Numpy

```
import numpy as np
```

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

```
array([1, 2, 3, 4, 5])
```

```
from numpy import doc
```

```
help(doc)
```

```
Help on package numpy.doc in numpy:

NAME
    numpy.doc

DESCRIPTION
    Topical documentation
    =====================

    The following topics are available:

    - basics
    - broadcasting
    - byteswapping
    - constants
    - creation
    - dispatch
    - glossary
    - indexing
    - internals
    - misc
    - structured_arrays
    - subclassing
    - ufuncs

    You can view them by

    >>> help(np.doc.TOPIC)                                  #doctest: +SKIP

PACKAGE CONTENTS
    basics
    broadcasting
    byteswapping
    constants
```

```
        creation
        dispatch
        glossary
        indexing
        internals
        misc
        structured_arrays
        subclassing
        ufuncs

    DATA
        __all__ = ['basics', 'broadcasting', 'byteswapping', 'constants', 'cre...

    FILE
        /usr/local/lib/python3.7/dist-packages/numpy/doc/__init__.py
```

```
help(np.doc.basics)
```

```
            >>> np.iinfo(np.int32) # Bounds of a 32-bit integer
            iinfo(min=-2147483648, max=2147483647, dtype=int32)
            >>> np.iinfo(np.int64) # Bounds of a 64-bit integer
            iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)

        If 64-bit integers are still too small the result may be cast to a
        floating point number. Floating point numbers offer a larger, but inexact,
        range of possible values.

            >>> np.power(100, 100, dtype=np.int64) # Incorrect even with 64-bit int
            0
            >>> np.power(100, 100, dtype=np.float64)
            1e+200

        Extended Precision
        ==================

        Python's floating-point numbers are usually 64-bit floating-point numbers,
        nearly equivalent to ``np.float64``. In some unusual situations it may be
        useful to use floating-point numbers with more precision. Whether this
        is possible in numpy depends on the hardware and on the development
        environment: specifically, x86 machines provide hardware floating-point
        with 80-bit precision, and while most C compilers provide this as their
        ``long double`` type, MSVC (standard for Windows builds) makes
        ``long double`` identical to ``double`` (64 bits). NumPy makes the
        compiler's ``long double`` available as ``np.longdouble`` (and
        ``np.clongdouble`` for the complex numbers). You can find out what your
        numpy provides with ``np.finfo(np.longdouble)``.

        NumPy does not provide a dtype with more precision than C's
        ``long double``\; in particular, the 128-bit IEEE quad precision
        data type (FORTRAN's ``REAL*16``\) is not available.

        For efficient memory alignment, ``np.longdouble`` is usually stored
        padded with zero bits, either to 96 or 128 bits. Which is more efficient
        depends on hardware and development environment; typically on 32-bit
```

```
    systems they are padded to 96 bits, while on 64-bit systems they are
    typically padded to 128 bits. ``np.longdouble`` is padded to the system
    default; ``np.float96`` and ``np.float128`` are provided for users who
    want specific padding. In spite of the names, ``np.float96`` and
    ``np.float128`` provide only as much precision as ``np.longdouble``,
    that is, 80 bits on most x86 machines and 64 bits in standard
    Windows builds.

    Be warned that even if ``np.longdouble`` offers more precision than
    python ``float``, it is easy to lose that extra precision, since
    python often forces values to pass through ``float``. For example,
    the ``%`` formatting operator requires its arguments to be converted
    to standard python types, and it is therefore impossible to preserve
    extended precision even if many decimal places are requested. It can
    be useful to test your code with the value
    ``1 + np.finfo(np.longdouble).eps``.

FILE
    /usr/local/lib/python3.7/dist-packages/numpy/doc/basics.py
```

```
help(np.doc)
```

```
Help on package numpy.doc in numpy:

NAME
    numpy.doc

DESCRIPTION
    Topical documentation
    =====================

    The following topics are available:

    - basics
    - broadcasting
    - byteswapping
    - constants
    - creation
    - dispatch
    - glossary
    - indexing
    - internals
    - misc
    - structured_arrays
    - subclassing
    - ufuncs

    You can view them by

    >>> help(np.doc.TOPIC)                                      #doctest: +SKIP

PACKAGE CONTENTS
```

```
        basics
        broadcasting
        byteswapping
        constants
        creation
        dispatch
        glossary
        indexing
        internals
        misc
        structured_arrays
        subclassing
        ufuncs

    DATA
        __all__ = ['basics', 'broadcasting', 'byteswapping', 'constants', 'cre...

    FILE
        /usr/local/lib/python3.7/dist-packages/numpy/doc/__init__.py
```

```
help(np.doc.indexing)
```

```
        a new array is extracted from the original (as a temporary) containing
        the values at 1, 1, 3, 1, then the value 1 is added to the temporary,
        and then the temporary is assigned back to the original array. Thus
        the value of the array at x[1]+1 is assigned to x[1] three times,
        rather than being incremented 3 times.

        Dealing with variable numbers of indices within programs
        ========================================================

        The index syntax is very powerful but limiting when dealing with
        a variable number of indices. For example, if you want to write
        a function that can handle arguments with various numbers of
        dimensions without having to write special case code for each
        number of possible dimensions, how can that be done? If one
        supplies to the index a tuple, the tuple will be interpreted
        as a list of indices. For example (using the previous definition
        for the array z): ::

         >>> indices = (1,1,1,1)
         >>> z[indices]
         40

        So one can use code to construct tuples of any number of indices
        and then use these within an index.

        Slices can be specified within programs by using the slice() function
        in Python. For example: ::

         >>> indices = (1,1,1,slice(0,2)) # same as [1,1,1,0:2]
         >>> z[indices]
         array([39, 40])
```

```
    Likewise, ellipsis can be specified by code by using the Ellipsis
    object: ::

     >>> indices = (1, Ellipsis, 1) # same as [1,...,1]
     >>> z[indices]
     array([[28, 31, 34],
            [37, 40, 43],
            [46, 49, 52]])

    For this reason it is possible to use the output from the np.nonzero()
    function directly as an index since it always returns a tuple of index
    arrays.

    Because the special treatment of tuples, they are not automatically
    converted to an array as a list would be. As an example: ::

     >>> z[[1,1,1,1]] # produces a large array
     array([[[[27, 28, 29],
              [30, 31, 32], ...
     >>> z[(1,1,1,1)] # returns a single value
     40

FILE
    /usr/local/lib/python3.7/dist-packages/numpy/doc/indexing.py
```

creating ND arrays in Numpy

```
a = np.array(43)
```

```
print(a)
```

```
43
```

```
type(a)
```

```
numpy.ndarray
```

```
a = np.array([1, 2, 3, 4,5])
print(a, a.ndim)
```

```
[1 2 3 4 5] 1
```

```
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(a)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
print(a.shape)
```

```
(3, 3)
```

```
a.ndim
```

```
2
```

3-D array

```
b = np.array([
    [[1, 2, 3], [4, 5, 6]],
    [[7, 8, 9], [5, 6, 8]]
    ])
```

```
print(b, b.ndim, b.shape)
```

```
[[[1 2 3]
  [4 5 6]]

 [[7 8 9]
  [5 6 8]]] 3 (2, 2, 3)
```

```
print(b.shape)
```

```
(2, 2, 3)
```

```
zarray = np.zeros(6)
print(zarray)
```

```
[0. 0. 0. 0. 0. 0.]
```

```
oarr = np.ones(10)
```

```
print(oarr)
```

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

```
arr= np.ones((3,3))
```

```
print(arr)
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

```
print(np.full((4,3),5))
```

```
[[5 5 5]
 [5 5 5]
 [5 5 5]
 [5 5 5]]
```

```
emp_array = np.empty((3,3))
```

```
print(emp_array)
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

```
print(np.arange(1, 10, 2))
```

```
[1 3 5 7 9]
```

```
eye = np.eye(5,5)
```

```
print(eye)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

```
# linspace(start, end, no.of point)

print( np.linspace(1,2,5) )
```

```
[1.   1.25 1.5  1.75 2.  ]
```

```
print(np.linspace(1, 10, 10))
```

```
print(np.linspace(1, 10, 20))
```

```
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
[ 1.          1.47368421  1.94736842  2.42105263  2.89473684  3.36842105
  3.84210526  4.31578947  4.78947368  5.26315789  5.73684211  6.21052632
  6.68421053  7.15789474  7.63157895  8.10526316  8.57894737  9.05263158
  9.52631579 10.        ]
```

```
print(np.linspace(1, 10, 20))
```

```
[ 1.          1.47368421  1.94736842  2.42105263  2.89473684  3.36842105
  3.84210526  4.31578947  4.78947368  5.26315789  5.73684211  6.21052632
  6.68421053  7.15789474  7.63157895  8.10526316  8.57894737  9.05263158
  9.52631579 10.        ]
```

```
np.diag([1, 2, 3, 4, 5, 6])
```

```
array([[1, 0, 0, 0, 0, 0],
       [0, 2, 0, 0, 0, 0],
       [0, 0, 3, 0, 0, 0],
       [0, 0, 0, 4, 0, 0],
       [0, 0, 0, 0, 5, 0],
       [0, 0, 0, 0, 0, 6]])
```

```
arr = np.array(range(10))
print(arr)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
arr.dtype
```

```
dtype('int64')
```

```
newarr = np.array(list(range(10)), dtype=np.float64)
print(newarr)
```

```
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

```
# astype()
newarr1 = arr.astype(np.float64)
print(newarr1)
print(newarr1.dtype)
```

```
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
float64
```

psuedo Random number generation

```
from numpy import random
```

```
np.random.rand(4)
```

```
array([0.09356568, 0.48216109, 0.32781421, 0.54736556])
```

```
print(np.random.randn(2, 3))
```

```
[[-0.22044329 -1.16976095 -0.3446816 ]
 [ 0.59559935 -0.52101786  0.16833513]]
```

```
np.random.normal(size=(4,4))
```

```
array([[ 0.48527846, -0.11078382, -0.17768691, -0.4147434 ],
       [-0.47109295, -0.44152462, -1.39785362, -1.02204234],
       [ 2.20989306, -1.14417865, -0.90617341,  0.183254  ],
       [-0.83221073, -0.97828883,  0.54261853,  0.54016326]])
```

```
from numpy.random import *
```

```
np.random.randn(10)
```

```
array([ 0.73063176,  1.5972896 , -0.83372617, -0.93117149, -0.6958524 ,
       -0.47548677, -0.0574924 , -1.43178657, -0.33259214,  0.5448802 ])
```

```
randn(20).reshape(2,10)
```

```
array([[ 0.31249114, -0.50943099,  1.45004943,  0.39819424,  2.73069774,
        -0.81182962,  1.26279963,  1.04340696,  0.10424755, -1.08000984],
       [ 1.71830181, -0.92418291, -0.57954779,  1.59682325, -0.40324019,
         0.90031864,  0.75532357,  0.27233552,  0.45391535, -0.99415003]])
```

Double-click (or enter) to edit

```
np.arange(20).reshape(5,4)
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

```
np.random.randint(100, 200, size=(3,2))
```

```
array([[185, 137],
       [191, 126],
       [122, 198]])
```

```
arr1 = np.array(np.arange(30))
```

```
print(arr1)
```

```
[ 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]
```

```
arr1[-1]
```

```
29
```

```
arr1[15]
```

```
15
```

```
b = arr1[1:25:2]
print(b)
```

```
[ 1  3  5  7  9 11 13 15 17 19 21 23]
```

iterate numpy arrays

```
arr = np.arange(12).reshape(3,4)
```

```
print(arr)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
for rows in arr:
  for cell in rows:
    print(cell)
```

```
0
1
```

```
2
3
4
5
6
7
8
9
10
11
```

```
arr
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
arr.flatten()
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
arr
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
for cell in arr.flatten():
  print(cell)
```

```
0
1
2
3
4
5
6
7
8
9
10
11
```

```
# nditer
arr
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
print(np.nditer(arr, order="F"))
```

```
<numpy.nditer object at 0x7f1667646f30>
```

```
for i in np.nditer(arr, order="F"):
  print(i)
```

```
0
4
8
1
5
9
2
6
10
3
7
11
```

```
# c-order  - row wise
# fortran order - column wisw
```

Booelan Indexing

```
arr = np.random.randint(0, 20, 15)
```

```
print(arr)
```

```
[14  3  5 18  1 15 12  1  4  1 16  7  9 15  6]
```

```
mask = (arr%2==0)
extract_arr = arr[mask]
print(extract_arr)
```

```
[14 18 12  4 16  6]
```

```
days = np.array(["sun", "mon", "tue", "wed", "thu", "fri", "sat", "sun", "thu", "thu"])
print(days)
```

```
['sun' 'mon' 'tue' 'wed' 'thu' 'fri' 'sat' 'sun' 'thu' 'thu']
```

```
data = np.random.randn(10, 4)
```

```
data
```

```
array([[-1.21791355, -0.65308789, -0.46895031, -1.028097  ],
       [-0.78141572, -0.8648698 , -0.25635438,  1.43806247],
       [ 0.30984319, -0.6198358 ,  0.00751043,  0.86510119],
       [-0.20896448, -0.48322657, -0.05616368,  0.98667775],
       [-0.20017865,  0.77252572,  0.84672281, -0.22453797],
       [-1.15337971, -0.77378018, -1.53094597,  1.23519966],
       [ 0.73164535,  0.27301396,  0.87768848,  1.75866354],
       [ 0.90935155, -0.87943324,  0.08659347, -0.19558607],
       [ 1.93349674, -0.53470479, -0.21838727, -1.05764244],
       [ 0.43548523, -1.09092546,  0.72611193, -1.25835348]])
```

```
data[days=="sun"]
```

```
array([[-1.26936818, -0.2788741 ,  1.04045226, -0.94330671],
       [-0.6334256 , -1.27734053,  1.20875501, -1.6060226 ],
       [-0.9070461 ,  0.22872972, -0.68107593,  1.0018648 ],
       [-0.33832597,  0.16003037,  0.72470853, -0.42790994],
       [ 0.75413883, -0.85953989,  0.20212739, -0.15359552],
       [ 0.57847137, -0.46160906, -0.66155744,  0.70762897],
       [ 0.15582821,  0.98838783, -0.2098291 ,  0.49363346],
       [ 0.05029215, -0.16836097,  0.10931018, -0.50337919]])
```

```
data[days=='sun',  2 : ]
```

```
array([[-0.15804703,  0.25685633],
       [-0.43577115,  0.24243365]])
```

```
# ['sun' 'mon' 'tue' 'wed' 'thu' 'fri' 'sat' 'sun' 'thu' 'thu']
cond = ((days=='sun') | (days=='thu'))
```

```
data[cond]
```

```
array([[ 1.81458822, -0.27302088, -0.15804703,  0.25685633],
       [-0.33832597,  0.16003037,  0.72470853, -0.42790994],
       [ 0.28374701, -0.08352875, -0.43577115,  0.24243365],
       [ 0.15582821,  0.98838783, -0.2098291 ,  0.49363346],
       [ 0.05029215, -0.16836097,  0.10931018, -0.50337919]])
```

```
((days=='sun') | (days=='thu'))
```

```
array([ True, False, False, False,  True, False, False,  True,  True,
        True])
```

```
data[data < 0] = 1
print(data)
```

```
[[1.         1.         1.         1.        ]
 [1.         1.         1.         1.43806247]
 [0.30984319 1.         0.00751043 0.86510119]
```

```
 [1.         1.         1.         0.98667775]
 [1.         0.77252572 0.84672281 1.        ]
 [1.         1.         1.         1.23519966]
 [0.73164535 0.27301396 0.87768848 1.75866354]
 [0.90935155 1.         0.08659347 1.        ]
 [1.93349674 1.         1.         1.        ]
 [0.43548523 1.         0.72611193 1.        ]]
```

```
# stacking
```

```
arr1 = np.arange(6).reshape(3,2)
print(arr1)
```

```
[[0 1]
 [2 3]
 [4 5]]
```

```
arr2 = np.arange(6,12).reshape(3,2)
```

```
print(arr2)
```

```
[[ 6  7]
 [ 8  9]
 [10 11]]
```

```
# vstack
```

```
np.vstack((arr1, arr2))
```

```
array([[ 0,  1],
       [ 2,  3],
       [ 4,  5],
       [ 6,  7],
       [ 8,  9],
       [10, 11]])
```

```
# hstack
```

```
np.hstack((arr1, arr2))
```

```
array([[ 0,  1,  6,  7],
       [ 2,  3,  8,  9],
       [ 4,  5, 10, 11]])
```

```
arr = np.arange(30).reshape(2, 15)
```

```
arr
```

```
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]])
```

```
print(np.vsplit(arr, 3))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/usr/local/lib/python3.7/dist-packages/numpy/lib/shape_base.py in split(ary,
indices_or_sections, axis)
    866     try:
--> 867         len(indices_or_sections)
    868     except TypeError:

TypeError: object of type 'int' has no len()

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
```

2 frames

```
<__array_function__ internals> in vsplit(*args, **kwargs)

<__array_function__ internals> in split(*args, **kwargs)

/usr/local/lib/python3.7/dist-packages/numpy/lib/shape_base.py in split(ary,
indices_or_sections, axis)
    871         if N % sections:
    872             raise ValueError(
--> 873                 'array split does not result in an equal division')
    874     return array_split(ary, indices_or_sections, axis)
    875

ValueError: array split does not result in an equal division
```

```
arr = np.arange(30).reshape(15, 2)
```

```
arr
```

```
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]])
```

```
a, b, c = np.vsplit(arr, 3)
print(a)
```

```
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
```

```
print(b)
```

```
[[10 11]
 [12 13]
 [14 15]
 [16 17]
 [18 19]]
```

0s completed at 10:46 PM