TP1 : Numpy Part 1

1. Introduction à Numpy

Numpy est le paquet le plus basique et puissant pour travailler avec des données en Python. Si vous allez travailler sur des projets d'analyse de données ou d'apprentissage automatique, avoir une compréhension solide de numpy est presque obligatoire. C'est parce que d'autres paquets pour l'analyse de données (comme pandas) sont construits sur numpy, et le paquet scikit-learn, utilisé pour construire des applications d'apprentissage automatique, travaille également largement avec numpy.

Alors, que fournit numpy? Fondamentalement, numpy fournit d'excellents objets ndarray, abréviation de tableaux n-dimensionnels. Dans un objet 'ndarray', alias 'array', vous pouvez stocker plusieurs éléments du même type de données. Ce sont les fonctionnalités autour de l'objet array qui rendent numpy si pratique pour effectuer des manipulations mathématiques et des manipulations de données. Vous pourriez vous demander, 'Je peux stocker des nombres et d'autres objets dans une liste Python elle-même et effectuer toutes sortes de calculs et de manipulations via des compréhensions de liste, des boucles for, etc. Pourquoi ai-je besoin d'un tableau numpy?' Eh bien, il y a des avantages très significatifs à utiliser des tableaux numpy plutôt que des listes. Pour comprendre cela, regardons d'abord comment créer un tableau numpy.

2. Comment créer un tableau numpy ?

Il existe plusieurs façons de créer un tableau numpy, la plupart d'entre elles seront abordées au fur et à mesure de votre lecture. Cependant, l'une des façons les plus courantes est d'en créer un à partir d'une liste ou d'un objet semblable à une liste en le passant à la fonction np.array.

```
# Create an 1d array from a list
import numpy as np
list1 = [0,1,2,3,4]
arr1d = np.array(list1)

# Print the array and its type
print(type(arr1d))
arr1d

#> class 'numpy.ndarray'
#> array([0, 1, 2, 3, 4])
```

La principale différence entre un tableau et une liste est que les tableaux sont conçus pour gérer des opérations vectorisées, tandis qu'une liste Python ne l'est pas. Cela signifie que si vous appliquez une fonction, elle est exécutée sur chaque élément du tableau, plutôt que sur l'ensemble de l'objet tableau.

Supposons que vous souhaitiez ajouter le nombre 2 à chaque élément de la liste. La manière intuitive de le faire est quelque chose comme ceci :

```
list1 + 2  # error
```

Cela n'était pas possible avec une liste. Mais vous pouvez le faire avec un tableau ndarray.

```
# Add 2 to each element of arr1d
arr1d + 2

#> array([2, 3, 4, 5, 6])
```

Une autre caractéristique est que, une fois qu'un tableau numpy est créé, vous ne pouvez pas augmenter sa taille. Pour le faire, vous devrez créer un nouveau tableau. Mais un tel comportement d'extension de la taille est naturel dans une liste.

Cependant, il existe tellement plus d'avantages. Découvrons-les.

Donc, c'est à propos d'un tableau 1D. Vous pouvez également passer une liste de listes pour créer une matrice, comme un tableau 2D.

```
# Create a 2d array from a list of lists
list2 = [[0,1,2], [3,4,5], [6,7,8]]
arr2d = np.array(list2)
arr2d

#> array([[0, 1, 2],
#>        [3, 4, 5],
#>        [6, 7, 8]])
```

Vous pouvez également spécifier le type de données en définissant l'argument dtype. Certains des types de données numpy les plus couramment utilisés sont : 'float', 'int', 'bool', 'str' et 'object'.

Pour contrôler les allocations mémoire, vous pouvez choisir d'utiliser l'un des types suivants : 'float32', 'float64', 'int8', 'int16' ou 'int32'.

```
# Create a float 2d array
arr2d_f = np.array(list2, dtype='float')
arr2d_f
```

```
#> array([[ 0.,  1.,  2.],
#>        [ 3.,  4.,  5.],
#>        [ 6.,  7.,  8.]])
```

Le point décimal après chaque nombre indique le type de données float. Vous pouvez également le convertir en un type de données différent en utilisant la méthode astype.

```
# Convert to 'int' datatype
arr2d_f.astype('int')

#> array([[0, 1, 2],
#>        [3, 4, 5],
#>        [6, 7, 8]])
# Convert to int then to str datatype
arr2d_f.astype('int').astype('str')

#> array([['0', '1', '2'],
#>        ['3', '4', '5'],
#>        ['6', '7', '8']],
#>       dtype='U21')
```

Un tableau numpy doit avoir tous les éléments du même type de données, contrairement aux listes. C'est une autre différence significative.

Cependant, si vous êtes incertain du type de données que votre tableau contiendra, ou si vous souhaitez stocker à la fois des caractères et des nombres dans le même tableau, vous pouvez définir le type de données comme 'object'.

```
# Create a boolean array
arr2d_b = np.array([1, 0, 10], dtype='bool')
arr2d_b

#> array([ True, False,  True], dtype=bool)
# Create an object array to hold numbers as well as strings
arr1d_obj = np.array([1, 'a'], dtype='object')
arr1d_obj

#> array([1, 'a'], dtype=object)
```

Enfin, vous pouvez toujours convertir un tableau en une liste Python en utilisant la méthode tolist().

```
# Convert an array back to a list
arr1d_obj.tolist()

#> [1, 'a']
```

Pour résumer, les principales différences avec les listes Python sont les suivantes :

- Les tableaux prennent en charge les opérations vectorisées, tandis que les listes ne le font pas.
- Une fois qu'un tableau est créé, vous ne pouvez pas changer sa taille. Vous devrez créer un nouveau tableau ou écraser l'existante.
- Chaque tableau a un et un seul type de données. Tous les éléments doivent être de ce type.
- Un tableau numpy équivalent occupe beaucoup moins d'espace qu'une liste Python de listes.

.

3. Comment inspecter la taille et la forme d'un tableau numpy ?

Chaque tableau a certaines propriétés que je souhaite comprendre pour en savoir plus sur le tableau. Considérons le tableau **arr2d**. Étant donné qu'il a été créé à partir d'une liste de listes, il a 2 dimensions qui peuvent être représentées comme des lignes et des colonnes, comme dans une matrice. Si je l'avais créé à partir d'une liste de listes de listes, il aurait eu 3 dimensions, comme dans un cube. Et ainsi de suite.

Supposons que vous ayez reçu un vecteur numpy que vous n'avez pas créé vous-même. Quelles sont les choses que vous voudriez explorer pour en savoir plus sur ce tableau ? Eh bien, je veux savoir :

1. S'il s'agit d'un tableau 1D, 2D ou plus. (utilisez ndim)
2. Combien d'éléments sont présents dans chaque dimension (utilisez shape)
3. Quel est son type de données (utilisez dtype)
4. Quel est le nombre total d'éléments (utilisez size)
5. Des échantillons des premiers éléments du tableau (via l'indexation)

```
# Create a 2d array with 3 rows and 4 columns
list2 = [[1, 2, 3, 4],[3, 4, 5, 6], [5, 6, 7, 8]]
arr2 = np.array(list2, dtype='float')
arr2

#> array([[ 1.,  2.,  3.,  4.],
#>        [ 3.,  4.,  5.,  6.],
#>        [ 5.,  6.,  7.,  8.]])
# shape
print('Shape: ', arr2.shape)

# dtype
```

```
print('Datatype: ', arr2.dtype)

# size
print('Size: ', arr2.size)

# ndim
print('Num Dimensions: ', arr2.ndim)

#> Shape:  (3, 4)
#> Datatype:  float64
#> Size:  12
#> Num Dimensions:  2
```

4. Comment extraire des éléments spécifiques d'un tableau ?

```
arr2

#> array([[ 1.,  2.,  3.,  4.],
#>        [ 3.,  4.,  5.,  6.],
#>        [ 5.,  6.,  7.,  8.]])
```

Vous pouvez extraire des portions spécifiques d'un tableau en utilisant l'indexation à partir de 0, quelque chose de similaire à ce que vous feriez avec les listes Python. Mais contrairement aux listes, les tableaux numpy peuvent accepter en option autant de paramètres entre crochets qu'il y a de dimensions.

```
# Extract the first 2 rows and columns
arr2[:2, :2]
list2[:2, :2]  # error

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

De plus, les tableaux numpy prennent en charge l'indexation booléenne. Un tableau d'index booléen a la même forme que le tableau à filtrer et il ne contient que des valeurs True et False. Les valeurs correspondant aux positions True sont conservées dans le résultat.

```
# Get the boolean output by applying the condition to each element.
b = arr2 > 4
b

#> array([[False, False, False, False],
#>        [False, False,  True,  True],
#>        [ True,  True,  True,  True]], dtype=bool)
arr2[b]
```

```
#> array([ 5.,  6.,  5.,  6.,  7.,  8.])
```

4.1 Comment inverser les lignes et l'ensemble d'un tableau numpy ?

Inverser un tableau fonctionne comme vous le feriez avec des listes, mais vous devez le faire pour tous les axes (dimensions) si vous voulez un renversement complet.

```
# Reverse only the row positions
arr2[::-1, ]

#> array([[ 5.,  6.,  7.,  8.],
#>        [ 3.,  4.,  5.,  6.],
#>        [ 1.,  2.,  3.,  4.]])
# Reverse the row and column positions
arr2[::-1, ::-1]

#> array([[ 8.,  7.,  6.,  5.],
#>        [ 6.,  5.,  4.,  3.],
#>        [ 4.,  3.,  2.,  1.]])
```

4.2 Comment représenter les valeurs manquantes et l'infini ?

Les valeurs manquantes peuvent être représentées à l'aide de l'objet np.nan, tandis que np.inf représente l'infini. Plaçons-en quelques-unes dans arr2d.

```
# Insert a nan and an inf
arr2[1,1] = np.nan  # not a number
arr2[1,2] = np.inf  # infinite
arr2

#> array([[  1.,   2.,   3.,   4.],
#>        [  3.,  nan,  inf,   6.],
#>        [  5.,   6.,   7.,   8.]])
# Replace nan and inf with -1. Don't use arr2 == np.nan
missing_bool = np.isnan(arr2) | np.isinf(arr2)
arr2[missing_bool] = -1
arr2

#> array([[ 1.,  2.,  3.,  4.],
#>        [ 3., -1., -1.,  6.],
#>        [ 5.,  6.,  7.,  8.]])
```

4.3 Comment calculer la moyenne, le minimum et le maximum sur le ndarray ?

Le ndarray dispose des méthodes respectives pour calculer cela pour l'ensemble du tableau.

```
# mean, max and min
```

```
print("Mean value is: ", arr2.mean())
print("Max value is: ", arr2.max())
print("Min value is: ", arr2.min())

#> Mean value is:  3.58333333333
#> Max value is:  8.0
#> Min value is:  -1.0
```

Cependant, si vous souhaitez calculer les valeurs minimales par ligne ou par colonne, utilisez plutôt la version np.amin.

```
# Row wise and column wise min
print("Column wise minimum: ", np.amin(arr2, axis=0))
print("Row wise minimum: ", np.amin(arr2, axis=1))

#> Column wise minimum:  [ 1. -1. -1.  4.]
#> Row wise minimum:  [ 1. -1.  5.]
```

Calculer le minimum par ligne est bien. Mais que faire si vous souhaitez effectuer une autre opération/fonction par ligne ? Cela peut être fait en utilisant np.apply_over_axis, que vous verrez dans le prochain sujet.

```
# Cumulative Sum
np.cumsum(arr2)

#> array([  1.,   3.,   6.,  10.,  13.,  12.,  11.,  17.,  22.,  28.,
35., 43.])
```

5. Comment créer un nouveau tableau à partir d'un tableau existant ?

Si vous attribuez simplement une partie d'un tableau à un autre tableau, le nouveau tableau que vous venez de créer fait en réalité référence au tableau parent en mémoire.

Cela signifie que si vous apportez des modifications au nouveau tableau, elles se refléteront également dans le tableau parent.

Ainsi, pour éviter de perturber le tableau parent, vous devez en faire une copie à l'aide de copy(). Tous les tableaux numpy sont équipés de la méthode copy().

```
# Assign portion of arr2 to arr2a. Doesn't really create a new array.
arr2a = arr2[:2,:2]
arr2a[:1, :1] = 100  # 100 will reflect in arr2
arr2

#> array([[ 100.,    2.,    3.,    4.],
#>        [   3.,   -1.,   -1.,    6.],
```

```
#>        [   5.,    6.,    7.,    8.]])
# Copy portion of arr2 to arr2b
arr2b = arr2[:2, :2].copy()
arr2b[:1, :1] = 101  # 101 will not reflect in arr2
arr2
```

```
#> array([[ 100.,    2.,    3.,    4.],
#>        [   3.,   -1.,   -1.,    6.],
#>        [   5.,    6.,    7.,    8.]])
```

6. Remodelage et aplatissement des tableaux multidimensionnels

Le remodelage consiste à changer l'agencement des éléments de telle sorte que la forme du tableau change tout en maintenant le même nombre de dimensions.

En revanche, l'aplatissement convertira un tableau multidimensionnel en un tableau plat à une dimension (1D), mais pas à une autre forme.

Tout d'abord, remanions le tableau arr2 d'une forme 3×4 à une forme 4×3.

```
# Reshape a 3x4 array to 4x3 array
arr2.reshape(4, 3)
```

```
#> array([[ 100.,    2.,    3.],
#>        [   4.,    3.,   -1.],
#>        [  -1.,    6.,    5.],
#>        [   6.,    7.,    8.]])
```

6.1 Quelle est la différence entre `flatten()` et `ravel()` ?

Il existe deux méthodes populaires pour mettre à plat (flatten) un tableau, en utilisant la méthode `flatten()` et l'autre en utilisant la méthode `ravel()`.

La différence entre `ravel` et `flatten` est que le nouveau tableau créé à l'aide de `ravel` est en réalité une référence au tableau parent. Ainsi, toute modification apportée au nouveau tableau affectera également le tableau parent. Cependant, cela est efficace en termes de mémoire car il ne crée pas de copie.

```
# Flatten it to a 1d array
arr2.flatten()
```

```
#> array([ 100.,    2.,    3.,    4.,    3.,   -1.,   -1.,    6.,
5., 6.,    7.,    8.])
# Changing the flattened array does not change parent
b1 = arr2.flatten()
b1[0] = 100  # changing b1 does not affect arr2
arr2
```

```
#> array([[ 100.,    2.,    3.,    4.],
#>        [   3.,   -1.,   -1.,    6.],
#>        [   5.,    6.,    7.,    8.]])
# Changing the raveled array changes the parent also.
b2 = arr2.ravel()
b2[0] = 101  # changing b2 changes arr2 also
arr2

#> array([[ 101.,    2.,    3.,    4.],
#>        [   3.,   -1.,   -1.,    6.],
#>        [   5.,    6.,    7.,    8.]])
```

7. Comment créer des séquences, des répétitions et des nombres aléatoires avec numpy ?

La fonction np.arange est pratique pour créer des séquences de nombres personnalisées en tant que ndarray.

```
# Lower limit is 0 be default
print(np.arange(5))

# 0 to 9
print(np.arange(0, 10))

# 0 to 9 with step of 2
print(np.arange(0, 10, 2))

# 10 to 1, decreasing order
print(np.arange(10, 0, -1))

#> [0 1 2 3 4]
#> [0 1 2 3 4 5 6 7 8 9]
#> [0 2 4 6 8]
#> [10  9  8  7  6  5  4  3  2  1]
```

Vous pouvez définir les positions de début et de fin en utilisant np.arange. Mais si vous vous concentrez sur le nombre d'éléments dans le tableau, vous devrez calculer manuellement la valeur du pas approprié.

Disons que vous voulez créer un tableau exactement avec 10 nombres entre 1 et 50. Pouvez-vous calculer quelle serait la valeur du pas ?

Eh bien, je vais utiliser np.linspace à la place.

```
# Start at 1 and end at 50
np.linspace(start=1, stop=50, num=10, dtype=int)
```

```
#> array([ 1,  6, 11, 17, 22, 28, 33, 39, 44, 50])
```

Remarquez que, comme j'ai explicitement forcé le type de données à être int, les nombres ne sont pas également espacés en raison de l'arrondi.

Similaire à np.linspace, il y a aussi np.logspace qui augmente sur une échelle logarithmique. Dans np.logspace, la valeur de départ donnée est en réalité base^start et se termine par base^stop, avec une valeur de base par défaut de 10.

```
# Limit the number of digits after the decimal to 2
np.set_printoptions(precision=2)

# Start at 10^1 and end at 10^50
np.logspace(start=1, stop=50, num=10, base=10)

#> array([  1.00e+01,   2.78e+06,   7.74e+11,   2.15e+17,
5.99e+22,
#>          1.67e+28,   4.64e+33,   1.29e+39,   3.59e+44,
1.00e+50])
```

Les fonctions np.zeros et np.ones vous permettent de créer des tableaux de la forme souhaitée où tous les éléments sont soit des 0 ou des 1.

```
np.zeros([2,2])
#> array([[ 0.,  0.],
#>        [ 0.,  0.]])
np.ones([2,2])
#> array([[ 1.,  1.],
#>        [ 1.,  1.]])
```

7.1 Comment créer des séquences répétitives ?

`np.tile` répète toute une liste ou un tableau n fois. En revanche, **`np.repeat`** répète chaque élément n fois.

```
a = [1,2,3]

# Repeat whole of 'a' two times
print('Tile:   ', np.tile(a, 2))

# Repeat each element of 'a' two times
print('Repeat: ', np.repeat(a, 2))

#> Tile:    [1 2 3 1 2 3]
#> Repeat:  [1 1 2 2 3 3]
```

7.2 Comment générer des nombres aléatoires ?

Le module random propose des fonctions pratiques pour générer des nombres aléatoires (et également des distributions statistiques) de toute forme donnée.

```
# Random numbers between [0,1) of shape 2,2
print(np.random.rand(2,2))

# Normal distribution with mean=0 and variance=1 of shape 2,2
print(np.random.randn(2,2))

# Random integers between [0, 10) of shape 2,2
print(np.random.randint(0, 10, size=[2,2]))

# One random number between [0,1)
print(np.random.random())

# Random numbers between [0,1) of shape 2,2
print(np.random.random(size=[2,2]))

# Pick 10 items from a given list, with equal probability
print(np.random.choice(['a', 'e', 'i', 'o', 'u'], size=10))

# Pick 10 items from a given list with a predefined probability 'p'
print(np.random.choice(['a', 'e', 'i', 'o', 'u'], size=10,
p=[0.3, .1, 0.1, 0.4, 0.1]))  # picks more o's

#> [[ 0.84  0.7 ]
#>  [ 0.52  0.8 ]]

#> [[-0.06 -1.55]
#>  [ 0.47 -0.04]]

#> [[4 0]
#>  [8 7]]

#> 0.08737272424956832

#> [[ 0.45  0.78]
#>  [ 0.03  0.74]]

#> ['i' 'a' 'e' 'e' 'a' 'u' 'o' 'e' 'i' 'u']
#> ['o' 'a' 'e' 'a' 'a' 'o' 'o' 'o' 'a' 'o']
```

Maintenant, à chaque fois que vous exécutez l'une des fonctions ci-dessus, vous obtenez un ensemble différent de nombres aléatoires.

Si vous souhaitez répéter le même ensemble de nombres aléatoires à chaque fois, vous devez définir la graine (seed) ou l'état aléatoire. La graine peut être n'importe quelle valeur. La seule exigence est que vous devez définir la graine avec la même valeur à chaque fois que vous souhaitez générer le même ensemble de nombres aléatoires.

Une fois que `np.random.RandomState` est créé, toutes les fonctions du module `np.random` deviennent disponibles pour l'objet `randomstate` créé.

```
# Create the random state
rn = np.random.RandomState(100)

# Create random numbers between [0,1) of shape 2,2
print(rn.rand(2,2))

#> [[ 0.54  0.28]
#>  [ 0.42  0.84]]
# Set the random seed
np.random.seed(100)

# Create random numbers between [0,1) of shape 2,2
print(np.random.rand(2,2))

#> [[ 0.54  0.28]
#>  [ 0.42  0.84]]
```

7.3 Comment obtenir les éléments uniques et leurs fréquences ?

La méthode np.unique peut être utilisée pour obtenir les éléments uniques. Si vous souhaitez obtenir les comptages de répétition de chaque élément, définissez le paramètre return_counts sur True.

```
# Create random integers of size 10 between [0,10)
np.random.seed(100)
arr_rand = np.random.randint(0, 10, size=10)
print(arr_rand)

#> [8 8 3 7 7 0 4 2 5 2]
# Get the unique items and their counts
uniqs, counts = np.unique(arr_rand, return_counts=True)
print("Unique items : ", uniqs)
print("Counts       : ", counts)

#> Unique items :  [0 2 3 4 5 7 8]
#> Counts       :  [1 2 1 1 1 2 2]
```