeurs de la deuxième colonne :','\n',a[:,1:2])
              print('Valeurs arbitraires :','\n',a[0:1,1:3])

              Valeurs de la première ligne :
              [[1 2 3]]
              Valeurs alternatives de la première ligne :
              [[1 3]]
              Valeurs de la deuxième colonne :
              [[2]
              [5]]
              Valeurs arbitraires :
              [[2 3]]
```

Découpage de tableaux NumPy 3D

Jusqu'à ce niveau, on n'a pas encore parlé de tableau 3-D. Voyons à quoi ressemble un tableau 3-D :



```
a = np.array([[[1,2],[3,4],[5,6]],# premier axe tableau
[[7,8],[9,10],[11,12]] ,# tableau deuxième axe
[[13,14],[15,16],[17,18]])# tableau troisième axe
# tableau 3-D
print(a)
```

```
Entrée [3]: import numpy as np

Entrée [5]: a = np.array([[[1,2],[3,4],[5,6]],# premier axe tableau
[[7,8],[9,10],[11,12]] ,# tableau deuxième axe
[[13,14],[15,16],[17,18]])# tableau troisième axe
# tableau 3-D
print(a)

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

 [[ 7  8]
 [ 9 10]
 [11 12]]

 [[13 14]
 [15 16]
 [17 18]]]
```

En plus des lignes et des colonnes, comme le cas de tableau 2D, un tableau 3D a un axe de profondeur où il empile un tableau 2D l'un derrière l'autre.

Donc, en voulant découper un tableau 3D, vous allez d'abord mentionner le tableau 2D que vous découpez. C'est souvent la première valeur de l'index :

```
a = np.array([[[1,2],[3,4],[5,6]],# first axis array
[[7,8],[9,10],[11,12]],# second axis array
[[13,14],[15,16],[17,18]])# third axis array
# valeur
print('Premier tableau, première ligne, valeur de la première colonne :',\n',a[0,0,0])
print('Premier tableau dernière colonne :',\n',a[0,:,1])
print('Deux premières lignes pour les deuxième et troisième tableaux :',\n',a[1:,0:2,0:2])
```

Voici le résultat obtenu :

```
Entrée [3]: import numpy as np

Entrée [4]: a = np.array([[[1,2],[3,4],[5,6]],# first axis array
                        [[7,8],[9,10],[11,12]],# second axis array
                        [[13,14],[15,16],[17,18]])# third axis array
# valeur
print('Premier tableau, première ligne, valeur de la première colonne :', '\n', a[0,0,0])
print('Premier tableau dernière colonne :', '\n', a[0,:,1])
print('Deux premières lignes pour les deuxième et troisième tableaux :', '\n', a[1:,0:2,0:2])

Premier tableau, première ligne, valeur de la première colonne :
1
Premier tableau dernière colonne :
2 4 6
Deux premières lignes pour les deuxième et troisième tableaux :
[[[ 7  8]
  [ 9 10]]

 [[13 14]
  [15 16]]]
```

Si vous souhaitez travailler avec des valeurs sous la forme d'un tableau à une dimension, utiliser la méthode **flatten()** :

```
Entrée [3]: import numpy as np

Entrée [4]: a = np.array([[[1,2],[3,4],[5,6]],# first axis array
                        [[7,8],[9,10],[11,12]],# second axis array
                        [[13,14],[15,16],[17,18]])# third axis array
# valeur
print('Premier tableau, première ligne, valeur de la première colonne :', '\n', a[0,0,0])
print('Premier tableau dernière colonne :', '\n', a[0,:,1])
print('Deux premières lignes pour les deuxième et troisième tableaux :', '\n', a[1:,0:2,0:2])

Premier tableau, première ligne, valeur de la première colonne :
1
Premier tableau dernière colonne :
2 4 6
Deux premières lignes pour les deuxième et troisième tableaux :
[[[ 7  8]
  [ 9 10]]

 [[13 14]
  [15 16]]]
```

Découpage négatif des tableaux NumPy

Encore une autre manière intéressante de découper votre tableau est de passer par le découpage négatif. Voilà comment ça se passe :

```
a = np.array([1,2,3,4,5],
             [6,7,8,9,10])
print(a[:,-1])
```

```
Entrée [9]: import numpy as np

Entrée [12]: a = np.array([1,2,3,4,5],
                        [6,7,8,9,10])
print(a[:,-1])

[ 5 10]
```

Vous avez remarqué que c'est juste les dernières valeurs de chaque ligne qui est affichée. Maintenant, si vous voulez extraire les deux dernières valeurs, voilà ce qu'il faut faire :

```
a = np.array([1,2,3,4,5],
             [6,7,8,9,10])
print(a[:,-1:-3:-1])
```

```
Entrée [9]: import numpy as np

Entrée [13]: a = np.array([1,2,3,4,5],
                        [6,7,8,9,10])
print(a[:,-1:-3:-1])

[[ 5  4]
 [10  9]]
```

C'est la même logique, l'indice de fin ici est -3 et n'est pas pris en compte dans la sortie.

Une utilisation intéressante du découpage négatif est d'inverser complètement le tableau d'origine.

```
a = np.array([[1,2,3,4,5],
[6,7,8,9,10]])
print('Tableau d'origine :','\\n',a)
print('Tableau inversé :','\\n',a[::-1,:-1])
```

```
Entrée [15]: import numpy as np

Entrée [17]: a = np.array([[1,2,3,4,5],
[6,7,8,9,10]])
print('Tableau d'origine :','\\n',a)
print('Tableau inversé :','\\n',a[::-1,:-1])

Tableau d'origine :
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
Tableau inversé :
[[10  9  8  7  6]
 [ 5  4  3  2  1]]

Entrée [ ]:
```

Avec la méthode **flip()**, vous pouvez aussi inverser le tableau :

```
a = np.array([[1,2,3,4,5],
[6,7,8,9,10]])
print('Tableau inversé verticalement :','\\n',np.flip(a,axis=1))
print('Tableau inversé horizontalement :','\\n',np.flip(a,axis=0))
```

```
Entrée [15]: import numpy as np

Entrée [18]: a = np.array([[1,2,3,4,5],
[6,7,8,9,10]])
print('Tableau inversé verticalement :','\\n',np.flip(a,axis=1))
print('Tableau inversé horizontalement :','\\n',np.flip(a,axis=0))

Tableau inversé verticalement :
[[ 5  4  3  2  1]
 [10  9  8  7  6]]
Tableau inversé horizontalement :
[[ 6  7  8  9 10]
 [ 1  2  3  4  5]]

Entrée [ ]:
```

Empiler et concaténer des tableaux NumPy

Empiler des tableaux

Il existe deux manières de créer un tableau en combinant d'autres tableaux. Vous allez découvrir les deux manières :

- Combiner les tableaux sur la verticale, c'est à dire le long des lignes grâce à la méthode **vstack()**. Cela va augmenter évidemment le nombre de lignes dans le tableau final ;
- Effectuer une combinaison sur l'horizontal, c'est à dire le long des colonnes grâce à la méthode **hstack()**. Cela va augmenter évidemment le nombre de colonnes dans le tableau final.

Vetical stacking : np.vstack()

0	1	2	3	4
5	6	7	8	9

Horizontal stacking : np.hstack()

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

```
a = np.arange(0,5)
b = np.arange(5,10)
print('Tableau 1 :','\n',a)
print('Tableau 2 :','\n',b)
print('Empilement vertical :','\n',np.vstack((a,b)))
print('Empilement horizontal :','\n',np.hstack((a,b)))
```

```
Entrée [15]: import numpy as np

Entrée [19]: a = np.arange(0,5)
b = np.arange(5,10)
print('Tableau 1 :','\n',a)
print('Tableau 2 :','\n',b)
print('Empilement vertical :','\n',np.vstack((a,b)))
print('Empilement horizontal :','\n',np.hstack((a,b)))

Tableau 1 :
[0 1 2 3 4]
Tableau 2 :
[5 6 7 8 9]
Empilement vertical :
[[0 1 2 3 4]
 [5 6 7 8 9]]
Empilement horizontal :
[0 1 2 3 4 5 6 7 8 9]
```

Remarque : Vous devez absolument avoir la même taille sur le long de l'axe sur lequel vous voulez combiner les tableaux, dans le cas échéant vous aurez un message d'erreur :

```
Entrée [15]: import numpy as np

Entrée [20]: a = np.arange(0,5)
b = np.arange(5,9)
print('Array 1 :','\n',a)
print('Array 2 :','\n',b)
print('Vertical stacking :','\n',np.vstack((a,b)))
print('Horizontal stacking :','\n',np.hstack((a,b)))

Array 1 :
[0 1 2 3 4]
Array 2 :
[5 6 7 8]

-----
ValueError                                Traceback (most recent call last)
<ipython-input-20-b6a958ba57f9> in <module>
      3 print('Array 1 :','\n',a)
      4 print('Array 2 :','\n',b)
----> 5 print('Vertical stacking :','\n',np.vstack((a,b)))
      6 print('Horizontal stacking :','\n',np.hstack((a,b)))

<__array_function__ internals> in vstack(*args, **kwargs)
~\anaconda3\lib\site-packages\numpy\core\shape_base.py in vstack(tup)
    281     if not isinstance(arrs, list):
    282         arrs = [arrs]
--> 283     return _nx.concatenate(arrs, 0)
    284
    285

<__array_function__ internals> in concatenate(*args, **kwargs)

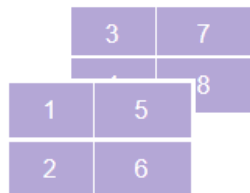
ValueError: all the input array dimensions for the concatenation axis must match exactly, but along dimension 1, the array at index 0 has size 5 and the array at index 1 has size 4
```

Une autre manière de créer de tableau en combinant d'autres et d'utiliser la méthode **dstack()**. Ici, la combinaison se fait index par index et les arrange le long de l'axe de profondeur.

```
Entrée [15]: import numpy as np

Entrée [21]: a = [[1,2],[3,4]]
b = [[5,6],[7,8]]
c = np.dstack((a,b))
print('Tableau 1 :', '\n', a)
print('Tableau 2 :', '\n', b)
print('Dstack :', '\n', c)
print(c.shape)

Tableau 1 :
[[1, 2], [3, 4]]
Tableau 2 :
[[5, 6], [7, 8]]
Dstack :
[[[1 5]
  [2 6]]
 [3 7]
 [4 8]]]
(2, 2, 2)
```



Concaténer des tableaux

Il n'y a pas que la méthode de combinaison qui permet de formuler un tableau à base d'autres tableaux. Avec la méthode **concatenate()**, le tour est joué :

```
a = np.arange(0,5).reshape(1,5)
b = np.arange(5,10).reshape(1,5)
print('Tableau 1 :', '\n', a)
print('Tableau 2 :', '\n', b)
print('Concaténer le long des lignes :', '\n', np.concatenate((a,b),axis=0))
print('Concaténer le long des colonnes :', '\n', np.concatenate((a,b),axis=1))
```

```
Entrée [22]: import numpy as np

Entrée [23]: a = np.arange(0,5).reshape(1,5)
b = np.arange(5,10).reshape(1,5)
print('Tableau 1 :', '\n', a)
print('Tableau 2 :', '\n', b)
print('Concaténer le long des lignes :', '\n', np.concatenate((a,b),axis=0))
print('Concaténer le long des colonnes :', '\n', np.concatenate((a,b),axis=1))

Tableau 1 :
[[0 1 2 3 4]]
Tableau 2 :
[[5 6 7 8 9]]
Concaténer le long des lignes :
[[0 1 2 3 4]
 [5 6 7 8 9]]
Concaténer le long des colonnes :
[[0 1 2 3 4 5 6 7 8 9]]
```

Le défaut dans cette méthode est que le tableau d'origine doit nécessairement avoir l'axe le long duquel vous souhaitez faire la combinaison. Sinon il y a une erreur.

Il y a également la méthode **append()** qui ajoute plutôt de nouvel élément dans le tableau :

```
# ajoute des valeurs à ndarray
a = np.array([[1,2],
              [3,4]])
np.append(a,[[5,6]], axis=0)
```

```
Entrée [22]: import numpy as np

Entrée [25]: # ajoute des valeurs à ndarray
a = np.array([[1,2],
              [3,4]])
np.append(a,[[5,6]], axis=0)

Out[25]: array([[1, 2],
               [3, 4],
               [5, 6]])

Entrée [ ]:
```

Diffusion dans les tableaux NumPy

Python – Une classe à part !

La diffusion fait partie des meilleures fonctionnalités de tableau. Vous pouvez effectuer des opérations arithmétiques entre des tableaux de dimensions différentes ou un nombre simple et un tableau.

```
a = np.arange(10,20,2)
b = np.array([[2],[2]])
print('Ajout de deux tableaux de tailles différentes :','\n',a+b)
print('Multiplication d un tableau et d un nombre :',a*2)
```

```
Entrée [22]: import numpy as np

Entrée [27]: a = np.arange(10,20,2)
b = np.array([[2],[2]])
print('Ajout de deux tableaux de tailles différentes :','\n',a+b)
print('Multiplication d un tableau et d un nombre :',a*2)

Ajout de deux tableaux de tailles différentes :
[[12 14 16 18 20]
 [12 14 16 18 20]]
Multiplication d un tableau et d un nombre : [20 24 28 32 36]

Entrée [ ]:
```

La diffusion modifie la taille du petit tableau pour qu'elle soit égale à celui du grand tableau.

NumPy Ufuncs – Le secret de son succès !

Python est un langage dynamique. Ce qui veut dire que lors d'une affectation, Python n'a pas besoin de connaître le type de donnée de la variable. Il le détermine automatiquement lors de l'exécution et cela ralentit Python malgré que son code soit propre et facile à écrire.

On rencontre ce problème lorsque Python opère plusieurs opérations en boucle, comme l'ajout des deux tableaux. En effet, pour chaque opération effectuée, Python doit déterminer le type de donnée de tous les éléments. Mais ce problème est résolu par NumPy grâce à la fonction **Ufuncs**.

NumPy utilise la vectorisation pour pouvoir accélérer ce travail. 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.

Ufuncs sont des fonctions universelles dans NumPy Python qui sont juste des fonctions mathématiques. Ces fonctions sont automatiquement appelées lorsque vous faites des opérations mathématiques.

Par exemple, en faisant la somme de deux tableaux, la fonction **add()** est

systématiquement appelée en arrière-plan et effectue le travail.

```
a = [1,2,3,4,5]
b = [6,7,8,9,10]
%timeit a+b
```

```
Entrée [31]: import numpy as np

Entrée [35]: a = [1,2,3,4,5]
             b = [6,7,8,9,10]
             %timeit a+b
             1.09 µs ± 491 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
a = np.arange(1,6)
b = np.arange(6,11)
%timeit a+b
```

```
Entrée [37]: import numpy as np

Entrée [39]: a = np.arange(1,6)
             b = np.arange(6,11)
             %timeit a+b
             5.31 µs ± 1.17 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Vous pouvez le constater, la même opération a été effectuée en moins de temps avec NumPy **Unfuncs**.

Mathématique avec les tableaux

NumPy

Voici quelques opérations importantes que vous devez connaître.

Opérations arithmétiques de base sur les tableaux NumPy

Les opérations arithmétiques de base sont faciles à effectuer sur les tableaux NumPy :

```
a = np.arange(1,6)
print('Soustraire : ',a-5)
print('Multiplier : ',a*5)
print('Diviser : ',a/5)
print('Puissance : ',a**2)
print('Reste : ',a%5)
```

```
Entrée [37]: import numpy as np

Entrée [40]: a = np.arange(1,6)
             print('Soustraire : ',a-5)
             print('Multiplier : ',a*5)
             print('Diviser : ',a/5)
             print('Puissance : ',a**2)
             print('Reste : ',a%5)
             Soustraire : [-4 -3 -2 -1  0]
             Multiplier : [ 5 10 15 20 25]
             Diviser : [0.2 0.4 0.6 0.8 1. ]
             Puissance : [ 1  4  9 16 25]
             Reste : [1 2 3 4 0]
```

Moyenne, médiane et écart type

On obtient la moyenne et l'écart type d'un tableau NumPy grâce à la méthode **Mean(), std() et median()** :

```
a = np.arange(5,15,2)
print('Mean :',np.mean(a))
print('Déviation standard :',np.std(a))
print('Median :',np.median(a))
```

```
Entrée [37]: import numpy as np

Entrée [41]: a = np.arange(5,15,2)
              print('Mean :',np.mean(a))
              print('Déviation standard :',np.std(a))
              print('Median :',np.median(a))

              Mean : 9.0
              Déviation standard : 2.8284271247461903
              Median : 9.0

Entrée [ ]:
```

Valeurs Min-Max et leur indice

Les valeurs Min et Max dans un tableau peuvent être trouvés grâce à la méthode **min()** et **max()** :

```
a = np.array([[1,6],
              [4,3]])
# minimum le long d'une colonne
print('Min :',np.min(a,axis=0))
# maximum le long d'une ligne
print('Max :',np.max(a,axis=1))
```

```
Entrée [37]: import numpy as np

Entrée [42]: a = np.array([[1,6],
                          [4,3]])
              # minimum le long d'une colonne
              print('Min :',np.min(a,axis=0))
              # maximum le long d'une ligne
              print('Max :',np.max(a,axis=1))

              Min : [1 3]
              Max : [6 4]

Entrée [ ]:
```

Vous pouvez obtenir facilement l'index de la valeur minimale ou maximale dans le tableau le long d'un axe particulier à l'aide des méthodes **argmin()** et **argmax()**:

```
a = np.array([[1,6,5],
              [4,3,7]])
# minimum le long d'une colonne
print('Min :',np.argmin(a,axis=0))
# maximum le long d'une ligne
print('Max :',np.argmax(a,axis=1))
```

```
Entrée [37]: import numpy as np

Entrée [43]: a = np.array([[1,6,5],
                          [4,3,7]])
              # minimum le long d'une colonne
              print('Min :',np.argmin(a,axis=0))
              # maximum le long d'une ligne
              print('Max :',np.argmax(a,axis=1))

              Min : [0 1 0]
              Max : [1 2]

Entrée [ ]:
```

Tri dans les tableaux NumPy

La complexité d'un algorithme est primordiale pour tout programmeur. Le tri

est une opération basique utilisée au quotidien par les data scientists. Alors, ce sera mieux d'utiliser un bon algorithme de tri avec une complexité temporelle minimale.

La bibliothèque NumPy est une légende quand il s'agit de trier les éléments d'un tableau. Vous pouvez retrouver toute une gamme de fonctions de tri de votre choix :

```
a = np.array([1,4,2,5,3,6,8,7,9])
np.sort(a, kind='quicksort')
```

```
Entrée [37]: import numpy as np

Entrée [44]: a = np.array([1,4,2,5,3,6,8,7,9])
             np.sort(a, kind='quicksort')

Out[44]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
Entrée [ ]:
```

Vous pouvez aussi effectuer le tri le long de l'axe de votre choix :

```
a = np . array ( [[ 5 , 6 , 7 , 4 ],
                  [ 9 , 2 , 3 , 7 ] ] ) # trier le long de la colonne
print ( 'Trier le long de la colonne : ' , '\n' , np . sort ( a , kind =
'mergesort' , axis = 1 ) )
# trier le long la ligne
print ( 'Trier le long de la ligne : ' , '\n' , np . sort ( a , kind = '
mergesort' , axis =0 ) )
```

```
Entrée [37]: import numpy as np

Entrée [46]: a = np . array ( [[ 5 , 6 , 7 , 4 ],
                             [ 9 , 2 , 3 , 7 ] ] ) # trier le long de la colonne
             print ( 'Trier le long de la colonne : ' , '\n' , np . sort ( a , kind = 'mergesort' , axis = 1 ) )
             # trier le long la ligne
             print ( 'Trier le long de la ligne : ' , '\n' , np . sort ( a , kind = 'mergesort' , axis =0 ) )

Trier le long de la colonne :
[[4 5 6 7]
 [2 3 7 9]]
Trier le long de la ligne :
[[5 2 3 4]
 [9 6 7 7]]
```

```
Entrée [ ]:
```

Tableau et Image NumPy Python

Les images ne sont rien d'autre qu'un tableau dont les éléments sont appelés des pixels. Les pixels ont une valeur comprise entre 0 et 255. Donc les tableaux NumPy de Python sont utilisés pour stocker et manipuler des données d'images.

Voici une image qu'on peut manipuler :



L'image sera lue sous forme d'un tableau grâce à la méthode `Image.open()` qui se trouve dans la bibliothèque PIL.

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
# lecture d'image
Image = Image.open( "C:/Users/DJAMELA KELIE/Desktop/image.jpg" )
image_array = np.array( Image )
plt.imshow( image_array )
plt.show()
```

```
Entrée [79]: import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

Entrée [80]: # lecture d'image
Image = Image.open( "C:/Users/DJAMELA KELIE/Desktop/image.jpg" )

Entrée [81]: image_array = np.array( Image )

Entrée [82]: plt.imshow( image_array )
plt.show()
```



```
Entrée [ ]: |
```

Pour voir cette image sous la forme d'un tableau, il faut juste taper :

```
image_array
```

```

Entrée [83]: image_array
Out[83]: array([[115, 106, 67],
               [113, 104, 65],
               [112, 103, 64],
               ...,
               [108, 138, 37],
               [108, 138, 37],
               [108, 138, 37]],

               [[117, 107, 71],
               [115, 105, 69],
               [114, 104, 68],
               ...,
               [157, 135, 34],
               [157, 135, 34],
               [158, 136, 35]],

               [[120, 110, 74],
               [118, 108, 72],
               [117, 107, 71],
               ...,
               [153, 133, 34],
               [153, 133, 34],
               [153, 133, 34]],

               ...,

               [[ 41, 45, 54],
               [ 40, 44, 53],
               [ 39, 43, 52],
               ...,
               [ 31, 33, 45],
               [ 31, 33, 45],
               [ 31, 33, 45]],

               [[ 42, 46, 55],
               [ 41, 45, 54],
               [ 40, 44, 53],
               ...,
               [ 32, 34, 46],
               [ 32, 34, 46],
               [ 32, 34, 46]],

               [[ 46, 50, 59],
               [ 44, 48, 57],
               [ 45, 49, 58],
               ...,
               [ 31, 34, 41],
               [ 31, 34, 41],
               [ 33, 36, 42]]], dtype=uint8)

```

Il est possible de vérifier la forme, le type et la taille du tableau :

```

print( 'classe :', type(image_array) )
print( 'type :', image_array.dtype )
print( 'taille :', image_array.shape )

```

```

Entrée [84]: print( 'classe :', type(image_array) )
              print( 'type :', image_array.dtype )
              print( 'taille :', image_array.shape )

              classe : <class 'numpy.ndarray'>
              type : uint8
              taille : (561, 997, 3)

```

Maintenant qu'on a les données de l'image sous forme de tableau, on peut facilement manipuler l'image à l'aide des fonctions de tableau. Par exemple, vous pouvez retourner l'image horizontalement grâce à la fonction **flip()** :

```


plt.imshow( np.flip(image_array, axis=1) )
plt.show()

```

```

Entrée [86]: plt.imshow( np.flip(image_array, axis=1) )
              plt.show()

```



Voilà, vous avez découvert beaucoup de fonctionnalités dans cet article. Vous connaissez déjà bien l'utilisation des tableaux NumPy et vous êtes prêts à manipuler toutes les données possibles.

Si vous souhaitez découvrir le langage Scala que nous avons mentionné dans l'introduction de cet article, nous vous invitons à télécharger la mini-formation sur ce langage.

Categories:

PROGRAMMATION INFORMATIQUE



Juvénal JVC

Juvénal est spécialisé depuis 2011 dans la valorisation à large échelle des données. Son but est d'aider les professionnels de la data à développer les compétences indispensables pour réussir dans le Big Data. Il travaille actuellement comme Lead Data Engineer auprès des grands comptes. Lorsqu'il n'est pas en voyage, Juvénal rédige des livres ou est en train de préparer la sortie d'un de ses livres. Vous pouvez télécharger un extrait de son dernier livre en date ici : <https://www.data-transitionnumerique.com/extrait-ecosystme-hadoop/>

Related Posts

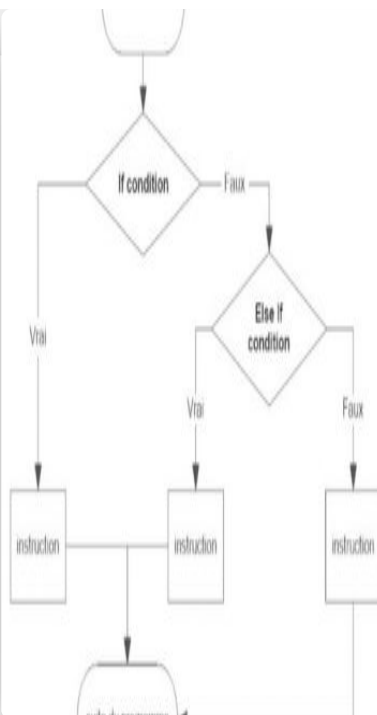


PROGRAMMATION INFORMATIQUE

Programmation en Spring Java : le guide complet

Le monde du Big Data et la programmation informatique sont quasi indissociables. En effet, le traitement des données volumineux requiert un minimum de compétences en programmation. De ce fait, il est

[Read](#)
important de connaître et [more...](#)

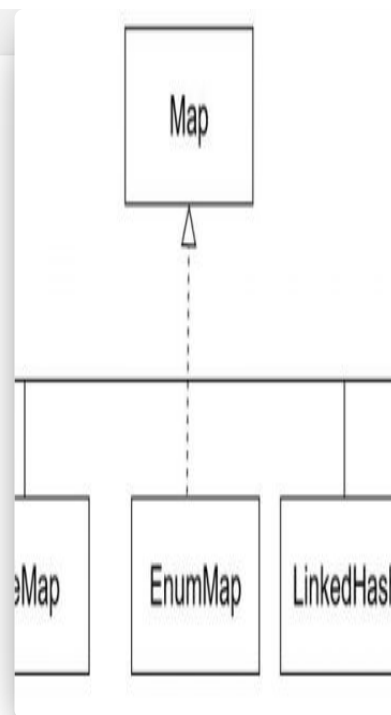


PROGRAMMATION INFORMATIQUE

Switch/Case : gérer les expressions conditionnelles en Java

En programmation informatique, les structures conditionnelles sont fondamentales pour implémenter la logique métier dans vos applications. Ces instructions vous permettent d'indiquer des conditions ou des embranchements d'exécution dans vos programmes informatiques. La plus

[Read](#)
connue d'entre [more...](#)



PROGRAMMATION INFORMATIQUE

Java.util.Map : Maîtrisez les collections et structures de données de Java

Java est à ce jour, le langage le plus utilisé de la planète. Grâce à sa vaste bibliothèque d'API, Il trouve des applications dans de nombreux domaines, tels que le développement mobile, les applications web,

[Read](#)
[more...](#)

A

[REGLEMENT CONCOURS](#)

[MENTIONS LÉGALES](#)

[CGU](#)

[CGV](#)

[PROPOS](#)

[CONTACT](#)

[LEXIQUE](#)