eBook Gratuit

APPRENEZ numpy

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#numpy

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: numpy

It is an unofficial and free numpy ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official numpy.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Commencer avec numpy

Remarques

NumPy (prononcé "numb pie" ou parfois "numb pea") 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 à utiliser sur ces baies.

Versions

Version	Date de sortie
1.3.0	2009-03-20
1.4.0	2010-07-21
1.5.0	2010-11-18
1.6.0	2011-05-15
1.6.1	2011-07-24
1.6.2	2012-05-20
1.7.0	2013-02-12
1.7.1	2013-04-07
1.7.2	2013-12-31
1.8.0	2013-11-10
1.8.1	2014-03-26
1.8.2	2014-08-09
1.9.0	2014-09-07
1.9.1	2014-11-02
1.9.2	2015-03-01
1.10.0	2015-10-07
1.10.1	2015-10-12
1.10.2	2015-12-14

Version	Date de sortie
1.10.4 *	2016-01-07
1.11.0	2016-05-29

Examples

Installation sur Mac

Le moyen le plus simple de configurer NumPy sur Mac est d' utiliser pip

```
pip install numpy
```

Installation avec Conda .
Conda disponible pour Windows, Mac et Linux

1. Installez Conda. Il existe deux manières d'installer Conda, soit avec Anaconda (package complet, include numpy) ou Miniconda (uniquement Conda, Python et les packages dont ils dépendent, sans aucun package supplémentaire). Anaconda et Miniconda installent le même Conda.
2. Commande supplémentaire pour Miniconda, tapez la commande `conda install numpy`

Installation sous Windows

L'installation de Numpy via pypi (l'index de package par défaut utilisé par pip) échoue généralement sur les ordinateurs Windows. La méthode la plus simple pour installer Windows est d'utiliser des fichiers binaires précompilés.

Le site de Christopher Gohkle est une source pour les roues précompilées de nombreux paquets. Choisissez une version en fonction de votre version et de votre système Python. Un exemple pour Python 3.5 sur un système 64 bits:

1. Téléchargez `numpy-1.11.1+mkl-cp35-cp35m-win_amd64.whl` d' ici
2. Ouvrez un terminal Windows (cmd ou powershell)
3. Tapez la commande `pip install C:\path_to_download\numpy-1.11.1+mkl-cp35-cp35m-win_amd64.whl`

Si vous ne voulez pas gâcher avec des packages uniques, vous pouvez utiliser la distribution Winpython qui regroupe la plupart des packages et fournit un environnement confiné avec lequel travailler. De même, la distribution Python Anaconda est pré-installée avec numpy et de nombreux autres paquets communs.

Une autre source populaire est le `conda` paquets de `conda` , qui prend également en charge les environnements virtuels .

1. Téléchargez et installez `conda` .
2. Ouvrez un terminal Windows.

3. Tapez la commande `conda install numpy`

Installation sous Linux

NumPy est disponible dans les référentiels par défaut des distributions Linux les plus populaires et peut être installé de la même manière que les packages d'une distribution Linux sont généralement installés.

Certaines distributions Linux ont des packages NumPy différents pour Python 2.x et Python 3.x. Dans Ubuntu et Debian, installez `numpy` au niveau du système en utilisant le gestionnaire de paquets APT:

```
sudo apt-get install python-numpy
sudo apt-get install python3-numpy
```

Pour les autres distributions, utilisez leurs gestionnaires de paquets, comme zypper (Suse), yum (Fedora), etc.

`numpy` peut également être installé avec le gestionnaire de package de Python `pip` pour Python 2 et `pip3` pour Python 3:

```
pip install numpy  # install numpy for Python 2
pip3 install numpy  # install numpy for Python 3
```

`pip` est disponible dans les référentiels par défaut des distributions Linux les plus populaires et peut être installé pour Python 2 et Python 3 en utilisant:

```
sudo apt-get install python-pip  # pip for Python 2
sudo apt-get install python3-pip  # pip for Python 3
```

Après l'installation, utilisez `pip` pour Python 2 et `pip3` pour Python 3 afin d'utiliser pip pour installer les paquets Python. Mais notez que vous devrez peut-être installer de nombreuses dépendances, qui sont nécessaires pour générer numpy à partir des sources (y compris les packages de développement, les compilateurs, fortran, etc.).

Outre l'installation de `numpy` au niveau du système, il est également courant (voire même fortement recommandé) d'installer `numpy` dans des environnements virtuels en utilisant des packages Python populaires tels que `virtualenv` . Dans Ubuntu, `virtualenv` peut être installé en utilisant:

```
sudo apt-get install virtualenv
```

Ensuite, créez et activez virtualenv pour Python 2 ou Python 3, puis utilisez `pip` pour installer `numpy` :

```
virtualenv venv  # create virtualenv named venv for Python 2
virtualenv venv -p python3  # create virtualenv named venv for Python 3
source venv/bin/activate  # activate virtualenv named venv
pip install numpy  # use pip for Python 2 and Python 3; do not use pip3 for Python3
```

Importation de base

Importez le module numpy pour en utiliser une partie.

```
import numpy as np
```

La plupart des exemples utiliseront `np` comme raccourci pour numpy. Supposons que "np" signifie "numpy" dans les exemples de code.

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

Bloc-notes temporaire Jupyter hébergé par Rackspace

Jupyter Notebooks est un environnement de développement interactif basé sur un navigateur. Ils ont été initialement développés pour exécuter le calcul python et, en tant que tels, ils fonctionnent très bien avec numpy. Pour essayer numpy dans un ordinateur portable Jupyter sans l'installer complètement sur son système local, Rackspace fournit gratuitement des ordinateurs portables temporaires sur tmpnb.org .

Remarque: il ne s'agit pas d'un service propriétaire avec toute sorte de vente. Jupyter est une technologie entièrement ouverte développée par UC Berkeley et Cal Poly San Luis Obispo. Rackspace fait don de ce service dans le cadre du processus de développement.

Pour essayer `numpy` sur tmpnb.org:

1. visitez tmpnb.org
2. soit sélectionnez `Welcome to Python.ipynb` ou
3. Nouveau >> Python 2 ou
4. Nouveau >> Python 3

Lire Commencer avec numpy en ligne: https://riptutorial.com/fr/numpy/topic/823/commencer-avec-numpy

Chapitre 2: Algèbre linéaire avec np.linalg

Remarques

Depuis la version 1.8, plusieurs routines de `np.linalg` peuvent fonctionner sur une «pile» de matrices. Autrement dit, la routine peut calculer les résultats pour plusieurs matrices si elles sont empilées ensemble. Par exemple, `A` est interprété comme deux matrices empilées 3 par 3:

```
np.random.seed(123)
A = np.random.rand(2,3,3)
b = np.random.rand(2,3)
x = np.linalg.solve(A, b)

print np.dot(A[0,:,:], x[0,:])
# array([ 0.53155137,  0.53182759,  0.63440096])

print b[0,:]
# array([ 0.53155137,  0.53182759,  0.63440096])
```

Les documents officiels `np` spécifient via des spécifications de paramètres telles `a : (..., M, M) array_like` .

Examples

Résoudre des systèmes linéaires avec np.solve

Considérons les trois équations suivantes:

```
x0 + 2 * x1 + x2 = 4
         x1 + x2 = 3
x0 +          x2 = 5
```

Nous pouvons exprimer ce système comme une équation matricielle `A * x = b` avec:

```
A = np.array([[1, 2, 1],
              [0, 1, 1],
              [1, 0, 1]])
b = np.array([4, 3, 5])
```

Ensuite, utilisez `np.linalg.solve` pour résoudre `x` :

```
x = np.linalg.solve(A, b)
# Out: x = array([ 1.5, -0.5,  3.5])
```

`A` doit être un carré et une matrice complète: toutes ses lignes doivent être linéairement indépendantes. `A` doit être inversible / non singulier (son déterminant n'est pas zéro). Par exemple, si une ligne de `A` est un multiple d'un autre, l'appel de `linalg.solve` soulèvera `LinAlgError: Singular matrix` :

```
A = np.array([[1, 2, 1],
              [2, 4, 2],   # Note that this row 2 * the first row
              [1, 0, 1]])
b = np.array([4,8,5])
```

De tels systèmes peuvent être résolus avec `np.linalg.lstsq` .

Trouver la solution des moindres carrés à un système linéaire avec np.linalg.lstsq

Les moindres carrés constituent une approche standard des problèmes comportant plus d'équations que d'inconnues, également connus sous le nom de systèmes surdéterminés.

Considérons les quatre équations:

```
x0 + 2 * x1 + x2 = 4
x0 + x1 + 2 * x2 = 3
2 * x0 + x1 + x2 = 5
x0 + x1 + x2 = 4
```

Nous pouvons exprimer cela comme une multiplication matricielle `A * x = b` :

```
A = np.array([[1, 2, 1],
              [1,1,2],
              [2,1,1],
              [1,1,1]])
b = np.array([4,3,5,4])
```

Ensuite, résolvez avec `np.linalg.lstsq` :

```
x, residuals, rank, s = np.linalg.lstsq(A,b)
```

`x` est la solution, les `residuals` la somme, `rank` le `rank` la matrice de l'entrée `A` et `s` les valeurs singulières de `A` Si `b` a plus d'une dimension, `lstsq` résoudra le système correspondant à chaque colonne de `b` :

```
A = np.array([[1, 2, 1],
              [1,1,2],
              [2,1,1],
              [1,1,1]])
b = np.array([[4,3,5,4],[1,2,3,4]]).T # transpose to align dimensions
x, residuals, rank, s = np.linalg.lstsq(A,b)
print x # columns of x are solutions corresponding to columns of b
#[[ 2.05263158  1.63157895]
# [ 1.05263158 -0.36842105]
# [ 0.05263158  0.63157895]]
print residuals # also one for each column in b
#[ 0.84210526  5.26315789]
```

`rank` et `s` ne dépendent que de `A` et sont donc les mêmes que ci-dessus.

Lire Algèbre linéaire avec np.linalg en ligne: https://riptutorial.com/fr/numpy/topic/3753/algebre-

Chapitre 3: Fichier IO avec numpy

Examples

Enregistrement et chargement de tableaux numpy à l'aide de fichiers binaires

```
x = np.random.random([100,100])
x.tofile('/path/to/dir/saved_binary.npy')
y = fromfile('/path/to/dir/saved_binary.npy')
z = y.reshape(100,100)
all(x==z)
# Output:
#     True
```

Chargement de données numériques à partir de fichiers texte avec une structure cohérente

La fonction `np.loadtxt` peut être utilisée pour lire des fichiers de type csv:

```
# File:
#    # Col_1 Col_2
#    1, 1
#    2, 4
#    3, 9
np.loadtxt('/path/to/dir/csvlike.txt', delimiter=',', comments='#')
# Output:
# array([[ 1.,  1.],
#        [ 2.,  4.],
#        [ 3.,  9.]])
```

Le même fichier peut être lu en utilisant une expression régulière avec `np.fromregex` :

```
np.fromregex('/path/to/dir/csvlike.txt', r'(\d+),\s(\d+)', np.int64)
# Output:
# array([[1, 1],
#        [2, 4],
#        [3, 9]])
```

Enregistrement de données sous forme de fichier ASCII de style CSV

Analog to `np.loadtxt` , `np.savetxt` peut être utilisé pour enregistrer des données dans un fichier ASCII

```
import numpy as np
x = np.random.random([100,100])
np.savetxt("filename.txt", x)
```

Pour contrôler le formatage:

```
np.savetxt("filename.txt", x, delimiter=", " ,
    newline="\n", comments="$ ", fmt="%1.2f",
    header="commented example text")
```

Sortie:

```
$ commented example text
0.30, 0.61, 0.34, 0.13, 0.52, 0.62, 0.35, 0.87, 0.48, [...]
```

Lecture de fichiers CSV

Trois fonctions principales disponibles (description à partir des pages de manuel):

> `fromfile` - Un moyen très efficace de lire des données binaires avec un type de données connu, ainsi que d'analyser des fichiers texte simplement formatés. Les données écrites à l'aide de la méthode tofile peuvent être lues à l'aide de cette fonction.
>
> `genfromtxt` - Charge des données à partir d'un fichier texte, avec les valeurs manquantes traitées comme spécifié. Chaque ligne après les premières lignes skip_header est fractionnée au caractère délimiteur et les caractères qui suivent le caractère commentaires sont ignorés.
>
> `loadtxt` - Charge des données à partir d'un fichier texte. Chaque ligne du fichier texte doit avoir le même nombre de valeurs.

`genfromtxt` est une fonction wrapper pour `loadtxt` . `genfromtxt` est le plus simple à utiliser car il dispose de nombreux paramètres pour gérer le fichier d'entrée.

Nombre cohérent de colonnes, type de données cohérent (numérique ou chaîne):

Étant donné un fichier d'entrée, `myfile.csv` avec le contenu:

```
#descriptive text line to skip
1.0, 2, 3
4, 5.5, 6

import numpy as np
np.genfromtxt('path/to/myfile.csv',delimiter=',',skiprows=1)
```

donne un tableau:

```
array([[ 1. ,  2. ,  3. ],
       [ 4. ,  5.5,  6. ]])
```

Nombre de colonnes cohérent, type de données mixte (entre les colonnes):

```
1   2.0000  buckle_my_shoe
3   4.0000  margery_door
```

```
import numpy as np
np.genfromtxt('filename', dtype= None)


array([(1, 2.0, 'buckle_my_shoe'), (3, 4.0, 'margery_door')],
dtype=[('f0', '<i4'), ('f1', '<f8'), ('f2', '|S14')])
```

Notez que l'utilisation de `dtype=None` entraîne un recarray.

Nombre incohérent de colonnes:

file: 1 2 3 4 5 6 7 8 9 10 11 22 13 14 15 16 17 18 19 20 21 22 23 24

Dans un tableau à une seule ligne:

```
result=np.fromfile(path_to_file,dtype=float,sep="\t",count=-1)
```

Lire Fichier IO avec numpy en ligne: https://riptutorial.com/fr/numpy/topic/4973/fichier-io-avec-numpy

Chapitre 4: Filtrage des données

Examples

Filtrage des données avec un tableau booléen

Quand un seul argument est fourni à numpy's `where` function, il retourne les indices du tableau en entrée (la `condition`) qui ont la valeur true (même comportement que `numpy.nonzero`). Cela peut être utilisé pour extraire les index d'un tableau qui satisfait une condition donnée.

```
import numpy as np

a = np.arange(20).reshape(2,10)
# a = array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
#            [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])

# Generate boolean array indicating which values in a are both greater than 7 and less than 13
condition = np.bitwise_and(a>7, a<13)
# condition = array([[False, False, False, False, False, False, False, False,  True, True],
#                    [True,  True,  True, False, False, False, False, False, False, False]],
dtype=bool)

# Get the indices of a where the condition is True
ind = np.where(condition)
# ind = (array([0, 0, 1, 1, 1]), array([8, 9, 0, 1, 2]))

keep = a[ind]
# keep = [ 8  9 10 11 12]
```

Si vous n'avez pas besoin des index, vous pouvez le faire en une étape en utilisant `extract` , où vous devez spécifier la `condition` comme premier argument, mais donner au `array` les valeurs d'où la condition est vraie en tant que second argument.

```
# np.extract(condition, array)
keep = np.extract(condition, a)
# keep = [ 8  9 10 11 12]
```

Deux arguments supplémentaires `x` et `y` peuvent être fournis à l' `where` , dans ce cas, la sortie contiendra les valeurs de `x` où la condition est `True` et les valeurs de `y` où la condition est `False` .

```
# Set elements of a which are NOT greater than 7 and less than 13 to zero, np.where(condition,
x, y)
a = np.where(condition, a, a*0)
print(a)
# Out: array([[ 0,  0,  0,  0,  0,  0,  0,  0,  8,  9],
#             [10, 11, 12,  0,  0,  0,  0,  0,  0,  0]])
```

Filtrage direct des indices

Pour les cas simples, vous pouvez filtrer les données directement.

```
a = np.random.normal(size=10)
print(a)
#[-1.19423121  1.10481873  0.26332982 -0.53300387 -0.04809928  1.77107775
# 1.16741359  0.17699948 -0.06342169 -1.74213078]
b = a[a>0]
print(b)
#[ 1.10481873  0.26332982  1.77107775  1.16741359  0.17699948]
```

Lire Filtrage des données en ligne: https://riptutorial.com/fr/numpy/topic/6187/filtrage-des-donnees

Chapitre 5: Générer des données aléatoires

Introduction

Le module `random` de NumPy fournit des méthodes pratiques pour générer des données aléatoires ayant la forme et la distribution souhaitées.

Voici la documentation officielle .

Examples

Créer un tableau aléatoire simple

```
# Generates 5 random numbers from a uniform distribution [0, 1)
np.random.rand(5)
# Out: array([ 0.4071833 ,  0.069167  ,  0.69742877,  0.45354268,  0.7220556 ])
```

Mettre la graine

En utilisant `random.seed` :

```
np.random.seed(0)
np.random.rand(5)
# Out: array([ 0.5488135 ,  0.71518937,  0.60276338,  0.54488318,  0.4236548 ])
```

En créant un objet générateur de nombres aléatoires:

```
prng = np.random.RandomState(0)
prng.rand(5)
# Out: array([ 0.5488135 ,  0.71518937,  0.60276338,  0.54488318,  0.4236548 ])
```

Créer des entiers aléatoires

```
# Creates a 5x5 random integer array ranging from 10 (inclusive) to 20 (inclusive)
np.random.randint(10, 20, (5, 5))

'''
Out: array([[12, 14, 17, 16, 18],
            [18, 11, 16, 17, 17],
            [18, 11, 15, 19, 18],
            [19, 14, 13, 10, 13],
            [15, 10, 12, 13, 18]])
'''
```

Sélection d'un échantillon aléatoire dans un tableau

```
letters = list('abcde')
```

Sélectionnez trois lettres au hasard (*avec remplacement* - le même article peut être choisi plusieurs fois):

```
np.random.choice(letters, 3)
'''
Out: array(['e', 'e', 'd'],
      dtype='<U1')
'''
```

Échantillonnage sans remplacement:

```
np.random.choice(letters, 3, replace=False)
'''
Out: array(['a', 'c', 'd'],
      dtype='<U1')
'''
```

Attribuer une probabilité à chaque lettre:

```
# Choses 'a' with 40% chance, 'b' with 30% and the remaining ones with 10% each
np.random.choice(letters, size=10, p=[0.4, 0.3, 0.1, 0.1, 0.1])

'''
Out: array(['a', 'b', 'e', 'b', 'a', 'b', 'b', 'c', 'a', 'b'],
  dtype='<U1')
'''
```

Génération de nombres aléatoires tirés de distributions spécifiques

Dessiner des échantillons d'une distribution normale (gaussienne)

```
# Generate 5 random numbers from a standard normal distribution
# (mean = 0, standard deviation = 1)
np.random.randn(5)
# Out: array([-0.84423086,  0.70564081, -0.39878617, -0.82719653, -0.4157447 ])

# This result can also be achieved with the more general np.random.normal
np.random.normal(0, 1, 5)
# Out: array([-0.84423086,  0.70564081, -0.39878617, -0.82719653, -0.4157447 ])

# Specify the distribution's parameters
# Generate 5 random numbers drawn from a normal distribution with mean=70, std=10
np.random.normal(70, 10, 5)
# Out: array([ 72.06498837,  65.43118674,  59.40024236,  76.14957316,  84.29660766])
```

Il existe plusieurs distributions supplémentaires dans `numpy.random` , par exemple `poisson` , `binomial` et `logistic`

```
np.random.poisson(2.5, 5)  # 5 numbers, lambda=5
# Out: array([0, 2, 4, 3, 5])

np.random.binomial(4, 0.3, 5)  # 5 numbers, n=4, p=0.3
# Out: array([1, 0, 2, 1, 0])
```

```
np.random.logistic(2.3, 1.2, 5)  # 5 numbers, location=2.3, scale=1.2
# Out: array([ 1.23471936,  2.28598718, -0.81045893,  2.2474899 ,  4.15836878])
```

Lire Générer des données aléatoires en ligne: https://riptutorial.com/fr/numpy/topic/2060/generer-des-donnees-aleatoires

Chapitre 6: Indexation booléenne

Examples

Créer un tableau booléen

Un tableau booléen peut être créé manuellement en utilisant `dtype=bool` lors de la création du tableau. Les valeurs autres que `0` , `None` , `False` ou les chaînes vides sont considérées comme True.

```
import numpy as np

bool_arr = np.array([1, 0.5, 0, None, 'a', '', True, False], dtype=bool)
print(bool_arr)
# output: [ True  True False False  True False  True False]
```

Numpy crée automatiquement un tableau booléen lorsque des comparaisons sont effectuées entre des tableaux et des scalaires ou entre des tableaux de la même forme.

```
arr_1 = np.random.randn(3, 3)
arr_2 = np.random.randn(3, 3)

bool_arr = arr_1 < 0.5
print(bool_arr.dtype)
# output: bool

bool_arr = arr_1 < arr_2
print(bool_arr.dtype)
# output: bool
```

Lire Indexation booléenne en ligne: https://riptutorial.com/fr/numpy/topic/6072/indexation-booleenne

Chapitre 7: numpy.cross

Syntaxe

- `numpy.cross(a, b)` # produit croisé de *a* et *b* (ou vecteurs dans *a* et *b*)
- `numpy.cross(a, b, axisa=-1)` #cross produit de vecteurs dans *a* avec *b* , st vecteurs dans a sont disposés le long de l'axe *axisa*
- `numpy.cross(a, b, axisa=-1, axisb=-1, axisc=-1)` # produits croisés de vecteurs dans *a* et *b* , vecteurs de sortie disposés selon l'axe spécifié par *axisc*
- `numpy.cross(a, b, axis=None)` # produit croisé des vecteurs en *a* et *b* , vecteurs en *a* , *b* et en sortie disposés suivant l'axe des *axes*

Paramètres

Colonne	Colonne
un B	Dans l'usage le plus simple, `a` et `b` sont deux vecteurs à 2 ou 3 éléments. Ils peuvent également être des tableaux de vecteurs (c.-à-d. Des matrices à deux dimensions). Si `a` est un tableau et que 'b' est un vecteur, `cross(a,b)` renvoie un tableau dont les éléments sont les produits croisés de chaque vecteur dans `a` vecteur `b` . Le `b` est un tableau et `a` est un seul vecteur, le `cross(a,b)` renvoie un tableau dont les éléments sont les produits croisés d' `a` avec chaque vecteur de `b` . `a` et `b` peuvent tous deux être des tableaux s'ils ont la même forme. Dans ce cas, `cross(a,b)` renvoie `cross(a[0],b[0]), cross(a[1], b[1]), ...`
axisa / b	Si `a` est un tableau, des vecteurs peuvent être disposés sur l'axe variant le plus rapidement, l'axe variant le plus lent ou quelque chose entre les deux. `axisa` dit en `cross()` comment les vecteurs sont disposés dans `a` . Par défaut, il prend la valeur de l'axe variant le plus lentement. `axisb` fonctionne de la même manière avec input `b` . Si la sortie de `cross()` doit être un tableau, les vecteurs de sortie peuvent être disposés sur différents axes de tableau; `axisc` indique à `cross` comment disposer les vecteurs dans son tableau de sortie. Par défaut, `axisc` indique l'axe qui varie le plus lentement.
axe	Un paramètre de commodité qui définit `axisa` , `axisb` et `axisc` à la même valeur si vous le souhaitez. Si l' `axis` et l'un des autres paramètres sont présents dans l'appel, la valeur de l' `axis` remplacera les autres valeurs.

Examples

Produit croisé de deux vecteurs

Numpy fournit une fonction `cross` pour le calcul des produits vectoriels croisés. Le produit croisé

des vecteurs `[1, 0, 0]` et `[0, 1, 0]` est `[0, 0, 1]` . Numpy nous dit:

```
>>> a = np.array([1, 0, 0])
>>> b = np.array([0, 1, 0])
>>> np.cross(a, b)
array([0, 0, 1])
```

comme prévu.

Alors que les produits croisés ne sont normalement définis que pour les vecteurs tridimensionnels. Cependant, l'un des arguments de la fonction Numpy peut être deux vecteurs d'éléments. Si le vecteur `c` est donné comme `[c1, c2]` , Numpy affecte zéro à la troisième dimension: `[c1, c2, 0]` . Alors,

```
>>> c = np.array([0, 2])
>>> np.cross(a, c)
array([0, 0, 2])
```

Contrairement au `dot` qui existe à la fois en tant que fonction Numpy et méthode de `ndarray` , le `cross` existe uniquement en tant que fonction autonome:

```
>>> a.cross(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'numpy.ndarray' object has no attribute 'cross'
```

Produits Cross Cross multiples avec un appel

Chacune des entrées peut être un tableau de trois (ou deux) vecteurs d'éléments.

```
>>> a=np.array([[1,0,0],[0,1,0],[0,0,1]])
>>> b=np.array([1,0,0])
>>> np.cross(a,b)
array([[ 0,  0,  0],
       [ 0,  0, -1],
       [ 0,  1,  0]])
```

Le résultat dans ce cas est array ([np.cross (a [0], b), np.cross (a [1], b), np.cross (a [2], b)])

`b` peut également être un tableau de vecteurs d'éléments 3 (ou 2), mais il doit avoir la même forme que `a` . Sinon, le calcul échoue avec une erreur de "non-concordance de forme". Nous pouvons donc avoir

```
>>> b=np.array([[0,0,1],[1,0,0],[0,1,0]])
>>> np.cross(a,b)
array([[ 0, -1,  0],
       [ 0,  0, -1],
       [-1,  0,  0]])
```

et maintenant le résultat est `array([np.cross(a[0],b[0]), np.cross(a[1],b[1]), np.cross(a[2],b[2])])`

Plus de flexibilité avec plusieurs produits croisés

Dans nos deux derniers exemples, numpy supposait qu'un `a[0,:]` était le premier vecteur, `a[1,:]` le deuxième et `a[2,:]` le troisième. Numpy.cross a un argument optionnel axisa qui nous permet de spécifier quel axe définit les vecteurs. Alors,

```
>>> a=np.array([[1,1,1],[0,1,0],[1,0,-1]])
>>> b=np.array([0,0,1])
>>> np.cross(a,b)
array([[ 1, -1,  0],
       [ 1,  0,  0],
       [ 0, -1,  0]])
>>> np.cross(a,b,axisa=0)
array([[ 0, -1,  0],
       [ 1, -1,  0],
       [ 0, -1,  0]])
>>> np.cross(a,b,axisa=1)
array([[ 1, -1,  0],
       [ 1,  0,  0],
       [ 0, -1,  0]])
```

Le résultat `axisa=1` et le résultat par défaut sont tous deux `(np.cross([1,1,1],b), np.cross([0,1,0],b), np.cross([1,0,-1],b))` . Par défaut, `axisa` indique toujours le dernier axe (variant le plus lentement) du tableau. Le résultat `axisa=0` est `(np.cross([1,0,1],b), np.cross([1,1,0],b), np.cross([1,0,-1],b))` .

Un paramètre optionnel similaire, `axisb` , exécute la même fonction pour l'entrée `b` , s'il s'agit également d'un tableau à deux dimensions.

Les paramètres axisa et axisb indiquent comment distribuer les données d'entrée. Un troisième paramètre, axisc, indique comment distribuer la sortie si `a` ou `b` est multidimensionnel. En utilisant les mêmes entrées `a` et `b` que ci-dessus, nous obtenons

```
>>> np.cross(a,b,1)
array([[ 1, -1,  0],
       [ 1,  0,  0],
       [ 0, -1,  0]])
>>> np.cross(a,b,1,axisc=0)
array([[ 1,  1,  0],
       [-1,  0, -1],
       [ 0,  0,  0]])
>>> np.cross(a,b,1,axisc=1)
array([[ 1, -1,  0],
       [ 1,  0,  0],
       [ 0, -1,  0]])
```

Donc `axisc=1` et `axisc` par défaut `axisc` tous deux le même résultat, c'est-à-dire que les éléments de chaque vecteur sont contigus dans l'index le plus rapide du tableau de sortie. axisc est par défaut le dernier axe du tableau. `axisc=0` distribue les éléments de chaque vecteur sur la dimension la plus lente du tableau.

Si vous voulez que `axisa` , `axisb` et `axisc` aient tous la même valeur, vous n'avez pas besoin de définir les trois paramètres. Vous pouvez définir un quatrième paramètre, `axis` , à la valeur unique

requise et les trois autres paramètres seront automatiquement définis. l'axe remplace axisa, axisb ou axisc si l'un d'entre eux est présent dans l'appel de fonction.

Lire numpy.cross en ligne: https://riptutorial.com/fr/numpy/topic/6166/numpy-cross

Chapitre 8: numpy.dot

Syntaxe

- numpy.dot (a, b, out = None)

Paramètres

prénom	Détails
une	un tableau numpy
b	un tableau numpy
en dehors	un tableau numpy

Remarques

numpy.dot

Renvoie le produit scalaire de a et b. Si a et b sont à la fois des scalaires ou des deux tableaux 1-D, un scalaire est renvoyé; sinon un tableau est retourné. Si out est donné, alors il est renvoyé.

Examples

Multiplication de matrice

La multiplication matricielle peut se faire de deux manières équivalentes avec la fonction point. L'une des méthodes consiste à utiliser la fonction membre dot de numpy.ndarray.

```
>>> import numpy as np
>>> A = np.ones((4,4))
>>> A
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
>>> B = np.ones((4,2))
>>> B
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
>>> A.dot(B)
array([[ 4.,  4.],
       [ 4.,  4.],
       [ 4.,  4.],
       [ 4.,  4.]])
```

La deuxième façon de procéder à la multiplication matricielle est d'utiliser la fonction de bibliothèque numpy.

```
>>> np.dot(A,B)
array([[ 4.,  4.],
       [ 4.,  4.],
       [ 4.,  4.],
       [ 4.,  4.]])
```

Produits de points Vector

La fonction de point peut également être utilisée pour calculer des produits de points vectoriels entre deux tableaux numpy unidimensionnels.

```
>>> v = np.array([1,2])
>>> w = np.array([1,2])
>>> v.dot(w)
5
>>> np.dot(w,v)
5
>>> np.dot(v,w)
5
```

Le paramètre out

La fonction numpy dot a un paramètre optionnel out = None. Ce paramètre vous permet de spécifier un tableau dans lequel écrire le résultat. Ce tableau doit avoir exactement la même forme et le même type que le tableau qui aurait été renvoyé, ou une exception sera lancée.

```
>>> I = np.eye(2)
>>> I
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> result = np.zeros((2,2))
>>> result
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> np.dot(I, I, out=result)
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> result
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Essayons de changer le type de résultat en int.

```
>>> np.dot(I, I, out=result)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: output array is not acceptable (must have the right type, nr dimensions, and be a
C-Array)
```

Et si nous essayons d'utiliser un ordre de mémoire sous-jacent différent, disons le style Fortran

(pour que les colonnes soient contiguës au lieu de lignes), une erreur se produit également.

```
>>> result = np.zeros((2,2), order='F')
>>> np.dot(I, I, out=result)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: output array is not acceptable (must have the right type, nr dimensions, and be a
C-Array)
```

Opérations matricielles sur des tableaux de vecteurs

numpy.dot peut être utilisé pour multiplier une liste de vecteurs par une matrice mais l'orientation des vecteurs doit être verticale pour qu'une liste de huit vecteurs à deux composantes apparaisse comme deux vecteurs de composants:

```
>>> a
array([[ 1.,  2.],
       [ 3.,  1.]])
>>> b
array([[  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.,  13.,  14.,  15.,  16.]])
>>> np.dot(a, b)
array([[ 19.,  22.,  25.,  28.,  31.,  34.,  37.,  40.],
       [ 12.,  16.,  20.,  24.,  28.,  32.,  36.,  40.]])
```

Si la liste des vecteurs est disposée avec ses axes dans le sens inverse (ce qui est souvent le cas), alors le tableau doit être transposé avant et après l'opération de point comme ceci:

```
>>> b
array([[  1.,   9.],
       [  2.,  10.],
       [  3.,  11.],
       [  4.,  12.],
       [  5.,  13.],
       [  6.,  14.],
       [  7.,  15.],
       [  8.,  16.]])
>>> np.dot(a, b.T).T
array([[ 19.,  12.],
       [ 22.,  16.],
       [ 25.,  20.],
       [ 28.,  24.],
       [ 31.,  28.],
       [ 34.,  32.],
       [ 37.,  36.],
       [ 40.,  40.]])
```

Bien que la fonction de point soit très rapide, il est parfois préférable d’utiliser einsum. L'équivalent de ce qui précède serait:

```
>>> np.einsum('...ij,...j', a, b)
```

Ce qui est un peu plus lent mais permet de multiplier une liste de sommets par une liste de matrices correspondante. Ce serait un processus très compliqué en utilisant dot:

```
>>> a
array([[[ 0,  1],
        [ 2,  3]],
       [[ 4,  5],
        [ 6,  7]],
       [[ 8,  9],
        [10, 11]],
       [[12, 13],
        [14, 15]],
       [[16, 17],
        [18, 19]],
       [[20, 21],
        [22, 23]],
       [[24, 25],
        [26, 27]],
       [[28, 29],
        [30, 31]]])
>>> np.einsum('...ij,...j', a, b)
array([[   9.,   29.],
       [  58.,   82.],
       [ 123.,  151.],
       [ 204.,  236.],
       [ 301.,  337.],
       [ 414.,  454.],
       [ 543.,  587.],
       [ 688.,  736.]])
```

numpy.dot peut être utilisé pour trouver le produit scalaire de chaque vecteur dans une liste avec un vecteur correspondant dans une autre liste, ce qui est assez désordonné et lent comparé à la multiplication par éléments et à la sommation le long du dernier axe. Quelque chose comme ça (ce qui nécessite un tableau beaucoup plus grand pour être calculé mais généralement ignoré)

```
>>> np.diag(np.dot(b,b.T))
array([  82.,  104.,  130.,  160.,  194.,  232.,  274.,  320.])
```

Produit Dot utilisant la multiplication et la sommation par éléments

```
>>> (b * b).sum(axis=-1)
array([  82.,  104.,  130.,  160.,  194.,  232.,  274.,  320.])
```

Utiliser einsum pourrait être réalisé avec

```
>>> np.einsum('...j,...j', b, b)
array([  82.,  104.,  130.,  160.,  194.,  232.,  274.,  320.])
```

Lire numpy.dot en ligne: https://riptutorial.com/fr/numpy/topic/3198/numpy-dot

Chapitre 9: Régression linéaire simple

Introduction

Adapter une ligne (ou une autre fonction) à un ensemble de points de données.

Examples

Utiliser np.polyfit

Nous créons un ensemble de données que nous ajustons ensuite avec une ligne droite $ f (x) = mx + c $.

```
npoints = 20
slope = 2
offset = 3
x = np.arange(npoints)
y = slope * x + offset + np.random.normal(size=npoints)
p = np.polyfit(x,y,1)           # Last argument is degree of polynomial
```

Pour voir ce que nous avons fait:

```
import matplotlib.pyplot as plt
f = np.poly1d(p)                # So we can call f(x)
fig = plt.figure()
ax  = fig.add_subplot(111)
plt.plot(x, y, 'bo', label="Data")
plt.plot(x,f(x), 'b-',label="Polyfit")
plt.show()
```

Remarque: Cet exemple suit de très près la documentation numpy disponible sur https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html .

Utiliser np.linalg.lstsq

Nous utilisons le même jeu de données qu'avec polyfit:

```
npoints = 20
slope = 2
offset = 3
x = np.arange(npoints)
y = slope * x + offset + np.random.normal(size=npoints)
```

Maintenant, nous essayons de trouver une solution en minimisant le système d'équations linéaires A b = c en minimisant | cA b | ** 2

```
import matplotlib.pyplot as plt # So we can plot the resulting fit
A = np.vstack([x,np.ones(npoints)]).T
```

```
m, c = np.linalg.lstsq(A, y)[0] # Don't care about residuals right now
fig = plt.figure()
ax  = fig.add_subplot(111)
plt.plot(x, y, 'bo', label="Data")
plt.plot(x, m*x+c, 'r--',label="Least Squares")
plt.show()
```

Remarque: Cet exemple suit de près la documentation numpy à l' adresse https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html .

Lire Régression linéaire simple en ligne: https://riptutorial.com/fr/numpy/topic/8808/regression-lineaire-simple

Chapitre 10: Sauvegarde et chargement de tableaux

Introduction

Les tableaux Numpy peuvent être enregistrés et chargés de différentes manières.

Examples

Utiliser numpy.save et numpy.load

np.save et np.load fournissent un cadre facile à utiliser pour enregistrer et charger des tableaux numpy de taille arbitraire:

```
import numpy as np

a = np.random.randint(10,size=(3,3))
np.save('arr', a)

a2 = np.load('arr.npy')
print a2
```

Lire Sauvegarde et chargement de tableaux en ligne:
https://riptutorial.com/fr/numpy/topic/10891/sauvegarde-et-chargement-de-tableaux

Chapitre 11: sous-classement ndarray

Syntaxe

- `def __array_prepare__(self, out_arr: ndarray, context: Tuple[ufunc, Tuple, int] = None) -> ndarray: # called on the way into a ufunc`
- `def __array_wrap__(self, out_arr: ndarray, context: Tuple[ufunc, Tuple, int] = None) -> ndarray: # called on the way out of a ufunc`
- `__array_priority__: int # used to determine which argument to invoke the above methods on when a ufunc is called`
- `def __array_finalize__(self, obj: ndarray): # called whenever a new instance of this class comes into existence, even if this happens by routes other than __new__`

Examples

Suivre une propriété supplémentaire sur des tableaux

```
class MySubClass(np.ndarray):
    def __new__(cls, input_array, info=None):
        obj = np.asarray(input_array).view(cls)
        obj.info = info
        return obj

    def __array_finalize__(self, obj):
        # handles MySubClass(...)
        if obj is None:
            pass

        # handles my_subclass[...] or my_subclass.view(MySubClass) or ufunc output
        elif isinstance(obj, MySubClass):
            self.info = obj.info

        # handles my_arr.view(MySubClass)
        else:
            self.info = None

    def __array_prepare__(self, out_arr, context=None):
        # called before a ufunc runs
        if context is not None:
            func, args, which_return_val = context

        return super().__array_prepare__(out_arr, context)

    def __array_wrap__(self, out_arr, context=None):
        # called after a ufunc runs
        if context is not None:
            func, args, which_return_val = context

        return super().__array_wrap__(out_arr, context)
```

Pour le tuple de `context` , `func` est un objet ufunc tel que `np.add` , `args` est un `tuple` , et `which_return_val` est un entier spécifiant la valeur de retour de ufunc en cours de traitement

Lire sous-classement ndarray en ligne: https://riptutorial.com/fr/numpy/topic/6431/sous-classement-ndarray

Chapitre 12: Tableaux

Introduction

Les tableaux ou les `ndarrays` n-dimensions sont l'objet principal de numpy utilisé pour stocker les éléments du même type de données. Ils fournissent une structure de données efficace supérieure aux baies Python ordinaires.

Remarques

Autant que possible, exprimer les opérations sur les données en termes de tableaux et d'opérations vectorielles. Les opérations vectorielles s'exécutent beaucoup plus vite qu'équivalent pour les boucles

Examples

Créer un tableau

Tableau vide

```
np.empty((2,3))
```

Notez que dans ce cas, les valeurs de ce tableau ne sont pas définies. Cette façon de créer un tableau n'est donc utile que si le tableau est rempli plus tard dans le code.

À partir d'une liste

```
np.array([0,1,2,3])
# Out: array([0, 1, 2, 3])
```

Créer une gamme

```
np.arange(4)
# Out: array([0, 1, 2, 3])
```

Créer des zéros

```
np.zeros((3,2))
# Out:
# array([[ 0.,  0.],
#        [ 0.,  0.],
#        [ 0.,  0.]])
```

Créer des objets

```
np.ones((3,2))
# Out:
# array([[ 1.,  1.],
#        [ 1.,  1.],
#        [ 1.,  1.]])
```

Créer des éléments de tableau à espacement linéaire

```
np.linspace(0,1,21)
# Out:
# array([ 0.  ,  0.05,  0.1 ,  0.15,  0.2 ,  0.25,  0.3 ,  0.35,  0.4 ,
#         0.45,  0.5 ,  0.55,  0.6 ,  0.65,  0.7 ,  0.75,  0.8 ,  0.85,
#         0.9 ,  0.95,  1.  ])
```

Créer des éléments de tableau à espacement de journal

```
np.logspace(-2,2,5)
# Out:
# array([  1.00000000e-02,   1.00000000e-01,   1.00000000e+00,
#          1.00000000e+01,   1.00000000e+02])
```

Créer un tableau à partir d'une fonction donnée

```
np.fromfunction(lambda i: i**2, (5,))
# Out:
# array([  0.,   1.,   4.,   9.,  16.])
np.fromfunction(lambda i,j: i**2, (3,3))
# Out:
# array([[ 0.,  0.,  0.],
#        [ 1.,  1.,  1.],
#        [ 4.,  4.,  4.]])
```

Opérateurs de tableau

```
x = np.arange(4)
x
#Out:array([0, 1, 2, 3])
```

l'addition scalaire est élémentaire

```
x+10
#Out: array([10, 11, 12, 13])
```

la multiplication scalaire est élémentaire

```
x*2
#Out: array([0, 2, 4, 6])
```

l'ajout de tableau est élémentaire

```
x+x
```

```
#Out: array([0, 2, 4, 6])
```

la multiplication de tableau est élémentaire

```
x*x
#Out: array([0, 1, 4, 9])
```

le produit scalaire (ou plus généralement la multiplication matricielle) se fait avec une fonction

```
x.dot(x)
#Out: 14
```

Dans Python 3.5, l'opérateur `@` été ajouté en tant qu'opérateur infixe pour la multiplication matricielle

```
x = np.diag(np.arange(4))
print(x)
'''
   Out: array([[0, 0, 0, 0],
   [0, 1, 0, 0],
   [0, 0, 2, 0],
   [0, 0, 0, 3]])
'''
print(x@x)
print(x)
'''
   Out: array([[0, 0, 0, 0],
   [0, 1, 0, 0],
   [0, 0, 4, 0],
   [0, 0, 0, 9]])
'''
```

Ajouter Renvoie une copie avec les valeurs ajoutées. Pas en place.

```
#np.append(array, values_to_append, axis=None)
x = np.array([0,1,2,3,4])
np.append(x, [5,6,7,8,9])
# Out: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
x
# Out: array([0, 1, 2, 3, 4])
y = np.append(x, [5,6,7,8,9])
y
# Out: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

hstack . Pile horizontale (colonne colonne)
vstack . Pile verticale (pile de lignes)

```
# np.hstack(tup), np.vstack(tup)
x = np.array([0,0,0])
y = np.array([1,1,1])
z = np.array([2,2,2])
np.hstack(x,y,z)
# Out: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
np.vstack(x,y,z)
```

```
# Out: array([[0, 0, 0],
#             [1, 1, 1],
#             [2, 2, 2]])
```

Accès aux tableaux

La syntaxe de tranche est `i:j:k` où `i` est l'indice de départ (inclus), `j` est l'index d'arrêt (exclusif) et `k` la taille de pas. Comme d'autres structures de données python, le premier élément a un index de 0:

```
x = np.arange(10)
x[0]
# Out: 0

x[0:4]
# Out: array([0, 1, 2, 3])

x[0:4:2]
# Out:array([0, 2])
```

Les valeurs négatives sont prises en compte à partir de la fin du tableau. `-1` accède donc au dernier élément d'un tableau:

```
x[-1]
# Out: 9
x[-1:0:-1]
# Out: array([9, 8, 7, 6, 5, 4, 3, 2, 1])
```

Les tableaux multidimensionnels sont accessibles en spécifiant chaque dimension séparée par des virgules. Toutes les règles précédentes s'appliquent.

```
x = np.arange(16).reshape((4,4))
x
# Out:
#     array([[ 0,  1,  2,  3],
#            [ 4,  5,  6,  7],
#            [ 8,  9, 10, 11],
#            [12, 13, 14, 15]])

x[1,1]
# Out: 5

x[0:3,0]
# Out: array([0, 4, 8])

x[0:3, 0:3]
# Out:
#     array([[ 0,  1,  2],
#            [ 4,  5,  6],
#            [ 8,  9, 10]])

x[0:3:2, 0:3:2]
# Out:
#     array([[ 0,  2],
#            [ 8, 10]])
```

Transposer un tableau

```
arr = np.arange(10).reshape(2, 5)
```

En `.transpose` méthode `.transpose` :

```
arr.transpose()
# Out:
#      array([[0, 5],
#             [1, 6],
#             [2, 7],
#             [3, 8],
#             [4, 9]])
```

`.T` méthode:

```
arr.T
# Out:
#     array([[0, 5],
#             [1, 6],
#             [2, 7],
#             [3, 8],
#             [4, 9]])
```

Ou `np.transpose` :

```
np.transpose(arr)
# Out:
#     array([[0, 5],
#             [1, 6],
#             [2, 7],
#             [3, 8],
#             [4, 9]])
```

Dans le cas d'un tableau à deux dimensions, cela équivaut à une transposition de matrice standard (comme décrit ci-dessus). Dans le cas de la dimension n, vous pouvez spécifier une permutation des axes du tableau. Par défaut, ceci inverse le `array.shape` .

```
a = np.arange(12).reshape((3,2,2))
a.transpose() # equivalent to a.transpose(2,1,0)
# Out:
#   array([[[ 0,  4,  8],
#           [ 2,  6, 10]],
#
#          [[ 1,  5,  9],
#           [ 3,  7, 11]]])
```

Mais toute permutation des index des axes est possible:

```
a.transpose(2,0,1)
# Out:
#    array([[[ 0,  2],
#            [ 4,  6],
```

```
#            [ 8, 10]],
#
#           [[ 1,  3],
#            [ 5,  7],
#            [ 9, 11]]])

a = np.arange(24).reshape((2,3,4))  # shape (2,3,4)
a.transpose(2,0,1).shape
# Out:
#    (4, 2, 3)
```

Indexation booléenne

```
arr = np.arange(7)
print(arr)
# Out: array([0, 1, 2, 3, 4, 5, 6])
```

La comparaison avec un scalaire renvoie un tableau booléen:

```
arr > 4
# Out: array([False, False, False, False, False,  True,  True], dtype=bool)
```

Ce tableau peut être utilisé dans l'indexation pour sélectionner uniquement les nombres supérieurs à 4:

```
arr[arr>4]
# Out: array([5, 6])
```

L'indexation booléenne peut être utilisée entre différents tableaux (par exemple, des tableaux parallèles associés):

```
# Two related arrays of same length, i.e. parallel arrays
idxs = np.arange(10)
sqrs = idxs**2

# Retrieve elements from one array using a condition on the other
my_sqrs = sqrs[idxs % 2 == 0]
print(my_sqrs)
# Out: array([0, 4, 16, 36, 64])
```

Remodeler un tableau

La `numpy.reshape` (identique à `numpy.ndarray.reshape`) renvoie un tableau de la même taille totale, mais sous une nouvelle forme:

```
print(np.arange(10).reshape((2, 5)))
# [[0 1 2 3 4]
#  [5 6 7 8 9]]
```

Il retourne un nouveau tableau et ne fonctionne pas sur place:

```
a = np.arange(12)
a.reshape((3, 4))
print(a)
# [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Cependant, il *est* possible de remplacer l'attribut de `shape` d'un `ndarray` :

```
a = np.arange(12)
a.shape = (3, 4)
print(a)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]
```

Ce comportement peut être surprenant au premier abord, mais `ndarray` s est stocké dans des blocs de mémoire contigus, et leur `shape` spécifie uniquement comment ce flux de données doit être interprété comme un objet multidimensionnel.

Jusqu'à un axe du tuple de `shape` peut avoir une valeur de `-1` . `numpy` alors la longueur de cet axe pour vous:

```
a = np.arange(12)
print(a.reshape((3, -1)))
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]
```

Ou:

```
a = np.arange(12)
print(a.reshape((3, 2, -1)))

# [[[ 0  1]
#   [ 2  3]]

#  [[ 4  5]
#   [ 6  7]]

#  [[ 8  9]
#   [10 11]]]
```

Plusieurs dimensions non spécifiées, par exemple `a.reshape((3, -1, -1))` ne sont pas autorisées et jettent un `ValueError` .

Opérations de tableau de diffusion

Les opérations arithmétiques sont effectuées élément par élément sur les tableaux Numpy. Pour les tableaux de forme identique, cela signifie que l'opération est exécutée entre des éléments aux indices correspondants.

```
# Create two arrays of the same size
a = np.arange(6).reshape(2, 3)
```

```
b = np.ones(6).reshape(2, 3)

a
# array([0, 1, 2],
#       [3, 4, 5])
b
# array([1, 1, 1],
#       [1, 1, 1])

# a + b: a and b are added elementwise
a + b
# array([1, 2, 3],
#       [4, 5, 6])
```

Les opérations arithmétiques peuvent également être exécutées sur des tableaux de formes différentes au moyen de la *diffusion* Numpy. En général, un tableau est "diffusé" sur l'autre, de sorte que les opérations élémentaires sont effectuées sur des sous-tableaux de forme congruente.

```
# Create arrays of shapes (1, 5) and (13, 1) respectively
a = np.arange(5).reshape(1, 5)
a
# array([[0, 1, 2, 3, 4]])
b = np.arange(4).reshape(4, 1)
b
# array([0],
#       [1],
#       [2],
#       [3])

# When multiplying a * b, slices with the same dimensions are multiplied
# elementwise. In the case of a * b, the one and only row of a is multiplied
# with each scalar down the one and only column of b.
a*b
# array([[ 0,  0,  0,  0,  0],
#        [ 0,  1,  2,  3,  4],
#        [ 0,  2,  4,  6,  8],
#        [ 0,  3,  6,  9, 12]])
```

Pour illustrer cela, considérons la multiplication de tableaux 2D et 3D avec des sous-dimensions congruentes.

```
# Create arrays of shapes (2, 2, 3) and (2, 3) respectively
a = np.arange(12).reshape(2, 2, 3)
a
# array([[[ 0  1  2]
#         [ 3  4  5]]
#
#        [[ 6  7  8]
#         [ 9 10 11]]])
b = np.arange(6).reshape(2, 3)
# array([[0, 1, 2],
#        [3, 4, 5]])

# Executing a*b broadcasts b to each (2, 3) slice of a,
# multiplying elementwise.
a*b
# array([[[ 0,  1,  4],
#         [ 9, 16, 25]],
```

```
#
#        [[ 0,  7, 16],
#         [27, 40, 55]]])

# Executing b*a gives the same result, i.e. the smaller
# array is broadcast over the other.
```

Quand la diffusion en réseau est-elle appliquée?

La diffusion a lieu lorsque deux tableaux ont *des* formes *compatibles* .

Les formes sont comparées au niveau des composants à partir des formes suivantes. Deux dimensions sont compatibles si elles sont identiques ou si l'une d'elles est `1` . Si l'une des formes a une dimension supérieure à l'autre, les composants excédentaires ne sont pas comparés.

Quelques exemples de formes compatibles:

```
(7, 5, 3)    # compatible because dimensions are the same
(7, 5, 3)


(7, 5, 3)    # compatible because second dimension is 1
(7, 1, 3)

(7, 5, 3, 5) # compatible because exceeding dimensions are not compared
      (3, 5)

(3, 4, 5)    # incompatible
   (5, 5)

(3, 4, 5)    # compatible
   (1, 5)
```

Voici la documentation officielle sur la diffusion en réseau .

Remplir un tableau avec le contenu d'un fichier CSV

```
filePath = "file.csv"
data = np.genfromtxt(filePath)
```

De nombreuses options sont prises en charge, voir la documentation officielle pour la liste complète:

```
data = np.genfromtxt(filePath, dtype='float', delimiter=';', skip_header=1, usecols=(0,1,3) )
```

Tableau n-dimensionnel numpy: le ndarray

La structure de données de base de numpy est le `ndarray` (abréviation de tableau à *n* dimensions).

`ndarray` s sont

- homogène (c'est-à-dire qu'ils contiennent des éléments du même type de données)
- contenir des objets de taille fixe (donnés par une *forme* , un tuple de *n* nombres entiers positifs qui spécifient les tailles de chaque dimension)

Tableau à une dimension:

```
x = np.arange(15)
# array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
x.shape
# (15,)
```

Tableau à deux dimensions:

```
x = np.asarray([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14]])
x
# array([[ 0, 1, 2, 3, 4],
# [ 5, 6, 7, 8, 9],
# [10, 11, 12, 13, 14]])
x.shape
# (3, 5)
```

Tridimensionnel:

```
np.arange(12).reshape([2,3,2])
```

Pour initialiser un tableau sans spécifier son contenu, utilisez:

```
x = np.empty([2, 2])
# array([[ 0., 0.],
# [ 0., 0.]])
```

Estimation du type de données et diffusion automatique

Le type de données est défini pour flotter par défaut

```
x = np.empty([2, 2])
# array([[ 0., 0.],
# [ 0., 0.]])

x.dtype
# dtype('float64')
```

Si certaines données sont fournies, numpy devinera le type de données:

```
x = np.asarray([[1, 2], [3, 4]])
x.dtype
# dtype('int32')
```

Notez que lors des affectations, numpy tentera de convertir automatiquement les valeurs en

fonction du `ndarray` de données de `ndarray` .

```
x[1, 1] = 1.5 # assign a float value
x[1, 1]
# 1
# value has been casted to int
x[1, 1] = 'z' # value cannot be casted, resulting in a ValueError
```

Diffusion de tableaux

Voir aussi Diffusion des opérations de tableau .

```
x = np.asarray([[1, 2], [3, 4]])
# array([[1, 2],
#        [3, 4]])
y = np.asarray([[5, 6]])
# array([[5, 6]])
```

En terminologie matricielle, nous aurions une matrice 2x2 et un vecteur 1x2. Nous sommes toujours en mesure de faire une somme

```
# x + y
array([[ 6, 8],
       [ 8, 10]])
```

C'est parce que le tableau `y` est " *étiré* " pour:

```
array([[5, 6],
       [5, 6]])
```

pour s'adapter à la forme de `x` .

Ressources:

- Introduction au ndarray à partir de la documentation officielle: Le tableau N-dimensionnel (ndarray)
- Référence de classe: ndarray .

Lire Tableaux en ligne: https://riptutorial.com/fr/numpy/topic/1296/tableaux

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec numpy	Andras Deak, Chris Mueller, Code-Ninja, Community, Dux, edwinksl, grovduck, Hammer, hashcode55, Joshua Cook, karel, Kersten, Mpaull, prasastoadi, rlee827, sebix, user2314737, Yassie
2	Algèbre linéaire avec np.linalg	Daniel, DataSwede, Fermi paradox, Mahdi, Sean Easter
3	Fichier IO avec numpy	Alex, atomh33ls, Sparkler
4	Filtrage des données	Alex, farleytpm
5	Générer des données aléatoires	amin, ayhan, B8vrede, Dux, Fermi paradox, user2314737
6	Indexation booléenne	Chris Mueller
7	numpy.cross	bob.sacamento, Mad Physicist
8	numpy.dot	bpachev, paddyg, Shubham Dang
9	Régression linéaire simple	Alex
10	Sauvegarde et chargement de tableaux	obachtos
11	sous-classement ndarray	Eric
12	Tableaux	Andras Deak, ayhan, B Samedi, B8vrede, Benjamin, DataSwede, Dux, Gwen, Hamlet, Hammer, KARANJ, Keith L, pixatlazaki, Ryan, Sean Easter, Sparkler, The Hagen, TPVasconcelos, user2314737