EBook Gratuito

APPRENDIMENTO

numpy

Free unaffiliated eBook created from
Stack Overflow contributors.

#numpy

Sommario

Di

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

Capitolo 1: Iniziare con Numpy

Osservazioni

NumPy (pronunciato "numb pie" o talvolta "numb pea") è un'estensione del linguaggio di programmazione Python che aggiunge il supporto per array di grandi dimensioni e multidimensionali, oltre a una vasta libreria di funzioni matematiche di alto livello per operare su questi array.

Versioni

Versione	Data di rilascio
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

Versione	Data di rilascio
1.10.4 *	2016/01/07
1.11.0	2016/05/29

Examples

Installazione su Mac

Il modo più semplice per configurare NumPy su Mac è con pip

```
pip install numpy
```

Installazione usando Conda .
Conda disponibile per Windows, Mac e Linux

1. Installa Conda. Ci sono due modi per installare Conda, sia con Anaconda (pacchetto completo, include numpy) o Miniconda (solo Conda, Python, e i pacchetti da cui dipendono, senza alcun pacchetto aggiuntivo). Sia Anaconda e Miniconda installano lo stesso Conda.
2. Comando aggiuntivo per Miniconda, digitare il comando `conda install numpy`

Installazione su Windows

L'installazione di Numpy tramite pypi (l'indice del pacchetto predefinito usato da pip) generalmente non funziona sui computer Windows. Il modo più semplice per installare su Windows è utilizzando i binari precompilati.

Una fonte per ruote precompilate di molti pacchetti è il sito di Christopher Gohkle . Scegli una versione in base alla tua versione e al tuo sistema Python. Un esempio per Python 3.5 su un sistema a 64 bit:

1. Scarica `numpy-1.11.1+mkl-cp35-cp35m-win_amd64.whl` da qui
2. Apri un terminale Windows (cmd o powershell)
3. Digitare il comando `pip install C:\path_to_download\numpy-1.11.1+mkl-cp35-cp35m-win_amd64.whl`

Se non vuoi scherzare con i singoli pacchetti, puoi usare la distribuzione Winpython che raggruppa la maggior parte dei pacchetti e fornisce un ambiente confinato con cui lavorare. Allo stesso modo, la distribuzione di Anaconda Python viene preinstallata con numpy e numerosi altri pacchetti comuni.

Un'altra fonte popolare è il gestore di pacchetti `conda` , che supporta anche ambienti virtuali .

1. Scarica e installa `conda` .
2. Apri un terminale Windows.
3. Digita il comando `conda install numpy`

Installazione su Linux

NumPy è disponibile nei repository predefiniti delle distribuzioni Linux più diffuse e può essere installato nello stesso modo in cui vengono installati i pacchetti in una distribuzione Linux.

Alcune distribuzioni Linux hanno diversi pacchetti NumPy per Python 2.xe Python 3.x. In Ubuntu e Debian, installa `numpy` a livello di sistema utilizzando il gestore pacchetti APT:

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

Per altre distribuzioni, usa i loro gestori di pacchetti, come zypper (Suse), yum (Fedora) ecc.

`numpy` può anche essere installato con Python's package manager `pip` per Python 2 e con `pip3` per Python 3:

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

`pip` è disponibile nei repository predefiniti delle più diffuse distribuzioni Linux e può essere installato per Python 2 e Python 3 usando:

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

Dopo l'installazione, utilizzare `pip` per Python 2 e `pip3` per Python 3 per utilizzare pip per l'installazione dei pacchetti Python. Si noti tuttavia che potrebbe essere necessario installare molte dipendenze, che sono necessarie per compilare numpy dal sorgente (inclusi pacchetti di sviluppo, compilatori, fortran, ecc.).

Oltre a installare `numpy` a livello di sistema, è anche comune (forse anche altamente consigliato) installare `numpy` in ambienti virtuali usando i popolari pacchetti Python come `virtualenv` . In Ubuntu, `virtualenv` può essere installato utilizzando:

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

Quindi, crea e attiva un virtualenv per Python 2 o Python 3 e poi usa `pip` per installare `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
```

Importazione di base

Importa il modulo numpy per usarne una parte.

```
import numpy as np
```

La maggior parte degli esempi userà `np` come abbreviazione di numpy. Supponiamo che "np" significhi "numpy" negli esempi di codice.

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

Notebook temporaneo Jupyter ospitato da Rackspace

I notebook Jupyter sono un ambiente di sviluppo interattivo basato su browser. Sono stati originariamente sviluppati per eseguire computation python e come tali funzionano molto bene con Numpy. Per provare Numpy in un notebook Jupyter senza installare completamente sul proprio sistema locale, Rackspace fornisce gratuitamente taccuini temporanei su tmpnb.org .

Nota: questo non è un servizio proprietario con alcun tipo di upsell. Jupyter è una tecnologia interamente open source sviluppata da UC Berkeley e Cal Poly San Luis Obispo. Rackspace dona questo servizio come parte del processo di sviluppo.

Per provare `numpy` su tmpnb.org:

1. visita tmpnb.org
2. selezionare `Welcome to Python.ipynb` o
3. Nuovo >> Python 2 o
4. Nuovo >> Python 3

Leggi Iniziare con Numpy online: https://riptutorial.com/it/numpy/topic/823/iniziare-con-numpy

Capitolo 2: Algebra lineare con np.linalg

Osservazioni

A partire dalla versione 1.8, molte delle routine in `np.linalg` possono operare su una "pila" di matrici. Cioè, la routine può calcolare i risultati per più matrici se sono raggruppate insieme. Ad esempio, `A` qui viene interpretato come due matrici 3-per-3 sovrapposte:

```
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])
```

I documenti ufficiali `np` specificano tramite parametri come `a : (..., M, M) array_like` .

Examples

Risolvi i sistemi lineari con np.solve

Considera le seguenti tre equazioni:

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

Possiamo esprimere questo sistema come un'equazione di matrice `A * x = b` con:

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

Quindi, usa `np.linalg.solve` per risolvere `x` :

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

`A` deve essere una matrice quadrata e full-rank: tutte le sue righe devono essere linearmente indipendenti. `A` dovrebbe essere invertibile / non singolare (il suo determinante non è zero). Ad esempio, se una riga di `A` è un multiplo di un'altra, chiamando `linalg.solve` si alza `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])
```

Tali sistemi possono essere risolti con `np.linalg.lstsq` .

Trova la soluzione dei minimi quadrati su un sistema lineare con np.linalg.lstsq

I minimi quadrati rappresentano un approccio standard ai problemi con più equazioni rispetto alle incognite, noti anche come sistemi sovradeterminati.

Considera le quattro equazioni:

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

Possiamo esprimere questo come una moltiplicazione di matrice `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])
```

Quindi risolvi con `np.linalg.lstsq` :

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

`x` è la soluzione, `residuals` la somma, `rank` il rango matrice dell'input `A` ed `s` valori singolari di `A` Se `b` ha più di una dimensione, `lstsq` risolverà il sistema corrispondente a ciascuna colonna di `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` e `s` dipendono solo `A` , e sono quindi lo stesso come sopra.

Leggi Algebra lineare con np.linalg online: https://riptutorial.com/it/numpy/topic/3753/algebra-lineare-con-np-linalg

Capitolo 3: Array

introduzione

Gli array N-dimensionali o `ndarrays` sono gli oggetti principali di numpy utilizzati per memorizzare elementi dello stesso tipo di dati. Forniscono una struttura dati efficiente che è superiore agli ordinari array Python.

Osservazioni

Ogni volta che è possibile esprimere operazioni sui dati in termini di matrici e operazioni vettoriali. Le operazioni vettoriali vengono eseguite molto più velocemente dell'equivalente per i loop

Examples

Crea una matrice

Matrice vuota

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

Si noti che in questo caso, i valori in questo array non sono impostati. Questo modo di creare un array è quindi utile solo se l'array viene riempito più avanti nel codice.

Da una lista

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

Crea un intervallo

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

Crea zeri

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

Crea uno

```
np.ones((3,2))
# Out:
```

```
# array([[ 1.,  1.],
#        [ 1.,  1.],
#        [ 1.,  1.]])
```

Crea elementi di array con spaziatura lineare

```
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.  ])
```

Crea elementi di array con spazi logaritmici

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

Crea una matrice da una determinata funzione

```
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.]])
```

Operatori di array

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

l'aggiunta scalare è elementare

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

la moltiplicazione scalare è elementare

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

l'aggiunta dell'array è un elemento saggio

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

la moltiplicazione degli array è elementare

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

il prodotto punto (o più in generale la moltiplicazione della matrice) viene eseguito con una funzione

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

In Python 3.5, l'operatore `@` stato aggiunto come operatore infisso per la moltiplicazione della matrice

```
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]])
'''
```

Aggiungi Restituisce la copia con i valori aggiunti. NON sul posto.

```
#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 . Stack orizzontale. (pila di colonne)
vstack . Pila verticale. (pila di file)

```
# 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]])
```

Array Access

La sintassi Slice è `i:j:k` dove `i` è l'indice iniziale (incluso), `j` è l'indice di arresto (esclusivo) e `k` è la dimensione del passo. Come altre strutture di dati python, il primo elemento ha un indice di 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])
```

I valori negativi contano dalla fine dell'array. `-1` quindi accede all'ultimo elemento di un array:

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

È possibile accedere agli array multidimensionali specificando ogni dimensione separata da virgole. Si applicano tutte le regole precedenti.

```
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]])
```

Trasposizione di un array

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

Utilizzando il metodo `.transpose` :

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

Metodo `.T` :

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

O `np.transpose` :

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

Nel caso di un array bidimensionale, questo è equivalente a una trasposizione matrice standard (come illustrato sopra). Nel caso n-dimensionale, è possibile specificare una permutazione degli assi dell'array. Per impostazione predefinita, questo inverte `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]]])
```

Ma qualsiasi permutazione degli indici degli assi è possibile:

```
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)
```

Indicizzazione booleana

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

Il confronto con uno scalare restituisce un array booleano:

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

Questo array può essere utilizzato nell'indicizzazione per selezionare solo i numeri maggiori di 4:

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

L'indicizzazione booleana può essere utilizzata tra diversi array (ad es. Array paralleli correlati):

```
# 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])
```

Rimodellare una matrice

Il `numpy.reshape` (come `numpy.ndarray.reshape`) restituisce una matrice della stessa dimensione totale, ma in una nuova forma:

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

Restituisce un nuovo array e non funziona sul posto:

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

Tuttavia, è possibile sovrascrivere l'attributo `shape` di un `ndarray` :

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

`ndarray` questo comportamento potrebbe essere sorprendente, ma i file di `ndarray` sono memorizzati in blocchi contigui di memoria e la loro `shape` specifica solo in che modo questo flusso di dati deve essere interpretato come un oggetto multidimensionale.

Fino a un asse nella tupla di `shape` può avere un valore di `-1` . `numpy` quindi la lunghezza di questo asse per te:

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

O:

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

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

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

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

Molteplici dimensioni non specificate, ad es. `a.reshape((3, -1, -1))` non sono consentite e `a.reshape((3, -1, -1))` `ValueError` .

Trasmissione di operazioni su array

Le operazioni aritmetiche sono eseguite elementwise su array Numpy. Per matrici di forma identica, ciò significa che l'operazione viene eseguita tra gli elementi negli indici corrispondenti.

```
# 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])
```

Le operazioni aritmetiche possono anche essere eseguite su array di forme diverse tramite la *trasmissione* Numpy. In generale, un array viene "trasmesso" sull'altro in modo che le operazioni elementwise vengano eseguite su sottoselezioni di forma 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]])
```

Per illustrare ulteriormente questo, considerare la moltiplicazione di array 2D e 3D con sotto-dimensioni congruenti.

```
# 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.
```

Quando viene applicata la trasmissione di array?

La trasmissione ha luogo quando due array hanno forme *compatibili* .

Le forme vengono confrontate in base al componente a partire da quelle finali. Due dimensioni sono compatibili se sono uguali o uno di loro è `1` . Se una forma ha una dimensione superiore rispetto all'altra, i componenti eccedenti non vengono confrontati.

Alcuni esempi di forme compatibili:

```
(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)
```

Ecco la documentazione ufficiale sulla trasmissione di array .

Compila una matrice con il contenuto di un file CSV

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

Sono supportate molte opzioni, consultare la documentazione ufficiale per l'elenco completo:

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

Array n-dimensionale di Numpy: il narray

La struttura dei dati di base in numpy è `ndarray` (abbreviazione di matrice *n*- dimensionale). `ndarray` s sono

- omogeneo (cioè contengono elementi dello stesso tipo di dati)
- contengono elementi di dimensioni fisse (dati da una *forma* , una tupla di *n* interi positivi che specificano le dimensioni di ciascuna dimensione)

Array monodimensionale:

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

Array bidimensionale:

```
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)
```

Tridimensionale:

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

Per inizializzare un array senza specificarne il contenuto, utilizzare:

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

Identificazione del tipo di dati e casting automatico

Il tipo di dati è impostato su float per impostazione predefinita

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

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

Se vengono forniti alcuni dati, numpy indovina il tipo di dati:

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

Si noti che quando si eseguono assegnazioni numpy tenterà di eseguire automaticamente il cast dei valori per adattarsi al tipo di dati di `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
```

Trasmissione di array

Vedi anche Trasmissione di operazioni su array .

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

Nella terminologia matrix, avremmo un vettore 2x2 e un vettore 1x2. Ancora siamo in grado di fare una somma

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

Questo perché l'array `y` è " *allungato* " a:

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

per adattarsi alla forma di `x` .

risorse:

- Introduzione al narray dalla documentazione ufficiale: l'array N-dimensionale (ndarray)
- Riferimento di classe: ndarray .

Leggi Array online: https://riptutorial.com/it/numpy/topic/1296/array

Capitolo 4: File IO con numpy

Examples

Salvataggio e caricamento di array numpy usando file binari

```
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
```

Caricamento di dati numerici da file di testo con struttura coerente

La funzione `np.loadtxt` può essere utilizzata per leggere file simili a 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.]])
```

Lo stesso file può essere letto usando un'espressione regolare con `np.fromregex` :

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

Salvataggio dei dati come file ASCII in stile CSV

Analogico a `np.loadtxt` , `np.savetxt` può essere utilizzato per salvare i dati in un file ASCII

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

Per controllare la formattazione:

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

```
    header="commented example text")
```

Produzione:

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

Leggere i file CSV

Tre funzioni principali disponibili (descrizione dalle pagine man):

> `fromfile` - Un modo molto efficace di leggere i dati binari con un tipo di dati noto, nonché di analizzare semplicemente file di testo formattati. I dati scritti usando il metodo tofile possono essere letti usando questa funzione.
>
> `genfromtxt` - Carica i dati da un file di testo, con valori mancanti gestiti come specificato. Ogni riga oltre le prime righe di skip_header viene divisa sul carattere delimitatore e i caratteri che seguono il carattere dei commenti vengono scartati.
>
> `loadtxt` : carica i dati da un file di testo. Ogni riga nel file di testo deve avere lo stesso numero di valori.

`genfromtxt` è una funzione wrapper per `loadtxt` . `genfromtxt` è il più diretto da usare in quanto ha molti parametri per gestire il file di input.

Numero coerente di colonne, tipo di dati coerente (numerico o stringa):

Dato un file di input, `myfile.csv` con i contenuti:

```
#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)
```

dà una matrice:

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

Numero coerente di colonne, tipo di dati misti (tra colonne):

```
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')])
```

Notare che l'uso di `dtype=None` risultato un riepilogo.

Numero di colonne incoerente:

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

Nell'array a riga singola:

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

Leggi File IO con numpy online: https://riptutorial.com/it/numpy/topic/4973/file-io-con-numpy

Capitolo 5: Filtraggio dei dati

Examples

Filtraggio dei dati con un array booleano

Quando viene fornito un solo argomento alla funzione di numpy in `where` restituisce gli indici dell'array di input (la `condition`) che viene valutata come true (stesso comportamento di `numpy.nonzero`). Questo può essere usato per estrarre gli indici di una matrice che soddisfano una determinata condizione.

```
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]
```

Se non si ha bisogno degli indici, questo può essere ottenuto in un solo passaggio usando `extract` , dove si specifica la `condition` come primo argomento, ma si fornisce `array` di restituire i valori da dove la condizione è vera come secondo argomento.

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

Due ulteriori argomenti `x` e `y` possono essere forniti per `where` , nel qual caso l'uscita conterrà i valori di `x` in cui la condizione è `True` ei valori di `y` dove la condizione è `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]])
```

Indici filtranti diretti

Per casi semplici, puoi filtrare i dati direttamente.

```
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]
```

Leggi Filtraggio dei dati online: https://riptutorial.com/it/numpy/topic/6187/filtraggio-dei-dati

Capitolo 6: Generazione di dati casuali

introduzione

Il modulo `random` di NumPy fornisce metodi convenienti per generare dati casuali aventi la forma e la distribuzione desiderate.

Ecco la documentazione ufficiale .

Examples

Creazione di un array casuale semplice

```
# 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 ])
```

Impostare il seme

Utilizzando `random.seed` :

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

Creando un oggetto generatore di numeri casuali:

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

Creazione di numeri casuali

```
# 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]])
'''
```

Selezione di un campione casuale da una matrice

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

Seleziona tre lettere a caso (*con la sostituzione* - lo stesso oggetto può essere scelto più volte):

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

Campionamento senza sostituzione:

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

Assegna la probabilità a ogni lettera:

```
# 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')
'''
```

Generazione di numeri casuali tratte da specifiche distribuzioni

Disegna campioni da una distribuzione normale (gaussiana)

```
# 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])
```

Esistono diverse distribuzioni aggiuntive disponibili in `numpy.random` , ad esempio `poisson` , `binomial` e `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])
```

Leggi Generazione di dati casuali online: https://riptutorial.com/it/numpy/topic/2060/generazione-di-dati-casuali

Capitolo 7: Indicizzazione booleana

Examples

Creazione di un array booleano

Un array booleano può essere creato manualmente usando `dtype=bool` durante la creazione dell'array. Valori diversi da `0` , `None` , `False` o stringhe vuote sono considerati 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]
```

In alternativa, numpy crea automaticamente un array booleano quando vengono effettuati confronti tra array e scalari o tra array della stessa forma.

```
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
```

Leggi Indicizzazione booleana online: https://riptutorial.com/it/numpy/topic/6072/indicizzazione-booleana

Capitolo 8: numpy.cross

Sintassi

- `numpy.cross(a, b)` prodotto # croce di *A* e *B* (o vettori in *A* e *B)*
- `numpy.cross(a, b, axisa=-1)` #cross prodotto di vettori in *a* con *b* , i vettori in a sono disposti lungo l'asse *axisa*
- `numpy.cross(a, b, axisa=-1, axisb=-1, axisc=-1)` prodotti # trasversali di vettori in *un* e *b,* vettori di uscita disposte lungo l'asse specificato da *axisc*
- `numpy.cross(a, b, axis=None)` prodotti # trasversali di vettori in *A* e *B,* i vettori in *a, b,* e in uscita disposti lungo *l'asse* asse

Parametri

Colonna	Colonna
a, b	Nell'uso più semplice, `a` e `b` sono due vettori a 2 o 3 elementi. Possono anche essere array di vettori (cioè matrici bidimensionali). Se `a` è un array e 'b' è un vettore, `cross(a,b)` restituisce un array i cui elementi sono i prodotti incrociati di ciascun vettore in `a` con il vettore `b` . La `b` è una matrice e `a` è un singolo vettore, la `cross(a,b)` restituisce una matrice i cui elementi sono i prodotti incrociati di `a` con ciascun vettore in `b` . `a` e `b` possono essere entrambi array se hanno la stessa forma. In questo caso, `cross(a,b)` restituisce `cross(a[0],b[0]), cross(a[1], b[1]), ...`
Axisa / b	Se `a` è un array, può avere vettori disposti attraverso l'asse che varia più rapidamente, l'asse di variazione più lento o qualcosa in mezzo. `axisa` dice a `cross()` come sono disposti i vettori in `a` . Di default, prende il valore dell'asse che varia più lentamente. `axisb` funziona allo stesso modo con l'input `b` . Se l'output di `cross()` sta per essere un array, i vettori di output possono essere disposti su diversi assi dell'array; `axisc` dice `cross` come disporre i vettori nella sua matrice di uscita. Per impostazione predefinita, `axisc` indica l'asse che varia più lentamente.
asse	Un parametro di convenienza che imposta `axisa` , `axisb` e `axisc` tutti sullo stesso valore, se desiderato. Se l' `axis` e uno qualsiasi degli altri parametri sono presenti nella chiamata, il valore `axis` annullerà gli altri valori.

Examples

Prodotto incrociato di due vettori

Numpy fornisce una funzione `cross` per il calcolo di prodotti incrociati vettoriali. Il prodotto incrociato di vettori `[1, 0, 0]` e `[0, 1, 0]` è `[0, 0, 1]` . Numpy ci dice:

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

come previsto.

Mentre i prodotti incrociati sono normalmente definiti solo per vettori tridimensionali. Tuttavia, uno degli argomenti della funzione Numpy può essere costituito da due vettori di elementi. Se il vettore `c` è dato come `[c1, c2]` , Numpy assegna zero alla terza dimensione: `[c1, c2, 0]` . Così,

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

A differenza del `dot` che esiste sia come funzione Numpy sia come metodo `ndarray` , la `cross` esiste solo come funzione autonoma:

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

Più prodotti incrociati con una sola chiamata

Entrambi gli input possono essere una matrice di vettori di elementi 3- (o 2).

```
>>> 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]])
```

Il risultato in questo caso è array ([np.cross (a [0], b), np.cross (a [1], b), np.cross (a [2], b)])

`b` può anche essere una matrice di vettori di elementi 3- (o 2), ma deve avere la stessa forma di `a` . In caso contrario, il calcolo non riesce con un errore di "mancata corrispondenza della forma". Quindi possiamo avere

```
>>> 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]])
```

e ora il risultato è `array([np.cross(a[0],b[0]), np.cross(a[1],b[1]), np.cross(a[2],b[2])])`

Maggiore flessibilità con più prodotti incrociati

Nei nostri ultimi due esempi, numpy presupponeva che `a[0,:]` fosse il primo vettore, `a[1,:]` il

secondo e `a[2,:]` il terzo. Numpy.cross ha un argomento opzionale axisa che ci permette di specificare quale asse definisce i vettori. Così,

```
>>> 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]])
```

Il risultato `axisa=1` e il risultato predefinito sono entrambi `(np.cross([1,1,1],b), np.cross([0,1,0],b), np.cross([1,0,-1],b))` . Per impostazione predefinita, `axisa` indica sempre l'ultimo (più lentamente variabile) asse della matrice. Il risultato `axisa=0` è `(np.cross([1,0,1],b), np.cross([1,1,0],b), np.cross([1,0,-1],b))` .

Un parametro opzionale simile, `axisb` , esegue la stessa funzione per l'ingresso `b` , se è anche un array bidimensionale.

I parametri axisa e axisb indicano numpy come distribuire i dati di input. Un terzo parametro, axisc dice a numpy come distribuire l'output se `a` o `b` è multidimensionale. Usando gli stessi ingressi `a` e `b` come sopra, otteniamo

```
>>> 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]])
```

Quindi `axisc=1` e l' `axisc` default `axisc` entrambi lo stesso risultato, cioè gli elementi di ciascun vettore sono contigui nell'indice di spostamento più veloce dell'array di output. axisc è di default l'ultimo asse dell'array. `axisc=0` distribuisce gli elementi di ciascun vettore attraverso la dimensione variabile più lenta dell'array.

Se vuoi che `axisa` , `axisb` e `axisc` abbiano tutti lo stesso valore, non è necessario impostare tutti e tre i parametri. È possibile impostare un quarto parametro, `axis` , sul valore singolo desiderato e gli altri tre parametri verranno impostati automaticamente. l'asse sovrascrive axisa, axisb o axisc se qualcuno di essi è presente nella chiamata di funzione.

Leggi numpy.cross online: https://riptutorial.com/it/numpy/topic/6166/numpy-cross

Capitolo 9: numpy.dot

Sintassi

- numpy.dot (a, b, out = Nessuno)

Parametri

Nome	Dettagli
un	una matrice numpy
B	una matrice numpy
su	una matrice numpy

Osservazioni

numpy.dot

Restituisce il prodotto punto di aeb. Se a e b sono entrambi scalari o entrambi gli array 1-D, viene restituito uno scalare; altrimenti viene restituito un array. Se viene fornito, viene restituito.

Examples

Moltiplicazione della matrice

La moltiplicazione della matrice può essere eseguita in due modi equivalenti con la funzione punto. Un modo è usare la funzione membro del punto di 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.]])
```

Il secondo modo per fare la moltiplicazione della matrice è con la funzione di libreria numpy.

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

Prodotti punto vettoriale

La funzione punto può anche essere utilizzata per calcolare prodotti a punti vettoriali tra due array numpy monodimensionali.

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

Il parametro out

La funzione numpy dot ha un parametro opzionale out = None. Questo parametro consente di specificare una matrice in cui scrivere il risultato. Questo array deve essere esattamente della stessa forma e tipo della matrice che sarebbe stata restituita, oppure verrà generata un'eccezione.

```
>>> 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.]])
```

Proviamo a cambiare il dtype di risultato in 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)
```

E se proviamo a utilizzare un diverso ordine di memoria sottostante, ad esempio in stile Fortran (quindi le colonne sono contigue anziché righe), si verifica anche un errore.

```
>>> 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)
```

Operazioni con matrici su array di vettori

numpy.dot può essere usato per moltiplicare una lista di vettori per una matrice, ma l'orientamento dei vettori deve essere verticale in modo che una lista di otto vettori a due componenti appaia come due vettori di otto componenti:

```
>>> 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.]])
```

Se l'elenco dei vettori è disposto con i suoi assi al contrario (che è spesso il caso), allora l'array deve essere trasposto prima e dopo l'operazione punto in questo modo:

```
>>> 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.]])
```

Sebbene la funzione punto sia molto veloce, a volte è meglio usare einsum. L'equivalente di quanto sopra sarebbe:

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

Che è un po 'più lento ma consente di moltiplicare un elenco di vertici per un corrispondente elenco di matrici. Questo sarebbe un processo molto contorto usando il punto:

```
>>> 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 può essere usato per trovare il prodotto punto di ogni vettore in una lista con un vettore corrispondente in un'altra lista, questo è abbastanza disordinato e lento rispetto alla moltiplicazione e alla somma elemento-saggio lungo l'ultimo asse. Qualcosa di simile (che richiede una matrice molto più grande per essere calcolata ma per lo più ignorata)

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

Dot prodotto utilizzando la moltiplicazione e la somma di elementi

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

L'utilizzo di einsum potrebbe essere raggiunto con

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

Leggi numpy.dot online: https://riptutorial.com/it/numpy/topic/3198/numpy-dot

Capitolo 10: Regressione lineare semplice

introduzione

Adattamento di una linea (o altra funzione) a un insieme di punti dati.

Examples

Utilizzando np.polyfit

Creiamo un set di dati che poi rientra in una linea retta $ 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
```

Per vedere cosa abbiamo fatto:

```
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()
```

Nota: questo esempio segue la documentazione numpy su https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html abbastanza da vicino.

Utilizzando np.linalg.lstsq

Utilizziamo lo stesso set di dati di Polyfit:

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

Ora, proviamo a trovare una soluzione riducendo al minimo il sistema di equazioni lineari A b = c riducendo al minimo | 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()
```

Nota: questo esempio segue la documentazione numpy su https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html abbastanza da vicino.

Leggi Regressione lineare semplice online: https://riptutorial.com/it/numpy/topic/8808/regressione-lineare-semplice

Capitolo 11: Salvataggio e caricamento di array

introduzione

Gli array Numpy possono essere salvati e caricati in vari modi.

Examples

Utilizzando numpy.save e numpy.load

np.save e np.load forniscono un framework facile da usare per il salvataggio e il caricamento di array di numpy di dimensioni arbitrarie:

```
import numpy as np

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

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

Leggi Salvataggio e caricamento di array online:
https://riptutorial.com/it/numpy/topic/10891/salvataggio-e-caricamento-di-array

Capitolo 12: sottoclasse narray

Sintassi

- `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

Tracciamento di una proprietà aggiuntiva sugli array

```
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)
```

Per il `context` tuple, `func` è un oggetto ufunc come `np.add` , `args` è una `tuple` e `which_return_val` è un intero che specifica quale valore di ritorno di ufunc è in elaborazione

Leggi sottoclasse narray online: https://riptutorial.com/it/numpy/topic/6431/sottoclasse-narray

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Numpy	Andras Deak, Chris Mueller, Code-Ninja, Community, Dux, edwinksl, grovduck, Hammer, hashcode55, Joshua Cook, karel, Kersten, Mpaull, prasastoadi, rlee827, sebix, user2314737, Yassie
2	Algebra lineare con np.linalg	Daniel, DataSwede, Fermi paradox, Mahdi, Sean Easter
3	Array	Andras Deak, ayhan, B Samedi, B8vrede, Benjamin, DataSwede, Dux, Gwen, Hamlet, Hammer, KARANJ, Keith L, pixatlazaki, Ryan, Sean Easter, Sparkler, The Hagen, TPVasconcelos, user2314737
4	File IO con numpy	Alex, atomh33ls, Sparkler
5	Filtraggio dei dati	Alex, farleytpm
6	Generazione di dati casuali	amin, ayhan, B8vrede, Dux, Fermi paradox, user2314737
7	Indicizzazione booleana	Chris Mueller
8	numpy.cross	bob.sacamento, Mad Physicist
9	numpy.dot	bpachev, paddyg, Shubham Dang
10	Regressione lineare semplice	Alex
11	Salvataggio e caricamento di array	obachtos
12	sottoclasse narray	Eric