LEARNING numpy

#numpy

Table of Contents

About

Chapter 1: Getting started with numpy

Remarks

NumPy (pronounced “numb pie” or sometimes “numb pea”) is an extension to the Python programming language that adds support for large, multi-dimensional arrays, along with an extensive library of high-level mathematical functions to operate on these arrays.

Versions

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

Version	Release Date
1.10.4*	2016-01-07
1.11.0	2016-05-29

Examples

Installation on Mac

The easiest way to set up NumPy on Mac is with pip

```
pip install numpy
```

Installation using Conda.
Conda available for Windows, Mac, and Linux

1. Install Conda. There are two ways to install Conda, either with Anaconda (Full package, include numpy) or Miniconda (only Conda,Python, and the packages they depend on, without any additional package). Both Anaconda & Miniconda install the same Conda.
2. Additional command for Miniconda, type the command `conda install numpy`

Installation on Windows

Numpy installation through pypi (the default package index used by pip) generally fails on Windows computers. The easiest way to install on Windows is by using precompiled binaries.

One source for precompiled wheels of many packages is Christopher Gohkle's site. Choose a version according to your Python version and system. An example for Python 3.5 on a 64 bit system:

1. Download `numpy-1.11.1+mkl-cp35-cp35m-win_amd64.whl` from here
2. Open a Windows terminal (cmd or powershell)
3. Type the command `pip install C:\path_to_download\numpy-1.11.1+mkl-cp35-cp35m-win_amd64.whl`

If you don't want to mess around with single packages, you can use the Winpython distribution which bundles most packages together and provides a confined environment to work with. Similarly, the Anaconda Python distrubution comes pre-installed with numpy and numerous other common packages.

Another popular source is the `conda` package manager, which also supports virtual environments.

1. Download and install `conda`.
2. Open a Windows terminal.
3. Type the command `conda install numpy`

Installation on Linux

NumPy is available in the default repositories of most popular Linux distributions and can be installed in the same way that packages in a Linux distribution are usually installed.

Some Linux distributions have different NumPy packages for Python 2.x and Python 3.x. In Ubuntu and Debian, install `numpy` at the system level using the APT package manager:

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

For other distributions, use their package managers, like zypper (Suse), yum (Fedora) etc.

`numpy` can also be installed with Python's package manager `pip` for Python 2 and with `pip3` for Python 3:

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

`pip` is available in the default repositories of most popular Linux distributions and can be installed for Python 2 and Python 3 using:

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

After installation, use `pip` for Python 2 and `pip3` for Python 3 to use pip for installing Python packages. But note that you might need to install many dependencies, which are required to build numpy from source (including development-packages, compilers, fortran etc).

Besides installing `numpy` at the system level, it is also common (perhaps even highly recommended) to install `numpy` in virtual environments using popular Python packages such as `virtualenv`. In Ubuntu, `virtualenv` can be installed using:

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

Then, create and activate a virtualenv for either Python 2 or Python 3 and then use `pip` to install `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
```

Basic Import

Import the numpy module to use any part of it.

```
import numpy as np
```

Most examples will use `np` as shorthand for numpy. Assume "np" means "numpy" in code examples.

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

Temporary Jupyter Notebook hosted by Rackspace

Jupyter Notebooks are an interactive, browser-based development environment. They were originally developed to run computation python and as such play very well with numpy. To try numpy in a Jupyter notebook without fully installing either on one's local system Rackspace provides free temporary notebooks at tmpnb.org.

Note: that this is not a proprietary service with any sort of upsells. Jupyter is a wholly open-sourced technology developed by UC Berkeley and Cal Poly San Luis Obispo. Rackspace donates this service as part of the development process.

To try `numpy` at tmpnb.org:

1. visit tmpnb.org
2. either select `Welcome to Python.ipynb` or
3. New >> Python 2 or
4. New >> Python 3

Read Getting started with numpy online: https://riptutorial.com/numpy/topic/823/getting-started-with-numpy

Chapter 2: Arrays

Introduction

N-dimensional arrays or `ndarrays` are numpy's core object used for storing items of the same data type. They provide an efficient data structure that is superior to ordinary Python's arrays.

Remarks

Whenever possible express operations on data in terms of arrays and vector operations. Vector operations execute much faster than equivalent for loops

Examples

Create an Array

Empty array

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

Note that in this case, the values in this array are not set. This way of creating an array is therefore only useful if the array is filled later in the code.

From a list

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

Create a range

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

Create Zeros

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

Create Ones

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

Create linear-spaced array items

```
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.  ])
```

Create log-spaced array items

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

Create array from a given function

```
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 operators

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

scalar addition is element wise

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

scalar multiplication is element wise

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

array addition is element wise

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

array multiplication is element wise

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

dot product (or more generally matrix multiplication) is done with a function

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

In Python 3.5, the `@` operator was added as an infix operator for matrix multiplication

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

Append. Returns copy with values appended. NOT in-place.

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

hstack. Horizontal stack. (column stack)
vstack. Vertical stack. (row stack)

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

Array Access

Slice syntax is `i:j:k` where `i` is the starting index (inclusive), `j` is the stopping index (exclusive) and `k` is the step size. Like other python data structures, the first element has an index of 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 values count in from the end of the array. `-1` therefore accesses the last element in an array:

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

Multi-dimensional arrays can be accessed by specifying each dimension separated by commas. All previous rules apply.

```
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]])
```

Transposing an array

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

Using `.transpose` method:

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

`.T` method:

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

Or `np.transpose`:

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

In the case of a 2-dimensional array, this is equivalent to a standard matrix transpose (as depicted above). In the n-dimensional case, you may specify a permutation of the array axes. By default, this reverses `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]]])
```

But any permutation of the axis indices is possible:

```
a.transpose(2,0,1)
# Out:
#    array([[[ 0,  2],
#             [ 4,  6],
#             [ 8, 10]],
#
#            [[ 1,  3],
#             [ 5,  7],
#             [ 9, 11]]])

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

```
# Out:
#    (4, 2, 3)
```

Boolean indexing

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

Comparison with a scalar returns a boolean array:

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

This array can be used in indexing to select only the numbers greater than 4:

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

Boolean indexing can be used between different arrays (e.g. related parallel arrays):

```
# 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])
```

Reshaping an array

The `numpy.reshape` (same as `numpy.ndarray.reshape`) method returns an array of the same total size, but in a new shape:

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

It returns a new array, and doesn't operate in place:

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

However, it *is* possible to overwrite the `shape` attribute of an `ndarray`:

```
a = np.arange(12)
a.shape = (3, 4)
```

```
print(a)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]
```

This behavior might be surprising at first, but `ndarray`s are stored in contiguous blocks of memory, and their `shape` only specifies how this stream of data should be interpreted as a multidimensional object.

Up to one axis in the `shape` tuple can have a value of `-1`. `numpy` will then infer the length of this axis for you:

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

Or:

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

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

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

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

Multiple unspecified dimensions, e.g. `a.reshape((3, -1, -1))` are not allowed and will throw a `ValueError`.

Broadcasting array operations

Arithmetic operations are performed elementwise on Numpy arrays. For arrays of identical shape, this means that the operation is executed between elements at corresponding indices.

```
# 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]])
```

Arithmetic operations may also be executed on arrays of different shapes by means of Numpy *broadcasting*. In general, one array is "broadcast" over the other so that elementwise operations are performed on sub-arrays of congruent shape.

```
# 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]])
```

To illustrate this further, consider the multiplication of 2D and 3D arrays with congruent sub-dimensions.

```
# 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.
```

When is array broadcasting applied?

Broadcasting takes place when two arrays have *compatible* shapes.

Shapes are compared component-wise starting from the trailing ones. Two dimensions are compatible if either they're the same or one of them is `1`. If one shape has higher dimension than the other, the exceeding components are not compared.

Some examples of compatible shapes:

```
(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)
```

Here's the official documentation on array broadcasting.

Populate an array with the contents of a CSV file

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

Many options are supported, see official documentation for full list:

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

Numpy n-dimensional array: the ndarray

The core data structure in numpy is the `ndarray` (short for *n*-dimensional array). `ndarray`s are

- homogeneous (i.e. they contain items of the same data-type)
- contain items of fixed sizes (given by a *shape*, a tuple of *n* positive integers that specify the sizes of each dimension)

One-dimensional array:

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

Two-dimensional array:

```
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)
```

Three-dimensional:

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

To initialize an array without specifying its contents use:

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

Datatype guessing and automatic casting

The data-type is set to float by default

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

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

If some data is provided, numpy will guess the data-type:

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

Note that when doing assignments numpy will attempt to automatically cast values to suit the `ndarray`'s datatype

```
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 broadcasting

See also Broadcasting array operations.

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

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

In matrix terminology, we would have a 2x2 matrix and a 1x2 row vector. Still we're able to do a sum

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

This is because the array `y` is "*stretched*" to:

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

to suit the shape of `x`.

Resources:

- Introduction to the ndarray from the official documentation: The N-dimensional array (ndarray)
- Class reference: ndarray.

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

Chapter 3: Boolean Indexing

Examples

Creating a boolean array

A boolean array can be created manually by using `dtype=bool` when creating the array. Values other than `0`, `None`, `False` or empty strings are considered 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]
```

Alternatively, numpy automatically creates a boolean array when comparisons are made between arrays and scalars or between arrays of the same shape.

```
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
```

Read Boolean Indexing online: https://riptutorial.com/numpy/topic/6072/boolean-indexing

Chapter 4: File IO with numpy

Examples

Saving and loading numpy arrays using binary files

```
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
```

Loading numerical data from text files with consistent structure

The function `np.loadtxt` can be used to read csv-like files:

```
# 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.]])
```

The same file could be read using a regular expression with `np.fromregex`:

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

Saving data as CSV style ASCII file

Analog to `np.loadtxt`, `np.savetxt` can be used to save data in an ASCII file

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

To control formatting:

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

Output:

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

Reading CSV files

Three main functions available (description from man pages):

> `fromfile` - A highly efficient way of reading binary data with a known data-type, as well as parsing simply formatted text files. Data written using the tofile method can be read using this function.
>
> `genfromtxt` - Load data from a text file, with missing values handled as specified. Each line past the first skip_header lines is split at the delimiter character, and characters following the comments character are discarded.
>
> `loadtxt` - Load data from a text file. Each row in the text file must have the same number of values.

`genfromtxt` is a wrapper function for `loadtxt`. `genfromtxt` is the most straight-forward to use as it has many parameters for dealing with the input file.

Consistent number of columns, consistent data type (numerical or string):

Given an input file, `myfile.csv` with the contents:

```
#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)
```

gives an array:

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

Consistent number of columns, mixed data type (across columns):

```
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')])
```

Note the use of `dtype=None` results in a recarray.

Inconsistent number of columns:

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

Into single row array:

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

Read File IO with numpy online: https://riptutorial.com/numpy/topic/4973/file-io-with-numpy

Chapter 5: Filtering data

Examples

Filtering data with a boolean array

When only a single argument is supplied to numpy's `where` function it returns the indices of the input array (the `condition`) that evaluate as true (same behaviour as `numpy.nonzero`). This can be used to extract the indices of an array that satisfy a given condition.

```
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]
```

If you do not need the indices, this can be achieved in one step using `extract`, where you agian specify the `condition` as the first argument, but give the `array` to return the values from where the condition is true as the second argument.

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

Two further arguments `x` and `y` can be supplied to `where`, in which case the output will contain the values of `x` where the condition is `True` and the values of `y` where the condition is `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]])
```

Directly filtering indices

For simple cases, you can filter data directly.

```
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]
```

Read Filtering data online: https://riptutorial.com/numpy/topic/6187/filtering-data

Chapter 6: Generating random data

Introduction

The `random` module of NumPy provides convenient methods for generating random data having the desired shape and distribution.

Here's the official documentation.

Examples

Creating a simple random array

```
# 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 ])
```

Setting the seed

Using `random.seed`:

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

By creating a random number generator object:

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

Creating random integers

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

Selecting a random sample from an array

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

Select three letters randomly (*with replacement* - same item can be chosen multiple times):

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

Sampling without replacement:

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

Assign probability to each letter:

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

Generating random numbers drawn from specific distributions

Draw samples from a normal (gaussian) distribution

```
# 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])
```

There are several additional distributions available in `numpy.random`, for example `poisson`, `binomial` and `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])
```

Read Generating random data online: https://riptutorial.com/numpy/topic/2060/generating-random-data

Chapter 7: Linear algebra with np.linalg

Remarks

As of version 1.8, several of the routines in `np.linalg` can operate on a 'stack' of matrices. That is, the routine can calculate results for multiple matrices if they're stacked together. For example, `A` here is interpreted as two stacked 3-by-3 matrices:

```
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])
```

The official `np` docs specify this via parameter specifications like `a : (..., M, M) array_like`.

Examples

Solve linear systems with np.solve

Consider the following three equations:

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

We can express this system as a matrix equation `A * x = b` with:

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

Then, use `np.linalg.solve` to solve for `x`:

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

`A` must be a square and full-rank matrix: All of its rows must be be linearly independent. `A` should be invertible/non-singular (its determinant is not zero). For example, If one row of `A` is a multiple of another, calling `linalg.solve` will raise `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])
```

Such systems can be solved with `np.linalg.lstsq`.

Find the least squares solution to a linear system with np.linalg.lstsq

Least squares is a standard approach to problems with more equations than unknowns, also known as overdetermined systems.

Consider the four equations:

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

We can express this as a matrix multiplication `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])
```

Then solve with `np.linalg.lstsq`:

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

`x` is the solution, `residuals` the sum, `rank` the matrix rank of input `A`, and `s` the singular values of `A`. If `b` has more than one dimension, `lstsq` will solve the system corresponding to each column of `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` and `s` depend only on `A`, and are thus the same as above.

Read Linear algebra with np.linalg online: https://riptutorial.com/numpy/topic/3753/linear-algebra-with-np-linalg

Chapter 8: numpy.cross

Syntax

- `numpy.cross(a, b)` # cross product of *a* and *b* (or vectors in *a* and *b*)
- `numpy.cross(a, b, axisa=-1)` #cross product of vectors in *a* with *b*, s.t. vectors in a are laid out along axis *axisa*
- `numpy.cross(a, b, axisa=-1, axisb=-1, axisc=-1)` # cross products of vectors in *a* and *b*, output vectors laid out along axis specified by *axisc*
- `numpy.cross(a, b, axis=None)` # cross products of vectors in *a* and *b*, vectors in *a*, *b*, and in output laid out along axis *axis*

Parameters

Column	Column
a,b	In simplest usage, `a` and `b` are two 2- or 3-element vectors. They can also be arrays of vectors (i.e. two-dimensional matrices). If `a` is an array and 'b' is a vector, `cross(a,b)` returns an array whose elements are the cross products of each vector in `a` with the vector `b`. The `b` is an array and `a` is a single vector, `cross(a,b)` returns an array whose elements are the cross products of `a` with each vector in `b`. `a` and `b` can both be arrays if they have the same shape. In this case, `cross(a,b)` returns `cross(a[0],b[0]), cross(a[1], b[1]), ...`
axisa/b	If `a` is an array, it can have vectors laid out across the most quickly varying axis, the slowest varying axis, or something in between. `axisa` tells `cross()` how the vectors are laid out in `a`. By default, it takes the value of the most slowly varying axis. `axisb` works the same with input `b`. If the output of `cross()` is going to be an array, the output vectors can be laid out different array axes; `axisc` tells `cross` how to lay out the vectors in its output array. By default, `axisc` indicates the most slowly varying axis.
axis	A convenience parameter that sets `axisa`, `axisb`, and `axisc` all to the same value if desired. If `axis` and any of the other parameters are present in the call, the value of `axis` will override the other values.

Examples

Cross Product of Two Vectors

Numpy provides a `cross` function for computing vector cross products. The cross product of vectors `[1, 0, 0]` and `[0, 1, 0]` is `[0, 0, 1]`. Numpy tells us:

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

as expected.

While cross products are normally defined only for three dimensional vectors. However, either of the arguments to the Numpy function can be two element vectors. If vector `c` is given as `[c1, c2]`, Numpy assigns zero to the third dimension: `[c1, c2, 0]`. So,

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

Unlike `dot` which exists as both a Numpy function and a method of `ndarray`, `cross` exists only as a standalone function:

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

Multiple Cross Products with One Call

Either input can be an array of 3- (or 2-) element vectors.

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

The result in this case is array([np.cross(a[0],b), np.cross(a[1],b), np.cross(a[2],b)])

`b` can also be an array of 3- (or 2-) element vectors, but it must have the same shape as `a`. Otherwise the calculation fails with a "shape mismatch" error. So we can have

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

and now the result is `array([np.cross(a[0],b[0]), np.cross(a[1],b[1]), np.cross(a[2],b[2])])`

More Flexibility with Multiple Cross Products

In our last two examples, numpy assumed that `a[0,:]` was the first vector, `a[1,:]` the second, and `a[2,:]` the third. Numpy.cross has an optional argument axisa that allows us to specify which axis defines the vectors. 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]])
```

The `axisa=1` result and the default result are both `(np.cross([1,1,1],b), np.cross([0,1,0],b), np.cross([1,0,-1],b))`. By default, `axisa` always indicates the last (most slowly varying) axis of the array. The `axisa=0` result is `(np.cross([1,0,1],b), np.cross([1,1,0],b), np.cross([1,0,-1],b))`.

A similar optional parameter, `axisb`, performs the same function for the `b` input, if it is also a 2-dimensional array.

Parameters axisa and axisb tell numpy how to distribute the input data. A third parameter, axisc tells numpy how to distribute the output if `a` or `b` is multi-dimensional. Using the same inputs `a` and `b` as above, we get

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

So `axisc=1` and the default `axisc` both give the same result, that is, the elements of each vector are contiguous in the fastest moving index of the output array. axisc is by default the last axis of the array. `axisc=0` distributes the elements of each vector across the slowest varying dimension of the array.

If you want `axisa`, `axisb`, and `axisc` to all have the same value, you do not need to set all three parameters. You can set a fourth parameter, `axis`, to the needed single value and the other three parameters will be automatically set. axis overrides axisa, axisb, or axisc if any of them are present in the function call.

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

Chapter 9: numpy.dot

Syntax

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

Parameters

Name	Details
a	a numpy array
b	a numpy array
out	a numpy array

Remarks

numpy.dot

Returns the dot product of a and b. If a and b are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned. If out is given, then it is returned.

Examples

Matrix multiplication

Matrix multiplication can be done in two equivalent ways with the dot function. One way is to use the dot member function of 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.]])
```

The second way to do matrix multiplication is with the numpy library function.

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

Vector dot products

The dot function can also be used to compute vector dot products between two one-dimensional numpy arrays.

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

The out parameter

The numpy dot function has an optional parameter out=None. This parameter allows you to specify an array to write the result to. This array must be exactly the same shape and type as the array that would have been returned, or an exception will be thrown.

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

Let's try changing the dtype of result to 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)
```

And if we try using a different underlying memory order, say Fortran-style (so columns are contiguous instead of rows), an error also results.

```
>>> 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)
```

Matrix operations on arrays of vectors

numpy.dot can be used to multiply a list of vectors by a matrix but the orientation of the vectors must be vertical so that a list of eight two component vectors appears like two eight components vectors:

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

If the list of vectors is laid out with its axes the other way round (which is often the case) then the array needs to be transposed before and then after the dot operation like this:

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

Although the dot function is very fast, sometimes it is better to use einsum. The equivalent of the above would be:

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

Which is a little slower but allows a list of vertices to be multiplied by a corresponding list of matrices. This would be a very convoluted process using dot:

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

numpy.dot can be used to find the dot product of each vector in a list with a corresponding vector in another list this is quite messy and slow compared with element-wise multiplication and summing along the last axis. Something like this (which requires a much larger array to be calculated but mostly ignored)

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

Dot product using element-wise multiplication and summing

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

Using einsum could be achieved with

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

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

Chapter 10: Saving and loading of Arrays

Introduction

Numpy arrays can be saved and loaded in various ways.

Examples

Using numpy.save and numpy.load

np.save and np.load provide a easy to use framework for saving and loading of arbitrary sized 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
```

Read Saving and loading of Arrays online: https://riptutorial.com/numpy/topic/10891/saving-and-loading-of-arrays

Chapter 11: Simple Linear Regression

Introduction

Fitting a line (or other function) to a set of data points.

Examples

Using np.polyfit

We create a dataset that we then fit with a straight line $f(x) = m x + c$.

```
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
```

To see what we've done:

```
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()
```

Note: This example follows the numpy documentation at https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html quite closely.

Using np.linalg.lstsq

We use the same dataset as with polyfit:

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

Now, we try to find a solution by minimizing the system of linear equations A b = c by minimizing |c-A 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()
```

Note: This example follows the numpy documentation at https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html quite closely.

Read Simple Linear Regression online: https://riptutorial.com/numpy/topic/8808/simple-linear-regression

Chapter 12: subclassing 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

Tracking an extra property on 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)
```

For the `context` tuple, `func` is a ufunc object such as `np.add`, `args` is a `tuple`, and `which_return_val` is an integer specifying which return value of the ufunc is being processed

Read subclassing ndarray online: https://riptutorial.com/numpy/topic/6431/subclassing-ndarray

Credits

S. No	Chapters	Contributors
1	Getting started with 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	Boolean Indexing	Chris Mueller
4	File IO with numpy	Alex, atomh33ls, Sparkler
5	Filtering data	Alex, farleytpm
6	Generating random data	amin, ayhan, B8vrede, Dux, Fermi paradox, user2314737
7	Linear algebra with np.linalg	Daniel, DataSwede, Fermi paradox, Mahdi, Sean Easter
8	numpy.cross	bob.sacamento, Mad Physicist
9	numpy.dot	bpachev, paddyg, Shubham Dang
10	Saving and loading of Arrays	obachtos
11	Simple Linear Regression	Alex
12	subclassing ndarray	Eric