Kostenloses eBook

LERNEN

numpy

Free unaffiliated eBook created from
Stack Overflow contributors.

#numpy

Inhaltsverzeichnis

Über ... 1

Kapitel 1: Erste Schritte mit Numpy ... 2

Bemerkungen ... 2

Versionen ... 2

Examples ... 3

Installation unter Mac ... 3

Installation unter Windows ... 3

Installation unter Linux ... 4

Grundlegender Import ... 5

Temporäres Jupyter Notebook von Rackspace gehostet ... 5

Kapitel 2: Arrays ... 6

Einführung ... 6

Bemerkungen ... 6

Examples ... 6

Erstellen Sie ein Array ... 6

Array-Operatoren ... 7

Array-Zugriff ... 9

Transponieren eines Arrays ... 10

Boolesche Indizierung ... 11

Umformen eines Arrays ... 11

Broadcasting-Array-Operationen ... 12

Wann wird Array Broadcasting angewendet? ... 14

Füllen Sie ein Array mit dem Inhalt einer CSV-Datei ... 14

Numpy n-dimensionales Array: das Narray ... 14

Kapitel 3: Boolesche Indizierung ... 17

Examples ... 17

Erstellen eines booleschen Arrays ... 17

Kapitel 4: Datei-IO mit numpy ... 18

Examples ... 18

Speichern und Laden von numpy-Arrays mithilfe von Binärdateien ... 18

Laden numerischer Daten aus Textdateien mit konsistenter Struktur ... 18
Speichern von Daten als CSV-ASCII-Datei ... 18
CSV-Dateien lesen ... 19
Kapitel 5: Daten filtern ... 21
Examples ... 21
Filtern von Daten mit einem booleschen Array ... 21
Direktes Filtern von Indizes ... 21
Kapitel 6: Einfache lineare Regression ... 23
Einführung ... 23
Examples ... 23
Verwenden von np.polyfit ... 23
Np.linalg.lstsq verwenden ... 23
Kapitel 7: Lineare Algebra mit np.linalg ... 25
Bemerkungen ... 25
Examples ... 25
Lösen Sie lineare Systeme mit np.solve ... 25
Finden Sie mit np.linalg.lstsq die Lösung der kleinsten Quadrate für ein lineares System ... 26
Kapitel 8: numpy.cross ... 28
Syntax ... 28
Parameter ... 28
Examples ... 28
Kreuzprodukt zweier Vektoren ... 28
Mehrere Kreuzprodukte mit einem Anruf ... 29
Mehr Flexibilität mit mehreren Cross-Produkten ... 29
Kapitel 9: numpy.dot ... 32
Syntax ... 32
Parameter ... 32
Bemerkungen ... 32
Examples ... 32
Matrix-Multiplikation ... 32
Vektor-Punktprodukte ... 33
Der Out-Parameter ... 33

Matrixoperationen auf Arrays von Vektoren ... 34
Kapitel 10: Speichern und Laden von Arrays ... 36
Einführung ... 36
Examples ... 36
Verwenden Sie numpy.save und numpy.load ... 36
Kapitel 11: Unterklasse ndarray ... 37
Syntax ... 37
Examples ... 37
Verfolgung einer zusätzlichen Eigenschaft für Arrays ... 37
Kapitel 12: Zufallsdaten generieren ... 39
Einführung ... 39
Examples ... 39
Erstellen eines einfachen zufälligen Arrays ... 39
Samen setzen ... 39
Zufällige ganze Zahlen erstellen ... 39
Auswahl eines Zufallsmusters aus einem Array ... 39
Generieren von Zufallszahlen aus bestimmten Verteilungen ... 40
Credits ... 42

Über

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

Kapitel 1: Erste Schritte mit Numpy

Bemerkungen

NumPy (ausgesprochen "Num-Pie" oder manchmal "Taube-Erbse") ist eine Erweiterung der Programmiersprache Python, die Unterstützung für große, mehrdimensionale Arrays sowie eine umfangreiche Bibliothek mathematischer Funktionen auf hoher Ebene für die Verarbeitung dieser Arrays bietet.

Versionen

Ausführung	Veröffentlichungsdatum
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

Ausführung	Veröffentlichungsdatum
1.10.4 *	2016-01-07
1.11.0	2016-05-29

Examples

Installation unter Mac

Der einfachste Weg, NumPy auf einem Mac einzurichten, ist mit pip

```
pip install numpy
```

Installation mit Conda .
Conda für Windows, Mac und Linux verfügbar

1. Installieren Sie Conda. Es gibt zwei Möglichkeiten, Conda zu installieren, entweder mit Anaconda (Full package, include numpy) oder Miniconda (nur Conda, Python und die Pakete, auf die sie angewiesen sind, ohne zusätzliches Paket). Sowohl Anaconda als auch Miniconda installieren die gleiche Conda.
2. Zusätzlicher Befehl für Miniconda, geben Sie den Befehl `conda install numpy`

Installation unter Windows

Die unkomplizierte Installation über pypi (der von Pip verwendete Standardpaketindex) schlägt auf Windows-Computern im Allgemeinen fehl. Die einfachste Möglichkeit zur Installation unter Windows ist die Verwendung vorkompilierter Binärdateien.

Eine Quelle für vorkompilierte Räder vieler Pakete ist die Website von Christopher Gohkle . Wählen Sie eine Version entsprechend Ihrer Python-Version und Ihrem System. Ein Beispiel für Python 3.5 auf einem 64-Bit-System:

1. Laden Sie `numpy-1.11.1+mkl-cp35-cp35m-win_amd64.whl` von hier `numpy-1.11.1+mkl-cp35-cp35m-win_amd64.whl`
2. Öffnen Sie ein Windows-Terminal (Cmd oder Powershell)
3. `pip install C:\path_to_download\numpy-1.11.1+mkl-cp35-cp35m-win_amd64.whl` den Befehl `pip install C:\path_to_download\numpy-1.11.1+mkl-cp35-cp35m-win_amd64.whl`

Wenn Sie nicht mit einzelnen Paketen herumspielen möchten, können Sie die Winpython-Distribution verwenden, die die meisten Pakete zusammenfasst und eine begrenzte Umgebung für die Zusammenarbeit bietet. Ebenso ist die Anaconda Python-Distribution mit Numpy und zahlreichen anderen gebräuchlichen Paketen vorinstalliert.

Eine weitere beliebte Quelle ist der `conda` Package Manager , der auch virtuelle Umgebungen unterstützt.

1. `conda` herunterladen und installieren.

2. Öffnen Sie ein Windows-Terminal.
3. `conda install numpy` den Befehl `conda install numpy`

Installation unter Linux

NumPy ist in den Standard-Repositorys der meisten gängigen Linux-Distributionen verfügbar und kann auf dieselbe Weise installiert werden, wie dies normalerweise bei Paketen in einer Linux-Distribution der Fall ist.

Einige Linux-Distributionen enthalten unterschiedliche NumPy-Pakete für Python 2.x und Python 3.x. Installieren `numpy` in Ubuntu und Debian `numpy` auf Systemebene mit dem APT-Paketmanager:

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

Verwenden Sie für andere Distributionen ihre Paketmanager, wie zypper (Suse), yum (Fedora) usw.

`numpy` kann auch mit Pythons Paketmanager `pip` für Python 2 und mit `pip3` für Python 3 `pip3` :

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

`pip` ist in den Standard-Repositorys der meisten gängigen Linux-Distributionen verfügbar und kann für Python 2 und Python 3 folgendermaßen installiert werden:

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

Verwenden Sie nach der Installation `pip` für Python 2 und `pip3` für Python 3, um pip für die Installation von Python-Paketen zu verwenden. Beachten Sie jedoch, dass Sie möglicherweise viele Abhängigkeiten installieren müssen, die zum Erstellen von numpy aus der Quelle erforderlich sind (einschließlich Entwicklungspaketen, Compilern, Fortran usw.).

Neben der Installation von `numpy` auf Systemebene ist es auch üblich (vielleicht sogar sehr empfehlenswert), `numpy` in virtuellen Umgebungen mit gängigen Python-Paketen wie `virtualenv` . In Ubuntu kann `virtualenv` installiert werden mit:

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

Dann erstellen und aktivieren Sie eine Virtualenv für Python 2 oder Python 3 und installieren Sie dann `numpy` mit `pip` :

```
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
```

Grundlegender Import

Importieren Sie das numpy-Modul, um einen beliebigen Teil davon zu verwenden.

```
import numpy as np
```

Die meisten Beispiele verwenden `np` als Kurzform für numpy. Angenommen, "np" bedeutet in Codebeispielen "numpy".

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

Temporäres Jupyter Notebook von Rackspace gehostet

Jupyter Notebooks sind eine interaktive, browserbasierte Entwicklungsumgebung. Sie wurden ursprünglich entwickelt, um Rechenpython auszuführen, und spielen daher sehr gut mit Numpy. Um Numpy in einem Jupyter-Notebook auszuprobieren, ohne eines der beiden Systeme auf einem lokalen System vollständig zu installieren, bietet Rackspace kostenlose temporäre Notebooks auf tmpnb.org .

Hinweis: Dies ist kein proprietärer Service mit Upsells. Jupyter ist eine vollständig offene Technologie, die von UC Berkeley und Cal Poly San Luis Obispo entwickelt wurde. Rackspace spendet diesen Service als Teil des Entwicklungsprozesses.

So `numpy` Sie `numpy` bei tmpnb.org:

1. Besuchen Sie tmpnb.org
2. `Welcome to Python.ipynb` entweder `Welcome to Python.ipynb` oder
3. Neu >> Python 2 oder
4. Neu >> Python 3

Erste Schritte mit Numpy online lesen: https://riptutorial.com/de/numpy/topic/823/erste-schritte-mit-numpy

Kapitel 2: Arrays

Einführung

N-dimensionalen Arrays oder `ndarrays` sind numpy Kernobjekt verwendet für Elemente desselben Datentyps speichert. Sie bieten eine effiziente Datenstruktur, die herkömmlichen Arrays von Python überlegen ist.

Bemerkungen

Wann immer möglich, Operationen mit Daten in Form von Arrays und Vektoroperationen ausdrücken. Vektoroperationen werden für Schleifen viel schneller als die entsprechenden ausgeführt

Examples

Erstellen Sie ein Array

Leeres Array

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

Beachten Sie, dass in diesem Fall die Werte in diesem Array nicht festgelegt werden. Diese Art der Array-Erstellung ist daher nur sinnvoll, wenn das Array später im Code gefüllt wird.

Aus einer Liste

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

Erstellen Sie einen Bereich

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

Erstellen Sie Nullen

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

Erstellen Sie eine

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

Erstellen Sie linienförmige Array-Elemente

```
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.  ])
```

Erstellen Sie Array-Elemente mit Protokollabstand

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

Erstellen Sie ein Array aus einer bestimmten Funktion

```
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.]])
```

Array-Operatoren

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

Skalaraddition ist elementweise

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

Skalarmultiplikation ist elementweise

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

Array-Addition ist elementweise

```
x+x
```

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

Array-Multiplikation ist elementweise

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

Punktprodukt (oder allgemeiner Matrixmultiplikation) erfolgt mit einer Funktion

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

In Python 3.5 wurde der Operator `@` als Infix-Operator für die Matrixmultiplikation hinzugefügt

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

Anhängen Gibt eine Kopie mit angehängten Werten zurück. Nicht an Ort und Stelle.

```
#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 Horizontaler Stapel (Spaltenstapel)
Vstack Vertikaler Stapel. (Reihenstapel)

```
# 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-Zugriff

Die Slice-Syntax lautet `i:j:k` wobei `i` der Startindex (einschließlich) ist, `j` der Stoppindex (exklusiv) und `k` die Schrittgröße ist. Wie andere Python-Datenstrukturen hat das erste Element den Index 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])
```

Negative Werte zählen vom Ende des Arrays. `-1` greift daher auf das letzte Element in einem Array zu:

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

Auf mehrdimensionale Arrays kann zugegriffen werden, indem jede durch Kommas getrennte Dimension angegeben wird. Alle vorherigen Regeln gelten.

```
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]])
```

Transponieren eines Arrays

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

`.transpose` Methode verwenden:

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

`.T` Methode:

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

Oder `np.transpose` :

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

Im Falle eines 2-dimensionalen Arrays entspricht dies einer Standardmatrixtransponierung (wie oben dargestellt). Im n-dimensionalen Fall können Sie eine Permutation der Array-Achsen angeben. Standardmäßig werden `array.shape` umgekehrt:

```
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]]])
```

Es ist jedoch jede Permutation der Achsindizes möglich:

```
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)
```

Boolesche Indizierung

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

Ein Vergleich mit einem Skalar gibt ein boolesches Array zurück:

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

Dieses Array kann bei der Indizierung verwendet werden, um nur die Zahlen größer als 4 auszuwählen:

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

Die boolesche Indizierung kann zwischen verschiedenen Arrays (z. B. verwandten parallelen Arrays) verwendet werden:

```
# 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])
```

Umformen eines Arrays

Die Methode `numpy.reshape` (identisch mit `numpy.ndarray.reshape`) gibt ein Array derselben Gesamtgröße zurück, aber in einer neuen Form:

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

Es gibt ein neues Array zurück und funktioniert nicht an Ort und Stelle:

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

Es *ist jedoch* möglich, das `shape` Attribut eines `ndarray` zu überschreiben:

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

Dieses Verhalten mag zunächst überraschend sein, aber `ndarray` werden in zusammenhängenden Speicherblöcken gespeichert und ihre `shape` nur an, wie dieser Datenstrom als mehrdimensionales Objekt interpretiert werden soll.

Bis zu einer Achse in der `shape` Tupel kann einen Wert von `-1` . `numpy` dann die Länge dieser Achse für Sie:

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

Oder:

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

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

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

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

Mehrere nicht angegebene Dimensionen, z. B. `a.reshape((3, -1, -1))` sind nicht zulässig und werfen einen `ValueError` .

Broadcasting-Array-Operationen

Arithmetische Operationen werden elementweise auf Numpy-Arrays durchgeführt. Für Arrays mit identischer Form bedeutet dies, dass die Operation zwischen Elementen in entsprechenden Indizes ausgeführt wird.

```
# 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])
```

Arithmetische Operationen können auch auf Arrays verschiedener Formen mittels unkomplizierter *Übertragung ausgeführt werden* . Im Allgemeinen wird ein Array über das andere "Broadcast", so dass elementweise Operationen an Teilarrays mit kongruenter Form ausgeführt werden.

```
# 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]])
```

Um dies weiter zu veranschaulichen, betrachten Sie die Multiplikation von 2D- und 3D-Arrays mit kongruenten Unterdimensionen.

```
# 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.
```

Wann wird Array Broadcasting angewendet?

Die Übertragung findet statt, wenn zwei Arrays *kompatible* Formen haben.

Formen werden komponentenweise ausgehend von den nachgestellten verglichen. Zwei Dimensionen sind kompatibel, wenn sie gleich sind oder eine davon `1` . Wenn eine Form eine höhere Dimension als die andere hat, werden die übersteigenden Komponenten nicht verglichen.

Einige Beispiele kompatibler Formen:

```
(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)
```

Hier ist die offizielle Dokumentation zum Array-Broadcasting .

Füllen Sie ein Array mit dem Inhalt einer CSV-Datei

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

Viele Optionen werden unterstützt. Die vollständige Liste finden Sie in der offiziellen Dokumentation :

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

Numpy n-dimensionales Array: das Narray

Die Kerndatenstruktur in numpy ist das `ndarray` (kurz für *n*- dimensionales Array). `ndarray` s sind

- homogen (dh sie enthalten Elemente des gleichen Datentyps)

- enthalten Elemente fester Größen (gegeben durch eine *Form* , ein Tupel von *n* positiven ganzen Zahlen, die die Größe jeder Dimension angeben)

Eindimensionale Anordnung:

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

Zweidimensionale Anordnung:

```
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)
```

Dreidimensional:

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

So initialisieren Sie ein Array ohne Angabe des Inhalts use:

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

Datentyptypen und automatisches Casting

Der Datentyp ist standardmäßig auf Float gesetzt

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

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

Wenn einige Daten bereitgestellt werden, wird der Datentyp von numpy erraten:

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

Beachten Sie, dass numpy bei Zuweisungen versucht, die Werte automatisch an den Datentyp des `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
```

Array-Übertragung

Siehe auch Broadcasting-Array-Operationen .

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

In der Matrixterminologie hätten wir eine 2x2-Matrix und einen 1x2-Zeilenvektor. Trotzdem können wir eine Summe machen

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

Dies liegt daran, dass das Array `y` " *gestreckt* " ist:

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

an die Form von `x` .

Ressourcen:

- Einführung in das ndarray aus der offiziellen Dokumentation: Das N-dimensionale Array (ndarray)
- Klassenreferenz: ndarray .

Arrays online lesen: https://riptutorial.com/de/numpy/topic/1296/arrays

Kapitel 3: Boolesche Indizierung

Examples

Erstellen eines booleschen Arrays

Ein boolesches Array kann manuell erstellt werden, indem beim Erstellen des Arrays `dtype=bool` . Andere Werte als `0` , `None` , `False` oder leere Zeichenfolgen werden als wahr betrachtet.

```
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]
```

Alternativ erstellt numpy automatisch ein boolesches Array, wenn Vergleiche zwischen Arrays und Skalaren oder Arrays derselben Form durchgeführt werden.

```
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
```

Boolesche Indizierung online lesen: https://riptutorial.com/de/numpy/topic/6072/boolesche-indizierung

Kapitel 4: Datei-IO mit numpy

Examples

Speichern und Laden von numpy-Arrays mithilfe von Binärdateien

```
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
```

Laden numerischer Daten aus Textdateien mit konsistenter Struktur

Mit der Funktion `np.loadtxt` können csv-ähnliche Dateien gelesen werden:

```
# 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.]])
```

`np.fromregex` Datei kann mit einem regulären Ausdruck mit `np.fromregex` gelesen werden:

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

Speichern von Daten als CSV-ASCII-Datei

Analog zu `np.loadtxt` können mit `np.savetxt` Daten in einer ASCII-Datei gespeichert werden

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

So steuern Sie die Formatierung:

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

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

Ausgabe:

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

CSV-Dateien lesen

Drei Hauptfunktionen verfügbar (Beschreibung von Manpages):

> `fromfile` - Eine hocheffiziente Methode zum Lesen von binären Daten mit einem bekannten Datentyp sowie zum Analysieren einfach formatierter Textdateien. Mit der Tofile-Methode geschriebene Daten können mit dieser Funktion gelesen werden.
>
> `genfromtxt` - `genfromtxt` Daten aus einer Textdatei, wobei fehlende Werte wie angegeben behandelt werden. Jede Zeile nach den ersten skip_header-Zeilen wird am Trennzeichen aufgeteilt, und Zeichen, die auf das Kommentarzeichen folgen, werden verworfen.
>
> `loadtxt` - `loadtxt` Daten aus einer Textdatei. Jede Zeile in der Textdatei muss dieselbe Anzahl von Werten haben.

`genfromtxt` ist eine Wrapper-Funktion für `loadtxt` . `genfromtxt` ist am einfachsten zu verwenden, da es viele Parameter für den Umgang mit der Eingabedatei gibt.

Konsistente Spaltenanzahl, konsistenter Datentyp (numerisch oder String):

Gegeben eine Eingabedatei, `myfile.csv` mit dem Inhalt:

```
#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)
```

gibt ein Array:

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

Konsistente Spaltenanzahl, gemischter Datentyp (spaltenübergreifend):

```
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')])
```

Beachten Sie die Verwendung von `dtype=None` führt zu einer Wiederholung.

Inkonsistente Spaltenanzahl:

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

In einzeiliges Array:

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

Datei-IO mit numpy online lesen: https://riptutorial.com/de/numpy/topic/4973/datei-io-mit-numpy

Kapitel 5: Daten filtern

Examples

Filtern von Daten mit einem booleschen Array

Wenn nur ein einziges Argument zu numpy geliefert ist , `where` Funktion es die Indizes der Eingabe Array zurückgibt (der `condition`) , die als wahr (gleiches Verhalten wie bewerten `numpy.nonzero`). Hiermit können die Indizes eines Arrays extrahiert werden, die eine bestimmte Bedingung erfüllen.

```
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]
```

Wenn Sie die Indizes nicht benötigen, können Sie dies mithilfe von `extract` in einem Schritt erreichen. Hier können Sie die `condition` als erstes Argument angeben, dem `array` jedoch die Werte geben, aus denen die Bedingung als zweites Argument wahr ist.

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

Zwei weitere Argumente `x` und `y` können für `where` werden. In diesem Fall enthält die Ausgabe die Werte von `x` wobei die Bedingung `True` und die Werte von `y` denen die Bedingung `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]])
```

Direktes Filtern von Indizes

In einfachen Fällen können Sie Daten direkt filtern.

```
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]
```

Daten filtern online lesen: https://riptutorial.com/de/numpy/topic/6187/daten-filtern

Kapitel 6: Einfache lineare Regression

Einführung

Anpassen einer Linie (oder einer anderen Funktion) an eine Gruppe von Datenpunkten.

Examples

Verwenden von np.polyfit

Wir erstellen ein Dataset, das dann mit einer geraden Linie $ f (x) = mx + c $ zusammenpasst.

```
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
```

Um zu sehen, was wir getan haben:

```
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()
```

Hinweis: Dieses Beispiel folgt der numpy-Dokumentation unter https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html ziemlich genau.

Np.linalg.lstsq verwenden

Wir verwenden den gleichen Datensatz wie bei polyfit:

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

Nun versuchen wir eine Lösung zu finden, indem wir das System der linearen Gleichungen A b = c minimieren, indem wir | cA b | ** 2 minimieren

```
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()
```

Hinweis: Dieses Beispiel folgt der numpy-Dokumentation unter https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html sehr genau.

Einfache lineare Regression online lesen: https://riptutorial.com/de/numpy/topic/8808/einfache-lineare-regression

Kapitel 7: Lineare Algebra mit np.linalg

Bemerkungen

Ab Version 1.8 können einige der Routinen in `np.linalg` mit einem Stapel von Matrizen arbeiten. Das heißt, die Routine kann Ergebnisse für mehrere Matrizen berechnen, wenn sie zusammengestapelt sind. Zum Beispiel wird `A` hier als zwei gestapelte 3-mal-3-Matrizen interpretiert:

```
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])
```

Die offiziellen `np` Dokumente geben dies über Parameterspezifikationen an wie `a : (..., M, M) array_like` .

Examples

Lösen Sie lineare Systeme mit np.solve

Betrachten Sie die folgenden drei Gleichungen:

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

Wir können dieses System als Matrixgleichung `A * x = b` ausdrücken mit:

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

Verwenden Sie dann `np.linalg.solve` , um nach `x` zu lösen:

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

`A` muss eine quadratische und eine vollwertige Matrix sein: Alle Zeilen müssen linear unabhängig sein. `A` sollte invertierbar / nicht singulär sein (seine Determinante ist nicht Null). Wenn zum Beispiel eine Reihe von `A` ein Vielfaches einer anderen ist, wird der Aufruf von `linalg.solve` die

`LinAlgError: Singular matrix` erhöhen

```
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])
```

Solche Systeme können mit `np.linalg.lstsq` gelöst `np.linalg.lstsq` .

Finden Sie mit np.linalg.lstsq die Lösung der kleinsten Quadrate für ein lineares System

Least Squares ist ein Standardansatz für Probleme mit mehr Gleichungen als Unbekannten, die auch als überbestimmte Systeme bezeichnet werden.

Betrachten Sie die vier Gleichungen:

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

Wir können dies als Matrixmultiplikation `A * x = b` ausdrücken:

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

Dann lösen Sie mit `np.linalg.lstsq` :

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

`x` ist die Lösung, die `residuals` die Summe, `rank` der Matrix Rang von Eingabe `A` und `s` die singulären Werte von `A` . Wenn `b` mehr als eine Dimension hat, `lstsq` das System entsprechend jeder Spalte von `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` und `s` hängen nur von `A` und sind daher die gleichen wie oben.

Lineare Algebra mit np.linalg online lesen: https://riptutorial.com/de/numpy/topic/3753/lineare-algebra-mit-np-linalg

Kapitel 8: numpy.cross

Syntax

- `numpy.cross(a, b)` # Kreuzprodukt von *a* und *b* (oder Vektoren in *a* und *b*)
- `numpy.cross(a, b, axisa=-1)` #cross Produkt von Vektoren in *a* mit *b* , st Vektoren in a sind entlang der *Achsenachse angeordnet*
- `numpy.cross(a, b, axisa=-1, axisb=-1, axisc=-1)` # Kreuzprodukte der Vektoren in *a* und *b* , Ausgabevektoren entlang der durch *axisc* angegebenen *Achse*
- `numpy.cross(a, b, axis=None)` # Kreuzprodukte von Vektoren in *a* und *b* , Vektoren in *a* , *b* und in Ausgabe entlang der *Achsenachse*

Parameter

Säule	Säule
a, b	Bei der einfachsten Verwendung sind `a` und `b` zwei 2- oder 3-Element-Vektoren. Sie können auch Arrays von Vektoren sein (dh zweidimensionale Matrizen). Wenn `a` ein Array und 'b' ein Vektor ist, gibt `cross(a,b)` ein Array zurück, dessen Elemente die Kreuzprodukte jedes Vektors in `a` mit dem Vektor `b` . `b` ist ein Array und `a` ist ein einzelner Vektor. `cross(a,b)` gibt ein Array zurück, dessen Elemente die Kreuzprodukte von `a` mit jedem Vektor in `b` . `a` und `b` können beide Arrays sein, wenn sie die gleiche Form haben. In diesem Fall gibt `cross(a,b) cross(a[0],b[0]), cross(a[1], b[1]), ...`
Axisa / b	Wenn `a` ein Array ist, kann es Vektoren haben, die auf der am schnellsten variierenden Achse, der langsamsten variierenden Achse oder etwas dazwischen angeordnet sind. `axisa` teilt `cross()` wie die Vektoren in `a` . Standardmäßig nimmt es den Wert der am langsamsten variierenden Achse. `axisb` funktioniert genauso mit Eingabe `b` . Wenn die Ausgabe von `cross()` ein Array sein wird, können die Ausgabevektoren unterschiedliche Array-Achsen haben; `axisc` teilt `cross` wie die Vektoren in ihrem Ausgabearray angeordnet werden. Standardmäßig gibt `axisc` die am `axisc` veränderliche Achse an.
Achse	Ein komfortabler Parameter, der `axisa` , `axisb` und `axisc` auf den gleichen Wert `axisb` , falls gewünscht. Wenn die `axis` und einer der anderen Parameter in dem Aufruf vorhanden sind, überschreibt der Wert der `axis` die anderen Werte.

Examples

Kreuzprodukt zweier Vektoren

Numpy stellt eine `cross` Kreuzprodukte für die Berechnung. Das Kreuzprodukt der Vektoren `[1, 0, 0]` und `[0, 1, 0]` ist `[0, 0, 1]` . Numpy sagt uns:

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

wie erwartet.

Während Kreuzprodukte normalerweise nur für dreidimensionale Vektoren definiert werden. Jedes der Argumente für die Numpy-Funktion kann jedoch zwei Elementvektoren sein. Wenn der Vektor `c` als `[c1, c2]` , weist Numpy der dritten Dimension Null zu: `[c1, c2, 0]` . So,

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

Im Gegensatz zu `dot` das sowohl als Numpy-Funktion als auch als Methode von `ndarray` existiert, existiert `cross` nur als eigenständige Funktion:

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

Mehrere Kreuzprodukte mit einem Anruf

Bei beiden Eingaben kann es sich um ein Array von 3 (oder 2) Elementvektoren handeln.

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

Das Ergebnis ist in diesem Fall array ([np.cross (a [0], b), np.cross (a [1], b), np.cross (a [2], b)])

`b` kann auch ein Array von 3- (oder 2-) Elementvektoren sein, muss aber die gleiche Form wie `a` . Andernfalls schlägt die Berechnung mit einem "Shape Mismatch" -Fehler fehl. So können wir haben

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

und jetzt ist das Ergebnis `array([np.cross(a[0],b[0]), np.cross(a[1],b[1]), np.cross(a[2],b[2])])`

Mehr Flexibilität mit mehreren Cross-Produkten

In den letzten beiden Beispielen nahm numpy an, dass `a[0,:]` der erste Vektor war, `a[1,:]` der

zweite und `a[2,:]` der dritte. Numpy.cross hat ein optionales Argument axisa, mit dem wir festlegen können, welche Achse die Vektoren definiert. So,

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

Das Ergebnis `axisa=1` und das Standardergebnis sind beide `(np.cross([1,1,1],b), np.cross([0,1,0],b), np.cross([1,0,-1],b))` . Standardmäßig gibt `axisa` immer die letzte (am `axisa` veränderliche) Achse des Arrays an. Das Ergebnis von `axisa=0` ist `(np.cross([1,0,1],b), np.cross([1,1,0],b), np.cross([1,0,-1],b))` .

Ein ähnlicher optionaler Parameter, `axisb` , führt dieselbe Funktion für die Eingabe `b` , wenn es sich ebenfalls um ein zweidimensionales Array handelt.

Die Parameter axisa und axisb geben an, wie die Eingabedaten verteilt werden sollen. Ein dritter Parameter, axisc, gibt an, wie die Ausgabe verteilt werden soll, wenn `a` oder `b` mehrdimensional sind. Mit den gleichen Eingaben `a` und `b` wie oben erhalten wir

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

`axisc=1` und der Standard- `axisc` ergeben also dasselbe Ergebnis, `axisc` die Elemente jedes Vektors sind im schnellsten Index des Ausgabearrays zusammenhängend. axisc ist standardmäßig die letzte Achse des Arrays. `axisc=0` verteilt die Elemente jedes Vektors über die am langsamsten variierende Dimension des Arrays.

Wenn Sie möchten, dass `axisa` , `axisb` und `axisc` den gleichen Wert haben, müssen Sie nicht alle drei Parameter einstellen. Sie können einen vierten Parameter, `axis` , auf den erforderlichen Einzelwert setzen, und die anderen drei Parameter werden automatisch eingestellt. axes überschreibt axisa, axisb oder axisc, wenn einer von ihnen im Funktionsaufruf vorhanden ist.

numpy.cross online lesen: https://riptutorial.com/de/numpy/topic/6166/numpy-cross

Kapitel 9: numpy.dot

Syntax

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

Parameter

Name	Einzelheiten
ein	ein numpy-Array
b	ein numpy-Array
aus	ein numpy-Array

Bemerkungen

numpy.dot

Gibt das Punktprodukt von a und b zurück. Wenn a und b beide Skalare oder beide 1-D-Arrays sind, wird ein Skalar zurückgegeben. Andernfalls wird ein Array zurückgegeben. Wenn out angegeben ist, wird es zurückgegeben.

Examples

Matrix-Multiplikation

Die Matrixmultiplikation kann mit der Punktfunktion auf zwei gleichwertige Arten erfolgen. Eine Möglichkeit besteht darin, die Punktelementfunktion von numpy.ndarray zu verwenden.

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

Die zweite Möglichkeit, eine Matrixmultiplikation durchzuführen, ist die Bibliotheksfunktion numpy.

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

Vektor-Punktprodukte

Mit der Punktfunktion können auch Vektorpunktprodukte zwischen zwei eindimensionalen numpy-Arrays berechnet werden.

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

Der Out-Parameter

Die numpy-Punktfunktion hat einen optionalen Parameter out = None. Mit diesem Parameter können Sie ein Array angeben, in das das Ergebnis geschrieben werden soll. Dieses Array muss genau dieselbe Form und denselben Typ haben wie das Array, das zurückgegeben worden wäre. Andernfalls wird eine Ausnahme ausgelöst.

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

Versuchen wir, den dtype des Ergebnisses in int zu ändern.

```
>>> 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)
```

Wenn wir versuchen, eine andere zugrunde liegende Speicherreihenfolge zu verwenden, beispielsweise Fortran-Stil (Spalten sind also zusammenhängend statt Zeilen), führt dies ebenfalls zu einem Fehler.

```
>>> 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)
```

Matrixoperationen auf Arrays von Vektoren

numpy.dot kann verwendet werden, um eine Liste von Vektoren mit einer Matrix zu multiplizieren. Die Ausrichtung der Vektoren muss jedoch vertikal sein, damit eine Liste von acht zwei Komponentenvektoren wie zwei acht Komponentenvektoren erscheint:

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

Wenn die Liste der Vektoren mit ihren Achsen umgekehrt angeordnet ist (was häufig der Fall ist), muss das Array vor und dann nach der Punktoperation wie folgt verschoben werden:

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

Obwohl die Punktfunktion sehr schnell ist, ist es manchmal besser, einsum zu verwenden. Das Äquivalent des Vorstehenden wäre:

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

Dies ist etwas langsamer, ermöglicht jedoch die Multiplikation einer Liste von Scheitelpunkten mit einer entsprechenden Liste von Matrizen. Dies wäre ein sehr komplizierter Prozess mit Punkt:

```
>>> 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 kann verwendet werden, um das Punktprodukt jedes Vektors in einer Liste mit einem entsprechenden Vektor in einer anderen Liste zu finden. Dies ist ziemlich chaotisch und langsam, verglichen mit der elementweisen Multiplikation und dem Summieren entlang der letzten Achse. So etwas (das erfordert, dass ein viel größeres Array berechnet wird, aber meist ignoriert wird)

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

Produkt mit elementweiser Multiplikation und Summierung punktieren

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

Mit einsum konnte man mit erreichen

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

numpy.dot online lesen: https://riptutorial.com/de/numpy/topic/3198/numpy-dot

Kapitel 10: Speichern und Laden von Arrays

Einführung

Ungleichmäßige Arrays können auf verschiedene Arten gespeichert und geladen werden.

Examples

Verwenden Sie numpy.save und numpy.load

np.save und np.load bieten ein benutzerfreundliches Framework zum Speichern und Laden von beliebig großen numpy-Arrays:

```
import numpy as np

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

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

Speichern und Laden von Arrays online lesen:
https://riptutorial.com/de/numpy/topic/10891/speichern-und-laden-von-arrays

Kapitel 11: Unterklasse ndarray

Syntax

- `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

Verfolgung einer zusätzlichen Eigenschaft für Arrays

```
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)
```

Für den `context` Tupels, `func` ist ein wie Objekt ufunc `np.add` , `args` ist ein `tuple` , und `which_return_val` eine ganze Zahl angibt , welchen Wert des Rück ufunc verarbeitet wird

Unterklasse ndarray online lesen: https://riptutorial.com/de/numpy/topic/6431/unterklasse-ndarray

Kapitel 12: Zufallsdaten generieren

Einführung

Das `random` von NumPy stellt praktische Verfahren zum Erzeugen von Zufallsdaten mit der gewünschten Form und Verteilung bereit.

Hier ist die offizielle Dokumentation .

Examples

Erstellen eines einfachen zufälligen Arrays

```
# 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 ])
```

Samen setzen

`random.seed` :

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

Erstellen Sie ein Zufallszahlengeneratorobjekt:

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

Zufällige ganze Zahlen erstellen

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

Auswahl eines Zufallsmusters aus einem Array

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

Wählen Sie zufällig drei Buchstaben aus (*bei Ersetzung* - dasselbe Element kann mehrmals ausgewählt werden):

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

Probenahme ohne Ersatz:

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

Weisen Sie jedem Buchstaben eine Wahrscheinlichkeit zu:

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

Generieren von Zufallszahlen aus bestimmten Verteilungen

Zeichnen Sie Proben aus einer normalen (Gaußschen) Verteilung

```
# 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])
```

In `numpy.random` sind mehrere zusätzliche Distributionen verfügbar, zum Beispiel `poisson` , `binomial` und `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])
```

Zufallsdaten generieren online lesen: https://riptutorial.com/de/numpy/topic/2060/zufallsdaten-generieren

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit Numpy	Andras Deak, Chris Mueller, Code-Ninja, Community, Dux, edwinksl, grovduck, Hammer, hashcode55, Joshua Cook, karel, Kersten, Mpaull, prasastoadi, rlee827, sebix, user2314737, Yassie
2	Arrays	Andras Deak, ayhan, B Samedi, B8vrede, Benjamin, DataSwede, Dux, Gwen, Hamlet, Hammer, KARANJ, Keith L, pixatlazaki, Ryan, Sean Easter, Sparkler, The Hagen, TPVasconcelos, user2314737
3	Boolesche Indizierung	Chris Mueller
4	Datei-IO mit numpy	Alex, atomh33ls, Sparkler
5	Daten filtern	Alex, farleytpm
6	Einfache lineare Regression	Alex
7	Lineare Algebra mit np.linalg	Daniel, DataSwede, Fermi paradox, Mahdi, Sean Easter
8	numpy.cross	bob.sacamento, Mad Physicist
9	numpy.dot	bpachev, paddyg, Shubham Dang
10	Speichern und Laden von Arrays	obachtos
11	Unterklasse ndarray	Eric
12	Zufallsdaten generieren	amin, ayhan, B8vrede, Dux, Fermi paradox, user2314737