Бесплатная электронная книга

УЧУСЬ

numpy

Free unaffiliated eBook created from
Stack Overflow contributors.

#numpy

Около

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

глава 1: Начало работы с numpy

замечания

NumPy (произносится как «numb pie» или иногда «numb pea») является расширением языка программирования Python, который добавляет поддержку больших многомерных массивов, а также обширную библиотеку высокоуровневых математических функций для работы с этими массивами.

Версии

Версия	Дата выхода
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

Версия	Дата выхода
1.10.2	2015-12-14
1.10.4 *	2016-01-07
1.11.0	2016-05-29

Examples

Установка на Mac

Самый простой способ настроить NumPy на Mac - с помощью pip

```
pip install numpy
```

Установка с использованием Conda .
Conda доступен для Windows, Mac и Linux

1. Установите Conda. Существует два способа установки Conda: либо Anaconda (полный пакет, включая numpy), либо Miniconda (только Conda, Python и пакеты, от которых они зависят, без какого-либо дополнительного пакета). Как Anaconda, так и Miniconda устанавливают тот же Conda.
2. Дополнительная команда для Miniconda, введите команду `conda install numpy`

Установка в Windows

Установка Numpy через pypi (индекс пакета по умолчанию, используемый пипсом) обычно терпит неудачу на компьютерах Windows. Самый простой способ установки в Windows - использование предварительно скомпилированных двоичных файлов.

Одним из источников для предварительно скомпилированных колес многих пакетов является сайт Кристофера Гокле . Выберите версию в соответствии с вашей версией и системой Python. Пример для Python 3.5 в 64-битной системе:

1. Скачайте `numpy-1.11.1+mkl-cp35-cp35m-win_amd64.whl` отсюда
2. Откройте терминал Windows (cmd или powershell)
3. Введите команду `pip install C:\path_to_download\numpy-1.11.1+mkl-cp35-cp35m-win_amd64.whl`

Если вы не хотите вмешиваться в отдельные пакеты, вы можете использовать дистрибутив Winpython, который объединяет большинство пакетов и предоставляет ограниченную среду для работы. Аналогичным образом, распределение Anaconda Python поставляется с предустановленным количеством и множеством других распространенных пакетов.

Другим популярным источником является менеджер пакетов `conda` , который также

поддерживает виртуальные среды .

1. Загрузите и установите `conda` .
2. Откройте Windows-терминал.
3. Введите команду `conda install numpy`

Установка в Linux

NumPy доступен в репозиториях по умолчанию для большинства популярных дистрибутивов Linux и может быть установлен так же, как обычно устанавливаются пакеты в дистрибутиве Linux.

В некоторых дистрибутивах Linux есть разные пакеты NumPy для Python 2.x и Python 3.x. В Ubuntu и Debian установите `numpy` на системном уровне, используя диспетчер пакетов APT:

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

Для других дистрибутивов используйте своих менеджеров пакетов, таких как zypper (Suse), yum (Fedora) и т. Д.

`numpy` также может быть установлен с помощью диспетчера пакетов в Python `pip` для Python 2 и с `pip3` для Python 3:

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

`pip` доступен в репозиториях по умолчанию для большинства популярных дистрибутивов Linux и может быть установлен для Python 2 и Python 3, используя:

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

После установки используйте `pip` для Python 2 и `pip3` для Python 3, чтобы использовать pip для установки пакетов Python. Но обратите внимание, что вам может потребоваться установить множество зависимостей, которые необходимы для создания numpy из исходного кода (включая пакеты разработки, компиляторы, fortran и т. Д.).

Помимо установки `numpy` на системном уровне, также широко распространено (возможно, даже очень рекомендуется) устанавливать `numpy` в виртуальных средах с использованием популярных пакетов Python, таких как `virtualenv` . В Ubuntu `virtualenv` можно установить, используя:

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

Затем создайте и активируйте virtualenv для Python 2 или Python 3, а затем используйте `pip`

для установки `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
```

Основной импорт

Импортируйте модуль numpy для использования любой его части.

```
import numpy as np
```

Большинство примеров будут использовать `np` виде сокращения для numpy. Предположим, что «np» означает «numpy» в примерах кода.

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

Временный ноутбук Jupyter от Rackspace

Jupyter Notebooks - это интерактивная среда разработки на основе браузера. Первоначально они были разработаны для запуска вычисления python и, как результат, очень хорошо сочетались с numpy. Чтобы попробовать numpy в ноутбуке Jupyter без полной установки на локальную систему Rackspace предоставляет бесплатные временные ноутбуки на tmpnb.org .

Обратите внимание: это не проприетарная услуга с любыми взлетами. Jupyter - полностью открытая технология, разработанная UC Berkeley и Cal Poly San Luis Obispo. Rackspace жертвует эту услугу как часть процесса разработки.

Чтобы попробовать `numpy` на tmpnb.org:

1. посетить tmpnb.org
2. либо выберите « `Welcome to Python.ipynb` или
3. Новый >> Python 2 или
4. Новый >> Python 3

Прочитайте Начало работы с numpy онлайн: https://riptutorial.com/ru/numpy/topic/823/начало-работы-с-numpy

глава 2: numpy.cross

Синтаксис

- `numpy.cross(a, b)` # перекрестное произведение *a* и *b* (или векторов в *a* и *b*)
- `numpy.cross(a, b, axisa=-1)` #cross произведение векторов в с *Ь,* ул векторов в раскладывают вдоль оси *axisa*
- `numpy.cross(a, b, axisa=-1, axisb=-1, axisc=-1)` # кросс-продукты векторов в *a* и *b* , выходные векторы, выложенные вдоль оси, указанной *axisc*
- `numpy.cross(a, b, axis=None)` # перекрестные произведения векторов в *a* и *b* , векторы в *a* , *b* и в выводе, выложенные вдоль оси *оси*

параметры

колонка	колонка
а, б	В простейшем использовании `a` и `b` представляют собой два 2- или 3-элементных вектора. Они также могут быть массивами векторов (т. Е. Двумерных матриц). Если `a` - это массив, а «b» - вектор, `cross(a,b)` возвращает массив, элементы которого являются перекрестными произведениями каждого вектора в `a` с вектором `b` . `b` - массив, `a` - один вектор, `cross(a,b)` возвращает массив, элементы которого являются перекрестными произведениями `a` с каждым вектором в `b` . `a` и `b` могут быть массивами, если они имеют одинаковую форму. В этом случае `cross(a,b)` возвращает `cross(a[0],b[0]), cross(a[1], b[1]), ...`
axisa / б	Если `a` - массив, он может иметь векторы, расположенные на самой быстро меняющейся оси, самой медленной изменяющейся оси или что-то среднее между ними. `axisa` говорит `cross()` как векторы выложены в `a` . По умолчанию он принимает значение самой медленно меняющейся оси. `axisb` работает одинаково с входом `b` . Если вывод `cross()` будет массивом, выходные векторы могут быть выложены разными осями массива; `axisc` говорит `cross` , как раскладывать векторы в его выходном массиве. По умолчанию `axisc` указывает наиболее медленно меняющуюся ось.
ось	Удобный параметр, который по `axisa` устанавливает `axisa` , `axisb` и `axisc` значения в одно и то же значение. Если в вызове присутствует `axis` и любой другой параметр, значение `axis` будет переопределять другие значения.

Examples

Перекрестный продукт двух векторов

Numpy обеспечивает `cross` функцию для вычисления векторных кросс-продуктов. Перекрестное произведение векторов `[1, 0, 0]` и `[0, 1, 0]` - `[0, 0, 1]` . Numpy говорит нам:

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

как и ожидалось.

В то время как кросс-продукты обычно определяются только для трехмерных векторов. Тем не менее, любой из аргументов функции Numpy может быть двумя векторами элементов. Если вектор `c` задан как `[c1, c2]` , то Numpy присваивает ноль третьему размеру: `[c1, c2, 0]` . Так,

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

В отличие от `dot` которая существует как функция Numpy, так и метод `ndarray` , `cross` существует только как отдельная функция:

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

Множественные перекрестные продукты с одним вызовом

Любой вход может быть массивом трехмерных (или 2-) векторов элементов.

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

Результатом в этом случае является массив ([np.cross (a [0], b), np.cross (a [1], b), np.cross (a [2], b)])

`b` также может быть массивом трехмерных (или 2-) векторов элементов, но он должен иметь ту же форму, что и `a` . В противном случае вычисление не выполняется с ошибкой «несоответствие формы». Таким образом, мы можем

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

и теперь результатом является `array([np.cross(a[0],b[0]), np.cross(a[1],b[1]), np.cross(a[2],b[2])])`

Больше гибкости при использовании нескольких кросс-продуктов

В наших последних двух примерах numpy предположил, что `a[0,:]` был первым вектором, `a[1,:]` вторым и `a[2,:]` третьим. Numpy.cross имеет необязательную ось аргументов аргумента, которая позволяет нам указать, какая ось определяет векторы. Так,

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

Результат `axisa=1` результат по умолчанию - `(np.cross([1,1,1],b), np.cross([0,1,0],b), np.cross([1,0,-1],b))` . По умолчанию `axisa` всегда указывает последнюю (наиболее медленно меняющуюся) ось массива. Результат `axisa=0 (np.cross([1,0,1],b), np.cross([1,1,0],b), np.cross([1,0,-1],b))` .

Аналогичный необязательный параметр, `axisb` , выполняет ту же функцию для входа `b` , если он также является двумерным массивом.

Параметры axisa и axisb говорят о том, как распределять входные данные. Третий параметр, axisc говорит numpy, как распределять вывод, если `a` или `b` является многомерным. Используя те же самые входы `a` и `b` что и выше, мы получаем

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

Итак, `axisc=1` а `axisc` по умолчанию дают одинаковый результат, т. `axisc` Элементы каждого вектора смежны в индексе быстрого перемещения выходного массива. axisc по умолчанию является последней осью массива. `axisc=0` распределяет элементы каждого вектора через самую медленную переменную размерность массива.

Если вы хотите, чтобы `axisa` , `axisb` и `axisc` были одинаковыми, вам не нужно устанавливать все три параметра. Вы можете установить четвертый параметр, `axis` , на необходимое одиночное значение, а остальные три параметра будут автоматически установлены. ось переопределяет ось, ось b или axisc, если какая-либо из них присутствует в вызове функции.

Прочитайте numpy.cross онлайн: https://riptutorial.com/ru/numpy/topic/6166/numpy-cross

глава 3: numpy.dot

Синтаксис

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

параметры

название	подробности
	массив numpy
б	массив numpy
из	массив numpy

замечания

numpy.dot

Возвращает точечный продукт a и b. Если a и b являются скалярами или обоими 1-D массивами, тогда возвращается скаляр; в противном случае массив возвращается. Если дано, то оно возвращается.

Examples

Матричное умножение

Матричное умножение может выполняться двумя эквивалентными способами с помощью точечной функции. Один из способов - использовать функцию dot member функции 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.]])
```

Второй способ преобразования матрицы - с помощью функции numpy library.

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

Векторные точечные продукты

Функция точки также может использоваться для вычисления векторных точечных продуктов между двумя одномерными массивами numpy.

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

Параметр out

Функция numpy dot имеет необязательный параметр out = None. Этот параметр позволяет указать массив для записи результата. Этот массив должен быть точно такой же формы и типа, что и массив, который был бы возвращен, или будет выбрано исключение.

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

Попробуем изменить dtype результата на 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)
```

И если мы попробуем использовать другой базовый порядок памяти, скажем, стиль Fortran (поэтому столбцы смежны, а не строки), также возникает ошибка.

```
>>> 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)
```

Матричные операции над массивами векторов

numpy.dot может использоваться для умножения списка векторов на матрицу, но ориентация векторов должна быть вертикальной, так что список из восьми двух компонентных векторов появляется как два восьми вектора компонент:

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

Если список векторов выложен с его осями наоборот (что часто бывает), то массив необходимо перенести до и после операции с точкой, например:

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

Хотя функция точки очень быстрая, иногда лучше использовать einsum. Эквивалентом приведенного выше будет:

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

Это немного медленнее, но позволяет список вершин умножаться на соответствующий список матриц. Это был бы очень запутанный процесс с использованием точки:

```
>>> 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 можно использовать для нахождения точечного произведения каждого вектора в списке с соответствующим вектором в другом списке, это довольно грязно и медленно по сравнению с умножением по элементам и суммированием вдоль последней оси. Что-то вроде этого (что требует вычисления гораздо большего массива, но в основном игнорируется)

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

Точечный продукт с использованием умножения элементов и суммирования

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

Использование einsum может быть достигнуто с помощью

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

```
array([  82.,  104.,  130.,  160.,  194.,  232.,  274.,  320.])
```

Прочитайте numpy.dot онлайн: https://riptutorial.com/ru/numpy/topic/3198/numpy-dot

глава 4: Булевское индексирование

Examples

Создание логического массива

Булевский массив можно создать вручную, используя `dtype=bool` при создании массива. Значения, отличные от `0` , `None` , `False` или пустые строки, считаются 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]
```

В качестве альтернативы, numpy автоматически создает логический массив при сравнении между массивами и скалярами или между массивами одной и той же формы.

```
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
```

Прочитайте Булевское индексирование онлайн: https://riptutorial.com/ru/numpy/topic/6072/булевское-индексирование

глава 5: Генерация случайных данных

Вступление

`random` модуль NumPy предоставляет удобные методы для генерации случайных данных, имеющих желаемую форму и распределение.

Вот официальная документация .

Examples

Создание простого случайного массива

```
# 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 ])
```

Установка семян

Использование `random.seed` :

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

Создав объект-генератор случайных чисел:

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

Создание случайных целых чисел

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

Выбор случайной выборки из массива

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

Выберите три буквы случайным образом (*с заменой* - один и тот же элемент можно выбрать несколько раз):

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

Отбор проб без замены:

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

Назначьте вероятность каждой букве:

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

Генерация случайных чисел, полученных из конкретных распределений

Нарисуйте образцы из нормального (гауссовского) распределения

```
# 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])
```

Существует несколько дополнительных распределений, доступных в `numpy.random` , например, `poisson` , `binomial` и `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])
```

Прочитайте Генерация случайных данных онлайн: https://riptutorial.com/ru/numpy/topic/2060/генерация-случайных-данных

глава 6: Линейная алгебра с np.linalg

замечания

Начиная с версии 1.8, некоторые из подпрограмм в `np.linalg` могут работать с «стеком» матриц. То есть, процедура может вычислять результаты для нескольких матриц, если они сложены вместе. Например, `A` здесь интерпретируется как две сложенные матрицы 3 на 3:

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

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

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

Официальные `np` docs определяют это через спецификации параметров, такие `a : (..., M, M) array_like` .

Examples

Решить линейные системы с np.solve

Рассмотрим следующие три уравнения:

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

Мы можем выразить эту систему как матричное уравнение `A * x = b` с:

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

Затем используйте `np.linalg.solve` для решения для `x` :

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

`A` должна быть квадратной и полноразмерной матрицей: все ее строки должны быть линейно независимыми. `A` должно быть обратимым / неособенным (его определитель не равен нулю). Например, если одна строка из `A` является кратной другой, вызов `linalg.solve`

поднимет `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])
```

Такие системы могут быть решены с помощью `np.linalg.lstsq` .

Найти решение наименьших квадратов линейной системы с np.linalg.lstsq

Наименьшие квадраты - это стандартный подход к задачам с большим количеством уравнений, чем неизвестные, также известные как переопределенные системы.

Рассмотрим четыре уравнения:

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

Мы можем выразить это как матричное умножение `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])
```

Затем разрешите с помощью `np.linalg.lstsq` :

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

`x` - это решение, `residuals` суммы, `rank` матрицы входа `A` и `s` - особые значения `A` Если `b` имеет более одного измерения, `lstsq` будет решать систему, соответствующую каждому столбцу `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` и `s` зависят только от `A` и, таким образом, те же, что и выше.

Прочитайте Линейная алгебра с np.linalg онлайн: https://riptutorial.com/ru/numpy/topic/3753/линейная-алгебра-с-np-linalg

глава 7: Массивы

Вступление

N-мерные массивы или `ndarrays` являются основным объектом numpy, который используется для хранения элементов одного и того же типа данных. Они обеспечивают эффективную структуру данных, превосходящую обычные массивы Python.

замечания

По возможности вырабатывайте операции с данными в терминах массивов и векторных операций. Операции с вектором выполняются намного быстрее, чем эквивалент для циклов

Examples

Создать массив

Пустой массив

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

Обратите внимание, что в этом случае значения в этом массиве не заданы. Таким образом, создание массива полезно только тогда, когда массив заполняется позже в коде.

Из списка

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

Создать диапазон

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

Создать нули

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

Создать

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

Создание элементов массива с линейным интервалом

```
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.  ])
```

Создание элементов массива с разбиением на лог

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

Создать массив из заданной функции

```
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.]])
```

Операторы массива

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

скалярное добавление является элементарным

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

скалярное умножение является элементарным

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

добавление массива является элементарным

```
x+x
```

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

умножение массива является элементарным

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

dot (или, в более общем случае, матричное умножение) выполняется с помощью функции

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

В Python 3.5 оператор `@` был добавлен как оператор инфикса для матричного умножения

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

Добавить . Возвращает копию с добавленными значениями. НЕ на месте.

```
#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 . Горизонтальный стек. (столбец столбцов)
vstack . Вертикальный стек. (строка строк)

```
# 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]])
```

Доступ к массиву

Синтаксис Slice - это `i:j:k` где `i` - начальный индекс (включительно), `j` - индекс остановки (исключение), а `k` - размер шага. Как и другие структуры данных python, первый элемент имеет индекс 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])
```

Отрицательные значения рассчитываются с конца массива. `-1` поэтому обращается к последнему элементу массива:

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

Доступ к многомерным массивам можно получить, указав каждое измерение, разделенное запятыми. Все предыдущие правила применяются.

```
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]])
```

Транспонирование массива

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

Использование метода `.transpose` :

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

`.T` :

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

Или `np.transpose` :

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

В случае двумерного массива это эквивалентно стандартной матричной транспозиции (как показано выше). В n-мерном случае вы можете указать перестановку осей массива. По умолчанию это отменяет `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]]])
```

Но возможна любая перестановка индексов осей:

```
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)
```

Булевское индексирование

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

Сравнение со скаляром возвращает логический массив:

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

Этот массив можно использовать для индексирования, чтобы выбрать только числа больше 4:

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

Булевое индексирование может использоваться между различными массивами (например, связанные параллельные массивы):

```
# 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])
```

Изменение формы массива

Метод `numpy.reshape` (такой же, как `numpy.ndarray.reshape`) возвращает массив того же общего размера, но в новой форме:

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

Он возвращает новый массив и не работает на месте:

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

Тем не менее, можно перезаписать `shape` атрибут `ndarray` :

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

Сначала это поведение может быть неожиданным, но `ndarray` s хранятся в смежных блоках памяти, и их `shape` определяет только то, как этот поток данных следует интерпретировать как многомерный объект.

До одной оси в корте `shape` может иметь значение `-1` . Затем `numpy` выведет длину этой оси для вас:

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

Или же:

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

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

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

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

Множественные неопределенные измерения, например `a.reshape((3, -1, -1))` , не допускаются и будут `ValueError` .

Операции вещательного массива

Арифметические операции выполняются по элементам на массивах Numpy. Для массивов одинаковой формы это означает, что операция выполняется между элементами с соответствующими индексами.

```
# 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])
```

Арифметические операции могут также выполняться на массивах разной формы с помощью широковещательной *передачи* Numpy. В общем, один массив «передается» по другому, так что поэлементные операции выполняются на под-массивах конгруэнтной формы.

```
# 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]])
```

Чтобы проиллюстрировать это далее, рассмотрим умножение двумерных и трехмерных массивов с конгруэнтными субразмерами.

```
# 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.
```

Когда применяется широковещательная передача массива?

Вещание происходит, когда два массива имеют *совместимые* формы.

Формы сравниваются по компонентам, начиная с конечных. Два измерения совместимы, если они одинаковы или один из них равен `1` . Если одна форма имеет более высокий размер, чем другой, то превосходящие компоненты не сравниваются.

Некоторые примеры совместимых форм:

```
(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)
```

Вот официальная документация по широковещанию массивов .

Заполнение массива содержимым файла CSV

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

Многие варианты поддерживаются, см. Официальную документацию для полного списка:

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

Нумерометр n-мерного массива: ndarray

Основной структурой данных в numpy является `ndarray` (сокращение от *n*- мерного массива). `ndarray` s

- однородные (т.е. они содержат элементы одного и того же типа данных)
- содержат элементы фиксированных размеров (заданные *формой* , кортеж из *n* положительных целых чисел, которые определяют размеры каждого измерения)

Одномерный массив:

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

Двумерный массив:

```
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)
```

Трехмерная:

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

Для инициализации массива без указания его содержимого используйте:

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

Догадка типа данных и автоматическое литье

Тип данных установлен как плавающий по умолчанию

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

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

Если предоставлены некоторые данные, numpy угадает тип данных:

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

```
x.dtype
# dtype('int32')
```

Обратите внимание: при выполнении присвоений numpy попытается автоматически `ndarray` значения в соответствии с `ndarray` данных `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
```

Передача массивов

См. Также операции массива вещания .

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

В матричной терминологии мы будем иметь матрицу 2x2 и вектор строки 1x2. Тем не менее мы можем сделать сумму

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

Это связано с тем, что массив `y` « *растянут* » до:

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

в соответствии с формой `x` .

Ресурсы:

- Введение в ndarray из официальной документации: N-мерный массив (ndarray)
- Ссылка на класс: ndarray .

Прочитайте Массивы онлайн: https://riptutorial.com/ru/numpy/topic/1296/массивы

глава 8: подклассификация ndarray

Синтаксис

- `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

Отслеживание дополнительной собственности на массивы

```
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)
```

Для `context` кортежа `func` является объектом `np.add` , таким как `np.add` , `args` является `tuple` ,

a `which_return_val` - целым числом, определяющим, какое возвращаемое значение ufunc обрабатывается

Прочитайте подклассификация ndarray онлайн: https://riptutorial.com/ru/numpy/topic/6431/подклассификация-ndarray

глава 9: Простая линейная регрессия

Вступление

Установка линии (или другой функции) в набор точек данных.

Examples

Использование np.polyfit

Мы создаем набор данных, который мы тогда поместим с прямой $ 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
```

Чтобы посмотреть, что мы сделали:

```
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()
```

Примечание. В этом примере подробно описывается документация numpy по адресу https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html .

Использование np.linalg.lstsq

Мы используем тот же набор данных, что и с полифитом:

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

Теперь мы пытаемся найти решение, минимизируя систему линейных уравнений A b = c, минимизируя | 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()
```

Примечание. В этом примере подробно описывается документация numpy на https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html .

Прочитайте Простая линейная регрессия онлайн: https://riptutorial.com/ru/numpy/topic/8808/простая-линейная-регрессия

глава 10: Сохранение и загрузка массивов

Вступление

Массивы можно сохранять и загружать различными способами.

Examples

Использование numpy.save и numpy.load

np.save и np.load предоставляют простую в использовании платформу для сохранения и загрузки массивов numpy произвольного размера:

```
import numpy as np

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

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

Прочитайте Сохранение и загрузка массивов онлайн:
https://riptutorial.com/ru/numpy/topic/10891/сохранение-и-загрузка-массивов

глава 11: Файл IO с numpy

Examples

Сохранение и загрузка массивов numpy с использованием двоичных файлов

```
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
```

Загрузка числовых данных из текстовых файлов с последовательной структурой

Функция `np.loadtxt` может использоваться для чтения csv-подобных файлов:

```
# 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.]])
```

Тот же файл можно прочитать с помощью регулярного выражения с помощью `np.fromregex` :

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

Сохранение данных в формате CSV в стиле ASCII

Аналоговый с `np.loadtxt` , `np.savetxt` можно использовать для сохранения данных в ASCII-файле

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

Для управления форматированием:

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

Выход:

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

Чтение файлов CSV

Доступны три основные функции (описание на страницах руководства):

> `fromfile` - высокоэффективный способ чтения двоичных данных с известным типом данных, а также для разбора просто форматированных текстовых файлов. Данные, написанные с использованием метода tofile, могут быть прочитаны с использованием этой функции.
>
> `genfromtxt` - загрузка данных из текстового файла с отсутствующими значениями, обрабатываемыми как указано. Каждая строка, прошедшая первые строки skip_header, делится на символ разделителя, а символы, следующие за символом комментариев, отбрасываются.
>
> `loadtxt` - загрузка данных из текстового файла. Каждая строка в текстовом файле должна иметь одинаковое количество значений.

`genfromtxt` - это функция-обертка для `loadtxt` . `genfromtxt` является наиболее прямым для использования, поскольку он имеет множество параметров для работы с входным файлом.

Согласованное количество столбцов, согласованный тип данных (числовой или строковый):

Учитывая входной файл, `myfile.csv` с содержимым:

```
#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)
```

дает массив:

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

Согласованное количество столбцов, смешанный тип данных (по столбцам):

```
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')])
```

Обратите внимание, что использование `dtype=None` приводит к повторному анализу.

Непоследовательное количество столбцов:

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

В массив одиночных строк:

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

Прочитайте Файл IO с numpy онлайн: https://riptutorial.com/ru/numpy/topic/4973/файл-io-с-numpy

глава 12: Фильтрация данных

Examples

Фильтрация данных с помощью логического массива

Когда только один аргумент передается в numpy, `where` функция возвращает индексы входного массива (`condition`), которые оцениваются как истинные (такое же поведение, как `numpy.nonzero`). Это можно использовать для извлечения индексов массива, которые удовлетворяют заданному условию.

```
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]
```

Если вам не нужны индексы, это может быть достигнуто за один шаг с помощью `extract` , где agian задает `condition` как первый аргумент, но дайте `array` вернуть значения, из которых условие истинно, как второй аргумент.

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

Два дополнительных аргумента `x` и `y` могут быть предоставлены туда, `where` в этом случае вывод будет содержать значения `x` где условие равно `True` и значения `y` где условие `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]])
```

Прямая фильтрация индексов

Для простых случаев вы можете напрямую фильтровать данные.

```
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]
```

Прочитайте Фильтрация данных онлайн: https://riptutorial.com/ru/numpy/topic/6187/фильтрация-данных

кредиты

S. No	Главы	Contributors
1	Начало работы с numpy	Andras Deak, Chris Mueller, Code-Ninja, Community, Dux, edwinksl, grovduck, Hammer, hashcode55, Joshua Cook, karel, Kersten, Mpaull, prasastoadi, rlee827, sebix, user2314737, Yassie
2	numpy.cross	bob.sacamento, Mad Physicist
3	numpy.dot	bpachev, paddyg, Shubham Dang
4	Булевское индексирование	Chris Mueller
5	Генерация случайных данных	amin, ayhan, B8vrede, Dux, Fermi paradox, user2314737
6	Линейная алгебра с np.linalg	Daniel, DataSwede, Fermi paradox, Mahdi, Sean Easter
7	Массивы	Andras Deak, ayhan, B Samedi, B8vrede, Benjamin, DataSwede , Dux, Gwen, Hamlet, Hammer, KARANJ, Keith L, pixatlazaki, Ryan, Sean Easter, Sparkler, The Hagen, TPVasconcelos, user2314737
8	подклассификация ndarray	Eric
9	Простая линейная регрессия	Alex
10	Сохранение и загрузка массивов	obachtos
11	Файл IO с numpy	Alex, atomh33ls, Sparkler
12	Фильтрация данных	Alex, farleytpm