NumPy supports a much greater variety of numerical types than Python does. The following table shows different scalar data types defined in NumPy.

|  |  |
| --- | --- |
| **Sr.No.** | **Data Types & Description** |
| 1 | **bool\_ :** Boolean (True or False) stored as a byte |
| 2 | **int\_:** Default integer type (same as C long; normally either int64 or int32) |
| 3 | **Intc:** Identical to C int (normally int32 or int64) |
| 4 | **Intp:** Integer used for indexing (same as C ssize\_t; normally either int32 or int64) |
| 5 | **int8:** Byte (-128 to 127) |
| 6 | **int16:** Integer (-32768 to 32767) |
| 7 | **int32:** Integer (-2147483648 to 2147483647) |
| 8 | **int64:** Integer (-9223372036854775808 to 9223372036854775807) |
| 9 | **uint8:** Unsigned integer (0 to 255) |
| 10 | **uint16:** Unsigned integer (0 to 65535) |
| 11 | **uint32:**Unsigned integer (0 to 4294967295) |
| 12 | **uint64:** Unsigned integer (0 to 18446744073709551615) |
| 13 | **float\_:** Shorthand for float64 |
| 14 | **float16:** Half precision float: sign bit, 5 bits exponent, 10 bits mantissa |
| 15 | **float32:** Single precision float: sign bit, 8 bits exponent, 23 bits mantissa |
| 16 | **float64:** Double precision float: sign bit, 11 bits exponent, 52 bits mantissa |
| 17 | **complex\_ :** Shorthand for complex128 |
| 18 | **complex64:** Complex number, represented by two 32-bit floats (real and imaginary components) |
| 19 | **complex128:** Complex number, represented by two 64-bit floats (real and imaginary components) |

Each built-in data type has a character code that uniquely identifies it.

* **'b'** − boolean
* **'i'** − (signed) integer
* **'u'** − unsigned integer
* **'f'** − floating-point
* **'c'** − complex-floating point
* **'m'** − timedelta
* **'M'** − datetime
* **'O'** − (Python) objects
* **'S', 'a'** − (byte-)string
* **'U'** − Unicode
* **'V'** − raw data (void)

# NumPy - Array Attributes

ndarray.shape

This array attribute returns a tuple consisting of array dimensions. It can also be used to resize the array.

Example 1

import numpy as np

a = np.array([[1,2,3],[4,5,6]])

print(a.shape)

The output is as follows −

(2, 3)

Example 2

# this resizes the ndarray

import numpy as np

a = np.array([

[1,2,3],

[4,5,6]

])

a.shape = (3,2)

print(a)

The output is as follows −

[[1, 2]

[3, 4]

[5, 6]]

Example 3

NumPy also provides a reshape function to resize an array.

import numpy as np

a = np.array([[1,2,3],[4,5,6]])

b = a.reshape(3,2)

print b

The output is as follows −

[[1, 2]

[3, 4]

[5, 6]]

ndarray.ndim

This array attribute returns the number of array dimensions.

Example 1

# an array of evenly spaced numbers

import numpy as np

a = np.arange(24)

print a

The output is as follows −

[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]

Example 2

# this is one dimensional array

import numpy as np

a = np.arange(24)

a.ndim

# now reshape it

b = a.reshape(2,4,3)

print b

# b is having three dimensions

The output is as follows −

[[[ 0, 1, 2]

[ 3, 4, 5]

[ 6, 7, 8]

[ 9, 10, 11]]

[[12, 13, 14]

[15, 16, 17]

[18, 19, 20]

[21, 22, 23]]]

numpy.itemsize

This array attribute returns the length of each element of array in bytes.

Example 1

# dtype of array is int8 (1 byte)

import numpy as np

x = np.array([1,2,3,4,5], dtype = np.int8)

print x.itemsize

The output is as follows −

1

Example 2

# dtype of array is now float32 (4 bytes)

import numpy as np

x = np.array([1,2,3,4,5], dtype = np.float32)

print x.itemsize

The output is as follows −

4

# NumPy - Array Creation Routines

A new **ndarray** object can be constructed by any of the following array creation routines or using a low-level ndarray constructor.

numpy.empty

It creates an uninitialized array of specified shape and dtype. It uses the following constructor −

numpy.empty(shape, dtype = float, order = 'C')

The constructor takes the following parameters.

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **Shape:**Shape of an empty array in int or tuple of int |
| 2 | **Dtype:**Desired output data type. Optional |
| 3 | **Order:**'C' for C-style row-major array, 'F' for FORTRAN style column-major array |

Example

The following code shows an example of an empty array.

import numpy as np

x = np.empty([3,2], dtype = int)

print x

The output is as follows −

[[22649312 1701344351]

[1818321759 1885959276]

[16779776 156368896]]

**Note** − The elements in an array show random values as they are not initialized.

numpy.zeros

Returns a new array of specified size, filled with zeros.

numpy.zeros(shape, dtype = float, order = 'C')

The constructor takes the following parameters.

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **Shape:** Shape of an empty array in int or sequence of int |
| 2 | **Dtype:**Desired output data type. Optional |
| 3 | **Order:**'C' for C-style row-major array, 'F' for FORTRAN style column-major array |

Example 1

# array of five zeros. Default dtype is float

import numpy as np

x = np.zeros(5)

print x

The output is as follows −

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

Example 2

import numpy as np

x = np.zeros((5,), dtype = np.int)

print x

Now, the output would be as follows −

[0 0 0 0 0]

Example 3

# custom type

import numpy as np

x = np.zeros((2,2), dtype = [('x', 'i4'), ('y', 'i4')])

print x

It should produce the following output −

[[(0,0)(0,0)]

[(0,0)(0,0)]]

numpy.ones

Returns a new array of specified size and type, filled with ones.

numpy.ones(shape, dtype = None, order = 'C')

The constructor takes the following parameters.

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **Shape:**Shape of an empty array in int or tuple of int |
| 2 | **Dtype:**Desired output data type. Optional |
| 3 | **Order:**'C' for C-style row-major array, 'F' for FORTRAN style column-major array |

Example 1

# array of five ones. Default dtype is float

import numpy as np

x = np.ones(5)

print x

The output is as follows −

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

Example 2

import numpy as np

x = np.ones([2,2], dtype = int)

print x

Now, the output would be as follows −

[[1 1]

[1 1]]

# NumPy - Array From Existing Data

In this chapter, we will discuss how to create an array from existing data.

## numpy.asarray

This function is similar to numpy.array except for the fact that it has fewer parameters. This routine is useful for converting Python sequence into ndarray.

numpy.asarray(a, dtype = None, order = None)

The constructor takes the following parameters.

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **a:** Input data in any form such as list, list of tuples, tuples, tuple of tuples or tuple of lists |
| 2 | **Dtype:**By default, the data type of input data is applied to the resultant ndarray |
| 3 | **order**  C (row major) or F (column major). C is default |

The following examples show how you can use the **asarray** function.

### Example 1

# convert list to ndarray

import numpy as np

x = [1,2,3]

a = np.asarray(x)

print a

Its output would be as follows −

[1 2 3]

### Example 2

# dtype is set

import numpy as np

x = [1,2,3]

a = np.asarray(x, dtype = float)

print a

Now, the output would be as follows −

[ 1. 2. 3.]

### Example 3

# ndarray from tuple

import numpy as np

x = (1,2,3)

a = np.asarray(x)

print a

Its output would be −

[1 2 3]

### Example 4

# ndarray from list of tuples

import numpy as np

x = [(1,2,3),(4,5)]

a = np.asarray(x)

print a

Here, the output would be as follows −

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

## numpy.frombuffer

This function interprets a buffer as one-dimensional array. Any object that exposes the buffer interface is used as parameter to return an **ndarray**.

numpy.frombuffer(buffer, dtype = float, count = -1, offset = 0)

The constructor takes the following parameters.

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **buffer**  Any object that exposes buffer interface |
| 2 | **dtype**  Data type of returned ndarray. Defaults to float |
| 3 | **count**  The number of items to read, default -1 means all data |
| 4 | **offset**  The starting position to read from. Default is 0 |

### Example

The following examples demonstrate the use of **frombuffer** function.

import numpy as np

s = 'Hello World'

a = np.frombuffer(s, dtype = 'S1')

print a

Here is its output −

['H' 'e' 'l' 'l' 'o' ' ' 'W' 'o' 'r' 'l' 'd']

## numpy.fromiter

This function builds an **ndarray** object from any iterable object. A new one-dimensional array is returned by this function.

numpy.fromiter(iterable, dtype, count = -1)

Here, the constructor takes the following parameters.

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **iterable**  Any iterable object |
| 2 | **dtype**  Data type of resultant array |
| 3 | **count**  The number of items to be read from iterator. Default is -1 which means all data to be read |

The following examples show how to use the built-in **range()** function to return a list object. An iterator of this list is used to form an **ndarray** object.

### Example 1

# create list object using range function

import numpy as np

list = range(5)

print list

Its output is as follows −

[0, 1, 2, 3, 4]

### Example 2

# obtain iterator object from list

import numpy as np

list = range(5)

it = iter(list)

# use iterator to create ndarray

x = np.fromiter(it, dtype = float)

print x

Now, the output would be as follows −

[0. 1. 2. 3. 4.]

# NumPy - Array From Numerical Ranges

## numpy.arange

This function returns an **ndarray** object containing evenly spaced values within a given range. The format of the function is as follows −

numpy.arange(start, stop, step, dtype)

The constructor takes the following parameters.

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **start**  The start of an interval. If omitted, defaults to 0 |
| 2 | **stop**  The end of an interval (not including this number) |
| 3 | **step**  Spacing between values, default is 1 |
| 4 | **dtype**  Data type of resulting ndarray. If not given, data type of input is used |

The following examples show how you can use this function.

### Example 1

import numpy as np

x = np.arange(5)

print x

Its output would be as follows −

[0 1 2 3 4]

### Example 2

import numpy as np

# dtype set

x = np.arange(5, dtype = float)

print x

Here, the output would be −

[0. 1. 2. 3. 4.]

### Example 3

# start and stop parameters set

import numpy as np

x = np.arange(10,20,2)

print x

Its output is as follows −

[10 12 14 16 18]

## numpy.linspace

This function is similar to **arange()** function. In this function, instead of step size, the number of evenly spaced values between the interval is specified. The usage of this function is as follows −

numpy.linspace(start, stop, num, endpoint, retstep, dtype)

The constructor takes the following parameters.

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **start**  The starting value of the sequence |
| 2 | **stop**  The end value of the sequence, included in the sequence if endpoint set to true |
| 3 | **num**  The number of evenly spaced samples to be generated. Default is 50 |
| 4 | **endpoint**  True by default, hence the stop value is included in the sequence. If false, it is not included |
| 5 | **retstep**  If true, returns samples and step between the consecutive numbers |
| 6 | **dtype**  Data type of output **ndarray** |

The following examples demonstrate the use **linspace** function.

### Example 1

import numpy as np

x = np.linspace(10,20,5)

print x

Its output would be −

[10. 12.5 15. 17.5 20.]

### Example 2

# endpoint set to false

import numpy as np

x = np.linspace(10,20, 5, endpoint = False)

print x

The output would be −

[10. 12. 14. 16. 18.]

### Example 3

# find retstep value

import numpy as np

x = np.linspace(1,2,5, retstep = True)

print x

# retstep here is 0.25

Now, the output would be −

(array([ 1. , 1.25, 1.5 , 1.75, 2. ]), 0.25)

## numpy.logspace

This function returns an **ndarray** object that contains the numbers that are evenly spaced on a log scale. Start and stop endpoints of the scale are indices of the base, usually 10.

numpy.logspace(start, stop, num, endpoint, base, dtype)

Following parameters determine the output of **logspace** function.

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **start**  The starting point of the sequence is basestart |
| 2 | **stop**  The final value of sequence is basestop |
| 3 | **num**  The number of values between the range. Default is 50 |
| 4 | **endpoint**  If true, stop is the last value in the range |
| 5 | **base**  Base of log space, default is 10 |
| 6 | **dtype**  Data type of output array. If not given, it depends upon other input arguments |

The following examples will help you understand the **logspace** function.

### Example 1

import numpy as np

# default base is 10

a = np.logspace(1.0, 2.0, num = 10)

print a

Its output would be as follows −

[ 10. 12.91549665 16.68100537 21.5443469 27.82559402

35.93813664 46.41588834 59.94842503 77.42636827 100. ]

### Example 2

# set base of log space to 2

import numpy as np

a = np.logspace(1,10,num = 10, base = 2)

print a

Now, the output would be −

[ 2. 4. 8. 16. 32. 64. 128. 256. 512. 1024.]

# NumPy - Indexing & Slicing

Contents of ndarray object can be accessed and modified by indexing or slicing, just like Python's in-built container objects.

As mentioned earlier, items in ndarray object follows zero-based index. Three types of indexing methods are available − **field access, basic slicing** and **advanced indexing**.

Basic slicing is an extension of Python's basic concept of slicing to n dimensions. A Python slice object is constructed by giving **start, stop**, and **step** parameters to the built-in **slice** function. This slice object is passed to the array to extract a part of array.

## Example 1

import numpy as np

a = np.arange(10)

s = slice(2,7,2)

print a[s]

Its output is as follows −

[2 4 6]

In the above example, an **ndarray** object is prepared by **arange()** function. Then a slice object is defined with start, stop, and step values 2, 7, and 2 respectively. When this slice object is passed to the ndarray, a part of it starting with index 2 up to 7 with a step of 2 is sliced.

The same result can also be obtained by giving the slicing parameters separated by a colon : (start:stop:step) directly to the **ndarray** object.

## Example 2

import numpy as np

a = np.arange(10)

b = a[2:7:2]

print b

Here, we will get the same output −

[2 4 6]

If only one parameter is put, a single item corresponding to the index will be returned. If a : is inserted in front of it, all items from that index onwards will be extracted. If two parameters (with : between them) is used, items between the two indexes (not including the stop index) with default step one are sliced.

## Example 3

# slice single item

import numpy as np

a = np.arange(10)

b = a[5]

print b

Its output is as follows −

5

## Example 4

# slice items starting from index

import numpy as np

a = np.arange(10)

print a[2:]

Now, the output would be −

[2 3 4 5 6 7 8 9]

## Example 5

# slice items between indexes

import numpy as np

a = np.arange(10)

print a[2:5]

Here, the output would be −

[2 3 4]

The above description applies to multi-dimensional **ndarray** too.

## Example 6

import numpy as np

a = np.array([[1,2,3],[3,4,5],[4,5,6]])

print a

# slice items starting from index

print 'Now we will slice the array from the index a[1:]'

print a[1:]

The output is as follows −

[[1 2 3]

[3 4 5]

[4 5 6]]

Now we will slice the array from the index a[1:]

[[3 4 5]

[4 5 6]]

Slicing can also include ellipsis (…) to make a selection tuple of the same length as the dimension of an array. If ellipsis is used at the row position, it will return an ndarray comprising of items in rows.

## Example 7

# array to begin with

import numpy as np

a = np.array([[1,2,3],[3,4,5],[4,5,6]])

print 'Our array is:'

print a

print '\n'

# this returns array of items in the second column

print 'The items in the second column are:'

print a[...,1]

print '\n'

# Now we will slice all items from the second row

print 'The items in the second row are:'

print a[1,...]

print '\n'

# Now we will slice all items from column 1 onwards

print 'The items column 1 onwards are:'

print a[...,1:]

The output of this program is as follows −

Our array is:

[[1 2 3]

[3 4 5]

[4 5 6]]

The items in the second column are:

[2 4 5]

The items in the second row are:

[3 4 5]

The items column 1 onwards are:

[[2 3]

[4 5]

[5 6]]

# NumPy - Advanced Indexing

It is possible to make a selection from ndarray that is a non-tuple sequence, ndarray object of integer or Boolean data type, or a tuple with at least one item being a sequence object. Advanced indexing always returns a copy of the data. As against this, the slicing only presents a view.

There are two types of advanced indexing − **Integer** and **Boolean**.

## Integer Indexing

This mechanism helps in selecting any arbitrary item in an array based on its Ndimensional index. Each integer array represents the number of indexes into that dimension. When the index consists of as many integer arrays as the dimensions of the target ndarray, it becomes straightforward.

In the following example, one element of specified column from each row of ndarray object is selected. Hence, the row index contains all row numbers, and the column index specifies the element to be selected.

### Example 1

import numpy as np

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

y = x[[0,1,2], [0,1,0]]

print y

Its output would be as follows −

[1 4 5]

The selection includes elements at (0,0), (1,1) and (2,0) from the first array.

In the following example, elements placed at corners of a 4X3 array are selected. The row indices of selection are [0, 0] and [3,3] whereas the column indices are [0,2] and [0,2].

### Example 2

import numpy as np

x = np.array([[ 0, 1, 2],[ 3, 4, 5],[ 6, 7, 8],[ 9, 10, 11]])

print 'Our array is:'

print x

print '\n'

rows = np.array([[0,0],[3,3]])

cols = np.array([[0,2],[0,2]])

y = x[rows,cols]

print 'The corner elements of this array are:'

print y

The output of this program is as follows −

Our array is:

[[ 0 1 2]

[ 3 4 5]

[ 6 7 8]

[ 9 10 11]]

The corner elements of this array are:

[[ 0 2]

[ 9 11]]

The resultant selection is an ndarray object containing corner elements.

Advanced and basic indexing can be combined by using one slice (:) or ellipsis (…) with an index array. The following example uses slice for row and advanced index for column. The result is the same when slice is used for both. But advanced index results in copy and may have different memory layout.

### Example 3

import numpy as np

x = np.array([[ 0, 1, 2],[ 3, 4, 5],[ 6, 7, 8],[ 9, 10, 11]])

print 'Our array is:'

print x

print '\n'

# slicing

z = x[1:4,1:3]

print 'After slicing, our array becomes:'

print z

print '\n'

# using advanced index for column

y = x[1:4,[1,2]]

print 'Slicing using advanced index for column:'

print y

The output of this program would be as follows −

Our array is:

[[ 0 1 2]

[ 3 4 5]

[ 6 7 8]

[ 9 10 11]]

After slicing, our array becomes:

[[ 4 5]

[ 7 8]

[10 11]]

Slicing using advanced index for column:

[[ 4 5]

[ 7 8]

[10 11]]

## Boolean Array Indexing

This type of advanced indexing is used when the resultant object is meant to be the result of Boolean operations, such as comparison operators.

### Example 1

In this example, items greater than 5 are returned as a result of Boolean indexing.

import numpy as np

x = np.array([[ 0, 1, 2],[ 3, 4, 5],[ 6, 7, 8],[ 9, 10, 11]])

print 'Our array is:'

print x

print '\n'

# Now we will print the items greater than 5

print 'The items greater than 5 are:'

print x[x > 5]

The output of this program would be −

Our array is:

[[ 0 1 2]

[ 3 4 5]

[ 6 7 8]

[ 9 10 11]]

The items greater than 5 are:

[ 6 7 8 9 10 11]

### Example 2

In this example, NaN (Not a Number) elements are omitted by using ~ (complement operator).

import numpy as np

a = np.array([np.nan, 1,2,np.nan,3,4,5])

print a[~np.isnan(a)]

Its output would be −

[ 1. 2. 3. 4. 5.]

### Example 3

The following example shows how to filter out the non-complex elements from an array.

import numpy as np

a = np.array([1, 2+6j, 5, 3.5+5j])

print a[np.iscomplex(a)]

Here, the output is as follows −

[2.0+6.j 3.5+5.j]

# NumPy - Array Manipulation

Several routines are available in NumPy package for manipulation of elements in ndarray object. They can be classified into the following types −

## Changing Shape

|  |  |
| --- | --- |
| **Sr.No.** | **Shape & Description** |
| 1 | [reshape](https://www.tutorialspoint.com/numpy/numpy_reshape.htm): Gives a new shape to an array without changing its data |
| 2 | [flatten](https://www.tutorialspoint.com/numpy/numpy_ndarray_flatten.htm): Returns a copy of the array collapsed into one dimension |
| 4 | [ravel](https://www.tutorialspoint.com/numpy/numpy_ndarray_ravel.htm): Returns a contiguous flattened array |

# numpy.reshape

This function gives a new shape to an array without changing the data. It accepts the following parameters −

numpy.reshape(arr, newshape, order')

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **arr**  Array to be reshaped |
| 2 | **newshape**  int or tuple of int. New shape should be compatible to the original shape |
| 3 | **order**  'C' for C style, 'F' for Fortran style, 'A' means Fortran like order if an array is stored in Fortran-like contiguous memory, C style otherwise |

## Example

import numpy as np

a = np.arange(8)

print 'The original array:'

print a

print '\n'

b = a.reshape(4,2)

print 'The modified array:'

print b

Its output would be as follows −

The original array:

[0 1 2 3 4 5 6 7]

The modified array:

[[0 1]

[2 3]

[4 5]

[6 7]]

# numpy.ndarray.flatten

This function returns a copy of an array collapsed into one dimension. The function takes the following parameters.

ndarray.flatten(order)

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **order**  'C'− row major (default. 'F': column major 'A': flatten in column-major order, if a is Fortran contiguous in memory, row-major order otherwise 'K': flatten a in the order the elements occur in the memory |

## Example

import numpy as np

a = np.arange(8).reshape(2,4)

print 'The original array is:'

print a

print '\n'

# default is column-major

print 'The flattened array is:'

print a.flatten()

print '\n'

print 'The flattened array in F-style ordering:'

print a.flatten(order = 'F')

The output of the above program would be as follows −

The original array is:

[[0 1 2 3]

[4 5 6 7]]

The flattened array is:

[0 1 2 3 4 5 6 7]

The flattened array in F-style ordering:

[0 4 1 5 2 6 3 7]

# numpy.ravel

This function returns a flattened one-dimensional array. A copy is made only if needed. The returned array will have the same type as that of the input array. The function takes one parameter.

numpy.ravel(a, order)

The constructor takes the following parameters.

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **order**  'C': row major (default. 'F': column major 'A': flatten in column-major order, if a is Fortran contiguous in memory, row-major order otherwise 'K': flatten a in the order the elements occur in the memory |

## Example

import numpy as np

a = np.arange(8).reshape(2,4)

print 'The original array is:'

print a

print '\n'

print 'After applying ravel function:'

print a.ravel()

print '\n'

print 'Applying ravel function in F-style ordering:'

print a.ravel(order = 'F')

Its output would be as follows −

The original array is:

[[0 1 2 3]

[4 5 6 7]]

After applying ravel function:

[0 1 2 3 4 5 6 7]

Applying ravel function in F-style ordering:

[0 4 1 5 2 6 3 7]

## Transpose Operations

|  |  |
| --- | --- |
| **Sr.No.** | **Operation & Description** |
| 1 | [transpose](https://www.tutorialspoint.com/numpy/numpy_transpose.htm): Permutes the dimensions of an array |
| 2 | [ndarray :](https://www.tutorialspoint.com/numpy/numpy_ndarray_t.htm)Same as self.transpose() |
| 3 | [rollaxis](https://www.tutorialspoint.com/numpy/numpy_rollaxis.htm): Rolls the specified axis backwards |
| 4 | [swapaxes](https://www.tutorialspoint.com/numpy/numpy_swapaxes.htm): Interchanges the two axes of an array |

# numpy.transpose

This function permutes the dimension of the given array. It returns a view wherever possible. The function takes the following parameters.

numpy.transpose(arr, axes)

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **arr**  The array to be transposed |
| 2 | **axes**  List of ints, corresponding to the dimensions. By default, the dimensions are reversed |

## Example

import numpy as np

a = np.arange(12).reshape(3,4)

print 'The original array is:'

print a

print '\n'

print 'The transposed array is:'

print np.transpose(a)

Its output would be as follows −

The original array is:

[[ 0 1 2 3]

[ 4 5 6 7]

[ 8 9 10 11]]

The transposed array is:

[[ 0 4 8]

[ 1 5 9]

[ 2 6 10]

[ 3 7 11]]

# numpy.ndarray.T

This function belongs to **ndarray** class. It behaves similar to **numpy.transpose**.

## Example

import numpy as np

a = np.arange(12).reshape(3,4)

print 'The original array is:'

print a

print '\n'

print 'Array after applying the function:'

print a.T

The output of the above program would be −

The original array is:

[[ 0 1 2 3]

[ 4 5 6 7]

[ 8 9 10 11]]

Array after applying the function:

[[ 0 4 8]

[ 1 5 9]

[ 2 6 10]

[ 3 7 11]]

# numpy.rollaxis

This function rolls the specified axis backwards, until it lies in a specified position. The function takes three parameters.

numpy.rollaxis(arr, axis, start)

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **arr**  Input array |
| 2 | **axis**  Axis to roll backwards. The position of the other axes do not change relative to one another |
| 3 | **start**  Zero by default leading to the complete roll. Rolls until it reaches the specified position |

## Example

# It creates 3 dimensional ndarray

import numpy as np

a = np.arange(8).reshape(2,2,2)

print 'The original array:'

print a

print '\n'

# to roll axis-2 to axis-0 (along width to along depth)

print 'After applying rollaxis function:'

print np.rollaxis(a,2)

# to roll axis 0 to 1 (along width to height)

print '\n'

print 'After applying rollaxis function:'

print np.rollaxis(a,2,1)

Its output is as follows −

The original array:

[[[0 1]

[2 3]]

[[4 5]

[6 7]]]

After applying rollaxis function:

[[[0 2]

[4 6]]

[[1 3]

[5 7]]]

After applying rollaxis function:

[[[0 2]

[1 3]]

[[4 6]

[5 7]]]

## Numpy Axis Directions

**Axis 0 (Direction along Rows)**– Axis 0 is called the first axis of the Numpy array. This axis 0 runs vertically downward along the rows of Numpy multidimensional arrays, i.e., performs **column-wise** operations.

**Axis 1 (Direction along with** columns) – Axis 1 is called the second axis of multidimensional Numpy arrays. As a result, Axis 1 sums horizontally along with the columns of the arrays. It performs row-wiseoperations.

# numpy.swapaxes

This function interchanges the two axes of an array. For NumPy versions after 1.10, a view of the swapped array is returned. The function takes the following parameters.

numpy.swapaxes(arr, axis1, axis2)

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **arr**  Input array whose axes are to be swapped |
| 2 | **axis1**  An int corresponding to the first axis |
| 3 | **axis2**  An int corresponding to the second axis |

## Example

# It creates a 3 dimensional ndarray

import numpy as np

a = np.arange(8).reshape(2,2,2)

print 'The original array:'

print a

print '\n'

# now swap numbers between axis 0 (along depth) and axis 2 (along width)

print 'The array after applying the swapaxes function:'

print np.swapaxes(a, 2, 0)

Its output would be as follows −

The original array:

[[[0 1]

[2 3]]

[[4 5]

[6 7]]]

The array after applying the swapaxes function:

[[[0 4]

[2 6]]

[[1 5]

[3 7]]]

## Changing Dimensions

|  |  |
| --- | --- |
| **Sr.No.** | **Dimension & Description** |
| 1 | [expand\_dims](https://www.tutorialspoint.com/numpy/numpy_expand_dims.htm): Expands the shape of an array |
| 2 | [squeeze](https://www.tutorialspoint.com/numpy/numpy_squeeze.htm): Removes single-dimensional entries from the shape of an array |

# numpy.expand\_dims

This function expands the array by inserting a new axis at the specified position. Two parameters are required by this function.

numpy.expand\_dims(arr, axis)

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **Arr:**Input array |
| 2 | **Axis:** Position where new axis to be inserted |

## Example

import numpy as np

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

print 'Array x:'

print x

print '\n'

y = np.expand\_dims(x, axis = 0)

print 'Array y:'

print y

print '\n'

print 'The shape of X and Y array:'

print x.shape, y.shape

print '\n'

# insert axis at position 1

y = np.expand\_dims(x, axis = 1)

print 'Array Y after inserting axis at position 1:'

print y

print '\n'

print 'x.ndim and y.ndim:'

print x.ndim,y.ndim

print '\n'

print 'x.shape and y.shape:'

print x.shape, y.shape

The output of the above program would be as follows −

Array x:

[[1 2]

[3 4]]

Array y:

[[[1 2]

[3 4]]]

The shape of X and Y array:

(2, 2) (1, 2, 2)

Array Y after inserting axis at position 1:

[[[1 2]]

[[3 4]]]

x.ndim and y.ndim:

2 3

x.shape and y.shape:

(2, 2) (2, 1, 2)

# numpy.squeeze

This function removes one-dimensional entry from the shape of the given array. Two parameters are required for this function.

numpy.squeeze(arr, axis)

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **arr**  Input array |
| 2 | **axis**  int or tuple of int. selects a subset of single dimensional entries in the shape |

## Example

[Live Demo](http://tpcg.io/npt8hl)

import numpy as np

x = np.arange(9).reshape(1,3,3)

print 'Array X:'

print x

print '\n'

y = np.squeeze(x)

print 'Array Y:'

print y

print '\n'

print 'The shapes of X and Y array:'

print x.shape, y.shape

Its output is as follows −

Array X:

[[[0 1 2]

[3 4 5]

[6 7 8]]]

Array Y:

[[0 1 2]

[3 4 5]

[6 7 8]]

The shapes of X and Y array:

(1, 3, 3) (3, 3)

## Joining Arrays

|  |  |
| --- | --- |
| **Sr.No.** | **Array & Description** |
| 1 | [concatenate](https://www.tutorialspoint.com/numpy/numpy_concatenate.htm): Joins a sequence of arrays along an existing axis |
| 2 | [stack](https://www.tutorialspoint.com/numpy/numpy_stack.htm): Joins a sequence of arrays along a new axis |
| 3 | [hstack](https://www.tutorialspoint.com/numpy/numpy_hstack.htm): Stacks arrays in sequence horizontally (column wise) |
| 4 | [vstack](https://www.tutorialspoint.com/numpy/numpy_vstack.htm): Stacks arrays in sequence vertically (row wise) |

# numpy.concatenate

Concatenation refers to joining. This function is used to join two or more arrays of the same shape along a specified axis. The function takes the following parameters.

numpy.concatenate((a1, a2, ...), axis)

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **a1,a2..**  Sequence of arrays of the same type |
| 2 | **axis**  Axis along which arrays have to be joined. Default is 0 |

## Example

[Live Demo](http://tpcg.io/KdgLOG)

import numpy as np

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

print 'First array:'

print a

print '\n'

b = np.array([[5,6],[7,8]])

print 'Second array:'

print b

print '\n'

# both the arrays are of same dimensions

print 'Joining the two arrays along axis 0:'

print np.concatenate((a,b))

print '\n'

print 'Joining the two arrays along axis 1:'

print np.concatenate((a,b),axis = 1)

Its output is as follows −

First array:

[[1 2]

[3 4]]

Second array:

[[5 6]

[7 8]]

Joining the two arrays along axis 0:

[[1 2]

[3 4]

[5 6]

[7 8]]

Joining the two arrays along axis 1:

[[1 2 5 6]

[3 4 7 8]]

# numpy.stack

This function joins the sequence of arrays along a new axis. This function has been added since NumPy version 1.10.0. Following parameters need to be provided.

**Note** − This function is available in *version 1.10.0* onwards.

numpy.stack(arrays, axis)

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **arrays**  Sequence of arrays of the same shape |
| 2 | **axis**  Axis in the resultant array along which the input arrays are stacked |

## Example

import numpy as np

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

print 'First Array:'

print a

print '\n'

b = np.array([[5,6],[7,8]])

print 'Second Array:'

print b

print '\n'

print 'Stack the two arrays along axis 0:'

print np.stack((a,b),0)

print '\n'

print 'Stack the two arrays along axis 1:'

print np.stack((a,b),1)

It should produce the following output −

First array:

[[1 2]

[3 4]]

Second array:

[[5 6]

[7 8]]

Stack the two arrays along axis 0:

[[[1 2]

[3 4]]

[[5 6]

[7 8]]]

Stack the two arrays along axis 1:

[[[1 2]

[5 6]]

[[3 4]

[7 8]]]

# numpy.hstack

Variants of numpy.stack function to stack so as to make a single array horizontally.

## Example

import numpy as np

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

print 'First array:'

print a

print '\n'

b = np.array([[5,6],[7,8]])

print 'Second array:'

print b

print '\n'

print 'Horizontal stacking:'

c = np.hstack((a,b))

print c

print '\n'

It would produce the following output −

First array:

[[1 2]

[3 4]]

Second array:

[[5 6]

[7 8]]

Horizontal stacking:

[[1 2 5 6]

[3 4 7 8]]

# numpy.vstack

Variants of numpy.stack function to stack so as to make a single array vertically.

## Example

import numpy as np

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

print 'First array:'

print a

print '\n'

b = np.array([[5,6],[7,8]])

print 'Second array:'

print b

print '\n'

print 'Vertical stacking:'

c = np.vstack((a,b))

print c

It would produce the following output −

First array:

[[1 2]

[3 4]]

Second array:

[[5 6]

[7 8]]

Vertical stacking:

[[1 2]

[3 4]

[5 6]

[7 8]]

## Splitting Arrays

|  |  |
| --- | --- |
| **Sr.No.** | **Array & Description** |
| 1 | [split](https://www.tutorialspoint.com/numpy/numpy_split.htm): Splits an array into multiple sub-arrays |
| 2 | [hsplit](https://www.tutorialspoint.com/numpy/numpy_hsplit.htm): Splits an array into multiple sub-arrays horizontally (column-wise) |
| 3 | [vsplit](https://www.tutorialspoint.com/numpy/numpy_vsplit.htm): Splits an array into multiple sub-arrays vertically (row-wise) |

# numpy.split

This function divides the array into subarrays along a specified axis. The function takes three parameters.

numpy.split(ary, indices\_or\_sections, axis)

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **ary**  Input array to be split |
| 2 | **indices\_or\_sections**  Can be an integer, indicating the number of equal sized subarrays to be created from the input array. If this parameter is a 1-D array, the entries indicate the points at which a new subarray is to be created. |
| 3 | **axis**  Default is 0 |

## Example

[Live Demo](http://tpcg.io/V5kdSX)

import numpy as np

a = np.arange(9)

print 'First array:'

print a

print '\n'

print 'Split the array in 3 equal-sized subarrays:'

b = np.split(a,3)

print b

print '\n'

print 'Split the array at positions indicated in 1-D array:'

b = np.split(a,[4,7])

print b

Its output is as follows −

First array:

[0 1 2 3 4 5 6 7 8]

Split the array in 3 equal-sized subarrays:

[array([0, 1, 2]), array([3, 4, 5]), array([6, 7, 8])]

Split the array at positions indicated in 1-D array:

[array([0, 1, 2, 3]), array([4, 5, 6]), array([7, 8])]

# numpy.hsplit

The numpy.hsplit is a special case of split() function where axis is 1 indicating a horizontal split regardless of the dimension of the input array.

## Example

[Live Demo](http://tpcg.io/10SVIf)

import numpy as np

a = np.arange(16).reshape(4,4)

print 'First array:'

print a

print '\n'

print 'Horizontal splitting:'

b = np.hsplit(a,2)

print b

print '\n'

Its output would be as follows −

First array:

[[ 0 1 2 3]

[ 4 5 6 7]

[ 8 9 10 11]

[12 13 14 15]]

Horizontal splitting:

[array([[ 0, 1],

[ 4, 5],

[ 8, 9],

[12, 13]]), array([[ 2, 3],

[ 6, 7],

[10, 11],

[14, 15]])]

# numpy.vsplit

numpy.vsplit is a special case of split() function where axis is 1 indicating a vertical split regardless of the dimension of the input array. The following example makes this clear.

## Example

[Live Demo](http://tpcg.io/bPtVav)

import numpy as np

a = np.arange(16).reshape(4,4)

print 'First array:'

print a

print '\n'

print 'Vertical splitting:'

b = np.vsplit(a,2)

print b

Its output would be as follows −

First array:

[[ 0 1 2 3]

[ 4 5 6 7]

[ 8 9 10 11]

[12 13 14 15]]

Vertical splitting:

[array([[0, 1, 2, 3],

[4, 5, 6, 7]]), array([[ 8, 9, 10, 11],

[12, 13, 14, 15]])]

## Adding / Removing Elements

|  |  |
| --- | --- |
| **Sr.No.** | **Element & Description** |
| 1 | [resize](https://www.tutorialspoint.com/numpy/numpy_resize.htm): Returns a new array with the specified shape |
| 2 | [append](https://www.tutorialspoint.com/numpy/numpy_append.htm): Appends the values to the end of an array |
| 3 | [insert](https://www.tutorialspoint.com/numpy/numpy_insert.htm): Inserts the values along the given axis before the given indices |
| 4 | [delete](https://www.tutorialspoint.com/numpy/numpy_delete.htm): Returns a new array with sub-arrays along an axis deleted |
| 5 | [unique](https://www.tutorialspoint.com/numpy/numpy_unique.htm): Finds the unique elements of an array |

# numpy.resize

This function returns a new array with the specified size. If the new size is greater than the original, the repeated copies of entries in the original are contained. The function takes the following parameters.

numpy.resize(arr, shape)

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **arr**  Input array to be resized |
| 2 | **shape**  New shape of the resulting array |

## Example

[Live Demo](http://tpcg.io/Pa8Dr2)

import numpy as np

a = np.array([[1,2,3],[4,5,6]])

print 'First array:'

print a

print '\n'

print 'The shape of first array:'

print a.shape

print '\n'

b = np.resize(a, (3,2))

print 'Second array:'

print b

print '\n'

print 'The shape of second array:'

print b.shape

print '\n'

# Observe that first row of a is repeated in b since size is bigger

print 'Resize the second array:'

b = np.resize(a,(3,3))

print b

The above program will produce the following output −

First array:

[[1 2 3]

[4 5 6]]

The shape of first array:

(2, 3)

Second array:

[[1 2]

[3 4]

[5 6]]

The shape of second array:

(3, 2)

Resize the second array:

[[1 2 3]

[4 5 6]

[1 2 3]]

# numpy.append

This function adds values at the end of an input array. The append operation is not inplace, a new array is allocated. Also the dimensions of the input arrays must match otherwise ValueError will be generated.

The function takes the following parameters.

numpy.append(arr, values, axis)

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **arr**  Input array |
| 2 | **values**  To be appended to arr. It must be of the same shape as of arr (excluding axis of appending) |
| 3 | **axis**  The axis along which append operation is to be done. If not given, both parameters are flattened |

## Example

[Live Demo](http://tpcg.io/qt7a0X)

import numpy as np

a = np.array([[1,2,3],[4,5,6]])

print 'First array:'

print a

print '\n'

print 'Append elements to array:'

print np.append(a, [7,8,9])

print '\n'

print 'Append elements along axis 0:'

print np.append(a, [[7,8,9]],axis = 0)

print '\n'

print 'Append elements along axis 1:'

print np.append(a, [[5,5,5],[7,8,9]],axis = 1)

Its output would be as follows −

First array:

[[1 2 3]

[4 5 6]]

Append elements to array:

[1 2 3 4 5 6 7 8 9]

Append elements along axis 0:

[[1 2 3]

[4 5 6]

[7 8 9]]

Append elements along axis 1:

[[1 2 3 5 5 5]

[4 5 6 7 8 9]]

# numpy.insert

This function inserts values in the input array along the given axis and before the given index. If the type of values is converted to be inserted, it is different from the input array. Insertion is not done in place and the function returns a new array. Also, if the axis is not mentioned, the input array is flattened.

The insert() function takes the following parameters −

numpy.insert(arr, obj, values, axis)

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **arr**  Input array |
| 2 | **obj**  The index before which insertion is to be made |
| 3 | **values**  The array of values to be inserted |
| 4 | **axis**  The axis along which to insert. If not given, the input array is flattened |

## Example

[Live Demo](http://tpcg.io/YUsMc7)

import numpy as np

a = np.array([[1,2],[3,4],[5,6]])

print 'First array:'

print a

print '\n'

print 'Axis parameter not passed. The input array is flattened before insertion.'

print np.insert(a,3,[11,12])

print '\n'

print 'Axis parameter passed. The values array is broadcast to match input array.'

print 'Broadcast along axis 0:'

print np.insert(a,1,[11],axis = 0)

print '\n'

print 'Broadcast along axis 1:'

print np.insert(a,1,11,axis = 1)

Its output would be as follows −

First array:

[[1 2]

[3 4]

[5 6]]

Axis parameter not passed. The input array is flattened before insertion.

[ 1 2 3 11 12 4 5 6]

Axis parameter passed. The values array is broadcast to match input array.

Broadcast along axis 0:

[[ 1 2]

[11 11]

[ 3 4]

[ 5 6]]

Broadcast along axis 1:

[[ 1 11 2]

[ 3 11 4]

[ 5 11 6]]

# numpy.delete

This function returns a new array with the specified subarray deleted from the input array. As in case of insert() function, if the axis parameter is not used, the input array is flattened. The function takes the following parameters −

Numpy.delete(arr, obj, axis)

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **arr**  Input array |
| 2 | **obj**  Can be a slice, an integer or array of integers, indicating the subarray to be deleted from the input array |
| 3 | **axis**  The axis along which to delete the given subarray. If not given, arr is flattened |

## Example

[Live Demo](http://tpcg.io/cdXYS9)

import numpy as np

a = np.arange(12).reshape(3,4)

print 'First array:'

print a

print '\n'

print 'Array flattened before delete operation as axis not used:'

print np.delete(a,5)

print '\n'

print 'Column 2 deleted:'

print np.delete(a,1,axis = 1)

print '\n'

print 'A slice containing alternate values from array deleted:'

a = np.array([1,2,3,4,5,6,7,8,9,10])

print np.delete(a, np.s\_[::2])

Its output would be as follows −

First array:

[[ 0 1 2 3]

[ 4 5 6 7]

[ 8 9 10 11]]

Array flattened before delete operation as axis not used:

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

Column 2 deleted:

[[ 0 2 3]

[ 4 6 7]

[ 8 10 11]]

A slice containing alternate values from array deleted:

[ 2 4 6 8 10]

# numpy.unique

This function returns an array of unique elements in the input array. The function can be able to return a tuple of array of unique vales and an array of associated indices. Nature of the indices depend upon the type of return parameter in the function call.

numpy.unique(arr, return\_index, return\_inverse, return\_counts)

Where,

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **arr**  The input array. Will be flattened if not 1-D array |
| 2 | **return\_index**  If True, returns the indices of elements in the input array |
| 3 | **return\_inverse**  If True, returns the indices of unique array, which can be used to reconstruct the input array |
| 4 | **return\_counts**  If True, returns the number of times the element in unique array appears in the original array |

## Example

[Live Demo](http://tpcg.io/svxOXe)

import numpy as np

a = np.array([5,2,6,2,7,5,6,8,2,9])

print 'First array:'

print a

print '\n'

print 'Unique values of first array:'

u = np.unique(a)

print u

print '\n'

print 'Unique array and Indices array:'

u,indices = np.unique(a, return\_index = True)

print indices

print '\n'

print 'We can see each number corresponds to index in original array:'

print a

print '\n'

print 'Indices of unique array:'

u,indices = np.unique(a,return\_inverse = True)

print u

print '\n'

print 'Indices are:'

print indices

print '\n'

print 'Reconstruct the original array using indices:'

print u[indices]

print '\n'

print 'Return the count of repetitions of unique elements:'

u,indices = np.unique(a,return\_counts = True)

print u

print indices

Its output is as follows −

First array:

[5 2 6 2 7 5 6 8 2 9]

Unique values of first array:

[2 5 6 7 8 9]

Unique array and Indices array:

[1 0 2 4 7 9]

We can see each number corresponds to index in original array:

[5 2 6 2 7 5 6 8 2 9]

Indices of unique array:

[2 5 6 7 8 9]

Indices are:

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

Reconstruct the original array using indices:

[5 2 6 2 7 5 6 8 2 9]

Return the count of repetitions of unique elements:

[2 5 6 7 8 9]

[3 2 2 1 1 1]