
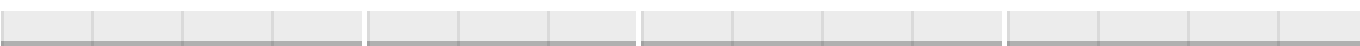


Accueil > Cours > Découvrez les librairies Python pour la Data Science > Plongez en détail dans la librairie NumPy

Découvrez les librairies Python pour la Data Science

 10 heures  Moyenne

Mis à jour le 07/06/2022



Plongez en détail dans la librairie NumPy

08:27

Dans ce chapitre, nous allons voir des techniques pour charger, stocker et manipuler efficacement les données. Elles peuvent venir de sources très variées, mais nous pouvons toujours les considérer comme des tableaux de nombres. Par exemple, une image peut être considérée comme un tableau de deux dimensions (une matrice) où chaque nombre représente l'intensité lumineuse d'un pixel. Pour cette raison, il est fondamental de pouvoir manipuler efficacement ces tableaux. Dans ce chapitre, nous allons voir un outil pour manipuler ces tableaux : Numpy.

NumPy (diminutif de *Numerical Python*) fournit une interface pour stocker et effectuer des opérations sur les données. D'une certaine manière, les tableaux Numpy sont comme les listes en Python, mais Numpy permet de rendre les opérations beaucoup plus efficaces, surtout sur les tableaux de large taille. Les tableaux Numpy sont au cœur de presque tout l'écosystème de data science en Python.

Commençons par importer Numpy.

python

```
1 import numpy as np
```

Créez des tableaux Numpy



Contrairement aux listes en Python, les tableaux Numpy ne peuvent contenir des membres que d'un seul type. Ce type est automatiquement déduit au moment de la création du tableau, et a un impact sur les opérations qui y seront appliquées. On peut aussi spécifier le type manuellement. Nous allons voir des exemples pour les deux cas tout de suite. On peut créer des tableaux de différentes façons dans Numpy.

Depuis une liste Python

python

```
1 # Tableau d'entiers:  
2 np.array([1, 4, 2, 5, 3])
```

Si dans la liste de départ, il y a des données de types différents, Numpy essaiera de les convertir toutes au type le plus général. Par exemple, les entiers (`int`) seront convertis en nombres à virgule flottante (`float`) :

python

```
1 np.array([3.14, 4, 2, 3])
```

```
array([ 3.14, 4. , 2. , 3. ])
```

Nous pouvons aussi manuellement spécifier un type :

python

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



Pour plus d'informations sur les types NumPy, consultez [cette page](#) de la documentation officielle.

Contrairement aux listes Python, les tableaux Numpy peuvent être explicitement multi-dimensionnels. C'est-à-dire que le *tableau multi-dimensionnel* (par exemple un tableau de nombres avec des lignes et des colonnes) est reconnu par NumPy. Mais en Python natif, on représente un tableau multidimensionnel par une liste de listes, car au final, un tableau à 2 entrées (lignes et colonnes), ce n'est rien d'autre qu'une liste de lignes, et une ligne est une liste de nombres !

python

```
1 # Une liste de listes est transformée en un tableau multi-dimensionnel  
2 np.array([range(i, i + 3) for i in [2, 4, 6]])
```



Nous avons utilisé une compréhension de liste pour générer la liste qui sert à créer le tableau Numpy. Rappelez-vous de la syntaxe `[f(x) for x in another_list]` qui crée une liste en appliquant la fonction `f` à chaque membre de `another_list` .

Créer les tableaux directement

Il est souvent plus efficace, surtout pour les tableaux larges, de les créer directement. Numpy contient plusieurs fonctions pour cette tâche.

python

```
1 # Un tableau de longueur 10, rempli d'entiers qui valent 0
2 np.zeros(10, dtype=int)
3
4 # Un tableau de taille 3x5 rempli de nombres à virgule flottante de valeur 1
5 np.ones((3, 5), dtype=float)
6
7 # Un tableau 3x5 rempli de 3,14
8 np.full((3, 5), 3.14)
9
10 # Un tableau rempli d'une séquence linéaire
11 # commençant à 0 et qui se termine à 20, avec un pas de 2
12 np.arange(0, 20, 2)
13
14 # Un tableau de 5 valeurs, espacées uniformément entre 0 et 1
15 np.linspace(0, 1, 5)
16
17 # Celle-ci vous la connaissez déjà! Essayez aussi "randint" et "normal"
18 np.random.random((3, 3))
19
20 # La matrice identité de taille 3x3
21 # (matrice identité : https://fr.wikipedia.org/wiki/Matrice\_identit%C3%A9)
22 np.eye(3)
```

Les propriétés des tableaux



Chaque tableau Numpy a des propriétés qui se révèlent souvent utiles.

python

```
1 np.random.seed(0)
2 x1 = np.random.randint(10, size=6) # Tableau de dimension 1
3 print("nombre de dimensions de x1: ", x1.ndim)
4 print("forme de x1: ", x1.shape)
5 print("taille de x1: ", x1.size)
6 print("type de x1: ", x1.dtype)
```

Essayez ces fonctions sur un tableau multidimensionnel.

Indexation et Slicing



Nous aurons souvent besoin d'accéder à un ou plusieurs éléments contigus d'un tableau. Heureusement, avec Numpy, c'est chose aisée.

Accéder à un seul élément

python

```
1 print(x1)
2
3 # Pour accéder au premier élément
4 print(x1[0])
5
6 # Pour accéder au dernier élément
7 print(x1[-1])
8
9 x2 = np.random.randint(10, size=(3, 4)) # Tableau de dimension 2
10 print(x2[0,1])
11
```

```

12 # On peut aussi modifier les valeurs
13 x1[1] = "1000"
14 print(x1)
15
16 # Attention au type
17 x1[1] = 3.14
18 print(x1)

```

Accéder à plusieurs éléments

De la même façon que nous pouvons indexer des éléments grâce à `[]`, nous pouvons accéder à un ensemble d'éléments en combinant `[]` et `:`. La syntaxe suit une règle simple : `x[début:fin:pas]`.



Le début peut être omis si on veut commencer au début de la liste (c'est à dire si début = 0). La fin peut être omise si on veut aller jusqu'au bout de la liste (c'est à dire fin = -1 ou fin = `len(liste)`). Le pas, ainsi que le dernier `:`, peuvent être omis si le pas est de 1 (-1 si la fin est inférieure au début).

python

```

1 print(x1[:5]) # Les cinq premiers éléments
2
3 print(x1[5:]) # Les éléments à partir de l'index 5
4
5 print(x1[::-2]) # Un élément sur deux

```

Si le pas est négatif, le début et la fin du slice sont inversés. On peut utiliser cette propriété pour inverser un tableau.

python

```

1 x1[::-1]

```

On peut accéder de la même façon aux éléments d'un tableau multi-dimensionnel. Par exemple, on a souvent besoin d'accéder à une ligne ou une colonne d'une matrice.

python

```

1 print(x2)
2
3 x2[0,:] # La première ligne

```

Concaténation

On peut concaténer deux ou plusieurs tableaux.

python

```

1 x = np.array([1, 2, 3])
2 y = np.array([3, 2, 1])
3 np.concatenate([x, y])

```

Si les tableaux sont de dimension > 1, on peut utiliser soit `vstack` (vertical) ou `hstack`.

python

```

1 x = np.array([1, 2, 3])
2 grid = np.array([[9, 8, 7],
3                 ...,
4                 ...,
5                 ...,
6                 [6, 5, 4]])
5 np.vstack([x, grid])

```

Opérations sur les tableaux Numpy



Jusqu'à maintenant dans ce chapitre, nous avons vu des choses très basiques sur les tableaux Numpy. A partir d'ici, nous allons voir ce qui rend Numpy vraiment indispensable.

Les boucles peuvent êtres lentes en Python

L'implémentation de référence de Python, encore appelée CPython, est très flexible, mais cette flexibilité l'empêche d'utiliser toutes les optimisations possibles. Par exemple, observez le temps d'exécution de ce morceau de code.

python

```
1 def calcul_inverse(values):
2     output = np.empty(len(values))
3     for i in range(len(values)):
4         output[i] = 1.0 / values[i]
5     return output
6
7 values = np.random.randint(1, 10, size=5)
8 print(calcul_inverse(values))
9
10 tableau_large = np.random.randint(1, 100, size=1000000)
11
12 # Ceci est une facilité des notebooks jupyter pour
13 # mesurer le temps d'exécution d'une instruction
14 %timeit calcul_inverse(tableau_large)
```

Il faut plusieurs secondes pour accomplir un million d'opérations. Sachant que les processeurs actuels sont capables d'exécuter des **milliards** d'opérations par seconde, cette durée peut apparaître absurde. Ce délai est dû à toutes les opérations annexes que doit accomplir l'interprète, comme les appels de fonction et vérifications de type.

Dans beaucoup de cas, Numpy fournit une interface pour ces opérations qui n'implique que des données du même type. Cette interface utilise une implémentation en langage C, qui peut faire appel à toutes les fonctionnalités des processeurs modernes. Par exemple, on peut calculer les inverses de tous les éléments d'un tableau Numpy comme ceci.

python

```
1 %timeit (1.0 / tableau_large)
```

L'opération a pris presque 1000 fois moins de temps sur ma machine.



À chaque fois que vous vous trouvez en train d'utiliser une boucle pour effectuer une opération en Python, demandez-vous si cette opération ne peut pas s'accomplir grâce à Numpy sans boucle.

Nous allons maintenant faire un tour d'horizon de ces opérations.

Les fonctions universelles

python

```
1 # Il y a tout d'abord des opération mathématiques simples
2 x = np.arange(4)
3 print("x      =", x)
4 print("x + 5 =", x + 5)
5 print("x - 5 =", x - 5)
6 print("x * 2 =", x * 2)
7 print("x / 2 =", x / 2)
8 print("x // 2 =", x // 2) # Division avec arrondid
```

Vous pouvez aussi appeler des fonctions sur les tableaux Numpy, et même sur les listes Python.

python

```
1 x = [-2, -1, 1, 2]
2 print("La valeur absolue: ", np.abs(x))
3 print("Exponentielle: ", np.exp(x))
4 print("Logarithme: ", np.log(np.abs(x)))
```



Il y a plein d'autres fonctions, que je vous invite à découvrir dans la [**documentation de Numpy**](#). La leçon à retenir ici est simple : évitez tant que possible les boucles.

Opérations Booléennes

Vous pouvez aussi exécuter des opérations booléennes sur vos tableaux. En clair, vous pouvez demander si une certaine condition est vraie pour chaque élément d'un tableau. Par exemple :

python

```
1 x = np.random.rand(3,3)
2 x > 0.5
```

Vous pouvez coupler cette capacité avec la fonction `np.where`, pour retourner parmi tous les éléments d'un tableau **les index** de ceux qui vérifient une certaine propriété :

python

```
1 np.where(x > 0.5)
```

Agrégation

Très souvent, face à de larges quantités de données, la première chose à faire est de calculer des statistiques sur nos données, comme la moyenne ou l'écart type. Numpy a des fonctions pour calculer ces données sur ses tableaux.

python

```
1 L = np.random.random(100)
2 np.sum(L)
```

Cette fonction existe aussi en Python, mais la version Numpy est beaucoup plus rapide. De même, Numpy a des équivalents pour `min` et `max`. Faites cependant attention aux arguments optionnels de chaque version de ces fonctions. Gardez aussi à l'esprit que seule la version Numpy gère correctement les tableaux multidimensionnels.

python

```
1 %timeit sum(tableau_large)
2 %timeit np.sum(tableau_large)
```

Les fonctions d'agrégation peuvent optionnellement ne s'appliquer qu'à une dimension d'un tableau multidimensionnel. Par exemple, nous pouvons avoir besoin de la somme des éléments de chaque colonne d'une matrice (nous avons vu un cas d'utilisation dans le chapitre 3 de la première partie de ce cours). Pour cela nous utilisons l'argument optionnel `axis`.

python

```
1 M = np.random.random((3, 4))
2 print(M)
3 # Notez la syntaxe variable.fonction au lieu de
4 # np.fonction(variable). Les deux sont possibles si
```

```
5 # la variable est un tableau Numpy.  
6 print("La somme de tous les éléments de M: ", M.sum())  
7 print("Les sommes des colonnes de M: ", M.sum(axis=0))
```

Parmi les nombreuses fonctions disponibles, notons :

- `np.std` pour calculer l'écart type
- `np.argmax` pour trouver l'index de l'élément minimum
- `np.percentile` pour calculer des statistiques sur les éléments.

Broadcasting



Le broadcasting désigne un ensemble de règles pour appliquer une opération qui normalement ne s'applique que sur une seule valeur à l'ensemble des membres d'un tableau Numpy. Par exemple, pour les tableaux de même taille, les opérations comme l'addition s'appliquent normalement élément par élément.

python

```
1 a = np.array([0, 1, 2])  
2 b = np.array([5, 5, 5])  
3 a + b
```

Le broadcasting nous permet d'appliquer ces opérations sur des tableaux de dimensions différentes. Par exemple :

python

```
1 a + 5
```

C'est comme si Numpy avait converti la valeur 5 en un tableau de taille 3, et ensuite avait additionné ce tableau à celui contenu dans a. Ce n'est qu'une vue d'esprit faite pour comprendre ce qui se passe. Numpy se passe de toutes ces opérations annexes, ce qui rend le tout plus rapide.

Cela marche aussi quand les deux arguments sont des tableaux :

python

```
1 M = np.ones((3, 3))  
2 print("M vaut: \n", M)  
3 print("M+a vaut: \n", M+a)
```

N'hésitez pas à vous essayer à ces opérations avec des tableaux de dimensions et tailles différentes pour bien prendre en main cet outil. Un dernier exemple cependant : que pensez-vous sera le résultat de l'opération suivante ?

python

```
1 a = np.arange(3)  
2 # La ligne suivante crée une matrice de taille 3x1  
3 # avec trois lignes et une colonne.  
4 b = np.arange(3)[: , np.newaxis]  
5 a+b
```

Pour aller plus loin



La librairie Numpy est très avancée et contient énormément de fonctions et de mécanismes. N'hésitez pas à aller sur [le site officiel](#) pour vous renseigner un peu plus sur ces fonctions et des choses que nous n'avons pas évoquées ici, comme les `mask`, qui permettent d'accéder à un sous-ensemble des tableaux Numpy composé d'éléments qui vérifient une certaine propriété (par exemple tous les éléments positifs).

J'ai terminé ce chapitre et je passe au suivant



Entraînez-vous en simulant le problème de
Monty Hall avec Numpy

Maîtrisez les possibilités offertes par
Matplotlib



Le professeur



Ali Neishabouri

Freelance Data Scientist, and teacher at OpenClassrooms

OPENCCLASSROOMS



OPPORTUNITÉS



AIDE



POUR LES ENTREPRISES



EN PLUS



Français



Télécharger dans
l'App Store

