Unsigned 8bits uchar 0~255

Mat: CV\_8UC1, CV\_8UC2, CV\_8UC3, CV\_8UC4

Signed 8bits char -128~127

Mat: CV\_8SC1，CV\_8SC2，CV\_8SC3，CV\_8SC4

Unsigned 16bits ushort 0~65535

Mat: CV\_16UC1，CV\_16UC2，CV\_16UC3，CV\_16UC4

Signed 16bits short -32768~32767

Mat: CV\_16SC1，CV\_16SC2，CV\_16SC3，CV\_16SC4

Signed 32bits int -2147483648~2147483647

Mat: CV\_32SC1，CV\_32SC2，CV\_32SC3，CV\_32SC4

Float 32bits float -1.18\*10-38~3.40\*10-38

Mat: CV\_32FC1，CV\_32FC2，CV\_32FC3，CV\_32FC4

Double 64bits double

Mat: CV\_64FC1，CV\_64FC2，CV\_64FC3，CV\_64FC4

zip() in Python

The purpose of zip() is to**map the similar index of multiple containers** so that they can be used just using as single entity.

***Syntax :*** *zip(\*iterators)****Parameters :*** *Python iterables or containers ( list, string etc )****Return Value :*** *Returns a single iterator object, having mapped values from all the  
containers.*

|  |
| --- |
| # Python code to demonstrate the working of  # zip()    # initializing lists  name = [ "Manjeet", "Nikhil", "Shambhavi", "Astha" ]  roll\_no = [ 4, 1, 3, 2 ]  marks = [ 40, 50, 60, 70 ]    # using zip() to map values  mapped = zip(name, roll\_no, marks)    # converting values to print as set  mapped = set(mapped)    # printing resultant values  print ("The zipped result is : ",end="")  print (mapped) |

Output:

The zipped result is : {('Shambhavi', 3, 60), ('Astha', 2, 70),

('Manjeet', 4, 40), ('Nikhil', 1, 50)}

**How to unzip?**  
Unzipping means converting the zipped values back to the individual self as they were. This is done with the help of “**\***” operator.

|  |
| --- |
| # Python code to demonstrate the working of  # unzip    # initializing lists    name = [ "Manjeet", "Nikhil", "Shambhavi", "Astha" ]  roll\_no = [ 4, 1, 3, 2 ]  marks = [ 40, 50, 60, 70 ]    # using zip() to map values  mapped = zip(name, roll\_no, marks)    # converting values to print as list  mapped = list(mapped)    # printing resultant values  print ("The zipped result is : ",end="")  print (mapped)    print("\n")    # unzipping values  namz, roll\_noz, marksz = zip(\*mapped)    print ("The unzipped result: \n",end="")    # printing initial lists  print ("The name list is : ",end="")  print (namz)    print ("The roll\_no list is : ",end="")  print (roll\_noz)    print ("The marks list is : ",end="")  print (marksz) |

Output:

The zipped result is : [('Manjeet', 4, 40), ('Nikhil', 1, 50),

('Shambhavi', 3, 60), ('Astha', 2, 70)]

The unzipped result:

The name list is : ('Manjeet', 'Nikhil', 'Shambhavi', 'Astha')

The roll\_no list is : (4, 1, 3, 2)

The marks list is : (40, 50, 60, 70)

**Practical Applications :** There are many possible applications that can be said to be exected using zip, be it **student database or scorecard** or any other utility that requires mapping of groups. A small example of scorecard is demonstrated below.

|  |
| --- |
| # Python code to demonstrate the application of  # zip()    # initializing list of players.  players = [ "Sachin", "Sehwag", "Gambhir", "Dravid", "Raina" ]    # initializing their scores  scores = [100, 15, 17, 28, 43 ]    # printing players and scores.  for pl, sc in zip(players, scores):      print ("Player :  %s     Score : %d" %(pl, sc)) |

Output:

Player : Sachin Score : 100

Player : Sehwag Score : 15

Player : Gambhir Score : 17

Player : Dravid Score : 28

Player : Raina Score : 43

# What is numpy.newaxis and when to use it.

[](https://medium.com/@ian.dzindo01?source=post_page-----8cb61c7ed6ae----------------------)

[Ian Dzindo](https://medium.com/@ian.dzindo01?source=post_page-----8cb61c7ed6ae----------------------)

Follow

[Apr 16, 2018](https://medium.com/@ian.dzindo01/what-is-numpy-newaxis-and-when-to-use-it-8cb61c7ed6ae?source=post_page-----8cb61c7ed6ae----------------------) · 3 min read

You might have encountered the np.newaxis expression here and there. Even though it might seem irrelevant at first it is a very powerful and valuable function, which finds many uses in different situations and settings.

In this article, I’m going to explain to you exactly what it does and show you when and how it should be implemented.

https://miro.medium.com/max/60/1*alRCFYpDDeuHYOeJH_Upeg.jpeg?q=20



Picture from pixelperfect-photo.com. Some rights reserved.

# What it does:

Simply put, the **newaxis** expression is used to **increase the dimension** of the existing array by one more dimension, when used once. Thus,

* **1D** array will become **2D** array
* **2D** array will become **3D** array
* **3D** array will become **4D** array

and so on. It’s role is quite simple and yet there are plenty of problems where it’s a life saver.

# When to use it:

**Scenario 1**: Np.newaxis comes in very handy when you want to explicitly convert an 1D array to either a row vector or a column vector.

## Example:

# 1D array  
In [7]: arr = np.arange(4)  
In [8]: arr.shape  
Out[8]: (4,)  
  
# make it as row vector by inserting an axis along first dimension  
In [9]: row\_vec = arr[np.newaxis, :]  
In [10]: row\_vec.shape  
Out[10]: (1, 4)  
  
# make it as column vector by inserting an axis along second dimension  
In [11]: col\_vec = arr[:, np.newaxis]  
In [12]: col\_vec.shape  
Out[12]: (4, 1)

**Scenario 2**: When we want to make use of [numpy broadcasting](https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html) as part of some operation, for instance while doing addition of some arrays. The term numpy broadcasting describes how numpy treats arrays with different shapes during arithmetic operation. The smaller array, subject to some constraints, is “broadcast” across the larger arrays so that they have compatible shapes.

## Example:

Let’s say you want to add the two following arrays:

x1 = np.array([1, 2, 3, 4, 5])  
x2 = np.array([5, 4, 3])

If you tried to add these like this, you would get a ValueError, like this:

ValueError: operands could not be broadcast together with shapes (5,) (3,)

Here you will first have to use np.newaxis to increase the dimension of the smaller array, so Numpy can do it’s thing.

In [2]: x1\_new = x1[:, np.newaxis]  
# now, the shape of x1\_new is (5, 1)  
# array([[1],  
# [2],  
# [3],  
# [4],  
# [5]])

And then:

In [3]: x1\_new + x2  
Out[3]:  
array([[ 6, 5, 4],  
 [ 7, 6, 5],  
 [ 8, 7, 6],  
 [ 9, 8, 7],  
 [10, 9, 8]])

Alternatively, you can add a new axis to x2, but in the end you actually get the same result.

**Scenario 3**: You can use np.newaxis more than once to promote the array to higher dimensions.

## Example:

In [124]: arr = np.arange(5\*5).reshape(5,5)  
  
In [125]: arr.shape  
Out[125]: (5, 5)  
  
# promoting 2D array to a 5D array  
In [126]: arr\_5D = arr[np.newaxis, ..., np.newaxis, np.newaxis]  
  
In [127]: arr\_5D.shape  
Out[127]: (1, 5, 5, 1, 1)

## Np.newaxis vs np.reshape:

Np.newaxis uses the slicing operator to recreate the array, while np.reshape reshapes the array to the desired layout (assuming that the dimensions match, this is **must** for a reshape to happen).

## Example:

In [13]: A = np.ones((3,4,5,6))  
In [14]: B = np.ones((4,6))  
In [15]: (A + B[:, np.newaxis, :]).shape  
Out[15]: (3, 4, 5, 6)

In the above example, we inserted a temporary axis between the first and second axes of B . A missing axis is filled-in here using np.newaxis to make the [broadcasting](https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html) operation work.

## Note:

Np.newaxis and None are, in fact, the same objects. You can use one instead of the other:

In [13]: np.newaxis is None  
Out[13]: True

Calculate angle between two vectors

**ote**: all of the other answers here will fail if the two vectors have either the same direction (ex, (1, 0, 0), (1, 0, 0)) or opposite directions (ex, (-1, 0, 0), (1, 0, 0)).

Here is a function which will correctly handle these cases:

import numpy as np

def unit\_vector(vector):

""" Returns the unit vector of the vector. """

return vector / np.linalg.norm(vector)

def angle\_between(v1, v2):

""" Returns the angle in radians between vectors 'v1' and 'v2'::

>>> angle\_between((1, 0, 0), (0, 1, 0))

1.5707963267948966

>>> angle\_between((1, 0, 0), (1, 0, 0))

0.0

>>> angle\_between((1, 0, 0), (-1, 0, 0))

3.141592653589793

"""

v1\_u = unit\_vector(v1)

v2\_u = unit\_vector(v2)

return np.arccos(np.clip(np.dot(v1\_u, v2\_u), -1.0, 1.0))

# numpy.clip[¶](https://docs.scipy.org/doc/numpy/reference/generated/numpy.clip.html#numpy-clip)

**numpy.clip(**a**,**a\_min**,**a\_max**,**out=None**,**\*\*kwargs**)**[**[source]**](https://github.com/numpy/numpy/blob/v1.17.0/numpy/core/fromnumeric.py#L1974-L2037)

Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of [0, 1] is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Equivalent to but faster than np.maximum(a\_min, np.minimum(a, a\_max)). No check is performed to ensure a\_min < a\_max.

|  |  |
| --- | --- |
| **Parameters:** | **a : *array\_like***  Array containing elements to clip.  **a\_min : *scalar or array\_like or*None**  Minimum value. If None, clipping is not performed on lower interval edge. Not more than one of a\_min and a\_max may be None.  **a\_max : *scalar or array\_like or*None**  Maximum value. If None, clipping is not performed on upper interval edge. Not more than one of a\_min and a\_max may be None. If a\_min or a\_max are array\_like, then the three arrays will be broadcasted to match their shapes.  **out : *ndarray, optional***  The results will be placed in this array. It may be the input array for in-place clipping. out must be of the right shape to hold the output. Its type is preserved.  **\*\*kwargs**  For other keyword-only arguments, see the [ufunc docs](https://docs.scipy.org/doc/numpy/reference/ufuncs.html#ufuncs-kwargs).  *New in version 1.17.0.* |
| **Returns:** | **clipped\_array : *ndarray***  An array with the elements of a, but where values < a\_min are replaced with a\_min, and those > a\_max with a\_max. |

**See also**

**numpy.doc.ufuncs**

Section “Output arguments”

Examples

>>>

**>>>** a = np.arange(10)

**>>>** np.clip(a, 1, 8)

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

**>>>** a

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

**>>>** np.clip(a, 3, 6, out=a)

array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])

**>>>** a = np.arange(10)

**>>>** a

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

**>>>** np.clip(a, [3, 4, 1, 1, 1, 4, 4, 4, 4, 4], 8)

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

# numpy.arccos

**numpy.arccos(x, /, out=None, \*, where=True, casting='same\_kind', order='K', dtype=None, subok=True[, signature, extobj]) = <ufunc 'arccos'>**

Trigonometric inverse cosine, element-wise.

The inverse of [**cos**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.cos.html#numpy.cos) so that, if y = cos(x), then x = arccos(y).

|  |  |
| --- | --- |
| **Parameters:** | **x : *array\_like***  x-coordinate on the unit circle. For real arguments, the domain is [-1, 1].  **out : *ndarray, None, or tuple of ndarray and None, optional***  A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.  **where : *array\_like, optional***  This condition is broadcast over the input. At locations where the condition is True, the out array will be set to the ufunc result. Elsewhere, the out array will retain its original value. Note that if an uninitialized out array is created via the default out=None, locations within it where the condition is False will remain uninitialized.  **\*\*kwargs**  For other keyword-only arguments, see the [ufunc docs](https://docs.scipy.org/doc/numpy/reference/ufuncs.html#ufuncs-kwargs). |
| **Returns:** | **angle : *ndarray***  The angle of the ray intersecting the unit circle at the given x-coordinate in radians [0, pi]. This is a scalar if x is a scalar. |

**See also**

[**cos**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.cos.html#numpy.cos), [**arctan**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.arctan.html#numpy.arctan), [**arcsin**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.arcsin.html#numpy.arcsin), **emath.arccos**

Notes

[**arccos**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.arccos.html#numpy.arccos) is a multivalued function: for each x there are infinitely many numbers z such that cos(z) = x. The convention is to return the angle z whose real part lies in [0, pi].

For real-valued input data types, [**arccos**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.arccos.html#numpy.arccos) always returns real output. For each value that cannot be expressed as a real number or infinity, it yields nan and sets the invalid floating point error flag.

For complex-valued input, [**arccos**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.arccos.html#numpy.arccos) is a complex analytic function that has branch cuts [-inf, -1] and [1, inf] and is continuous from above on the former and from below on the latter.

The inverse [**cos**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.cos.html#numpy.cos) is also known as acos or cos^-1.

References

M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 79. <http://www.math.sfu.ca/~cbm/aands/>

Examples

We expect the arccos of 1 to be 0, and of -1 to be pi:

>>>

**>>>** np.arccos([1, -1])

array([ 0. , 3.14159265])

Plot arccos:

>>>

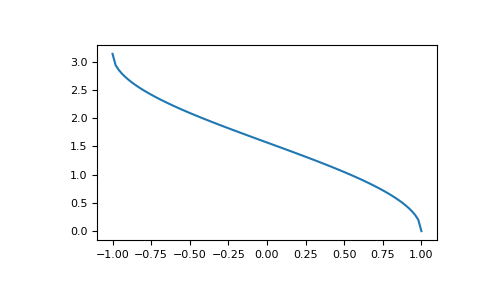
**>>> import** **matplotlib.pyplot** **as** **plt**

**>>>** x = np.linspace(-1, 1, num=100)

**>>>** plt.plot(x, np.arccos(x))

**>>>** plt.axis('tight')

**>>>** plt.show()



# numpy.linalg.norm

**numpy.linalg.norm(**x**,**ord=None**,**axis=None**,**keepdims=False**)**[**[source]**](https://github.com/numpy/numpy/blob/v1.17.0/numpy/linalg/linalg.py#L2325-L2563)

Matrix or vector norm.

This function is able to return one of eight different matrix norms, or one of an infinite number of vector norms (described below), depending on the value of the ord parameter.

|  |  |
| --- | --- |
| **Parameters:** | **x : *array\_like***  Input array. If axis is None, x must be 1-D or 2-D.  **ord : *{non-zero int, inf, -inf, ‘fro’, ‘nuc’}, optional***  Order of the norm (see table under Notes). inf means numpy’s inf object.  **axis : *{int, 2-tuple of ints, None}, optional***  If axis is an integer, it specifies the axis of x along which to compute the vector norms. If axis is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If axis is None then either a vector norm (when x is 1-D) or a matrix norm (when x is 2-D) is returned.  *New in version 1.8.0.*  **keepdims : *bool, optional***  If this is set to True, the axes which are normed over are left in the result as dimensions with size one. With this option the result will broadcast correctly against the original x.  *New in version 1.10.0.* |
| **Returns:** | **n : *float or ndarray***  Norm of the matrix or vector(s). |

Notes

For values of ord <= 0, the result is, strictly speaking, not a mathematical ‘norm’, but it may still be useful for various numerical purposes.

The following norms can be calculated:

| **ord** | **norm for matrices** | **norm for vectors** |
| --- | --- | --- |
| None | Frobenius norm | 2-norm |
| ‘fro’ | Frobenius norm | – |
| ‘nuc’ | nuclear norm | – |
| inf | max(sum(abs(x), axis=1)) | max(abs(x)) |
| -inf | min(sum(abs(x), axis=1)) | min(abs(x)) |
| 0 | – | sum(x != 0) |
| 1 | max(sum(abs(x), axis=0)) | as below |
| -1 | min(sum(abs(x), axis=0)) | as below |
| 2 | 2-norm (largest sing. value) | as below |
| -2 | smallest singular value | as below |
| other | – | sum(abs(x)\*\*ord)\*\*(1./ord) |

The Frobenius norm is given by [[1]](https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.norm.html#rac1c834adb66-1):

The nuclear norm is the sum of the singular values.

References

|  |  |
| --- | --- |
| [**[1]**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.norm.html#id1) | G. H. Golub and C. F. Van Loan, Matrix Computations, Baltimore, MD, Johns Hopkins University Press, 1985, pg. 15 |

Examples

>>>

**>>> from** **numpy** **import** linalg **as** LA

**>>>** a = np.arange(9) - 4

**>>>** a

array([-4, -3, -2, ..., 2, 3, 4])

**>>>** b = a.reshape((3, 3))

**>>>** b

array([[-4, -3, -2],

[-1, 0, 1],

[ 2, 3, 4]])

>>>

**>>>** LA.norm(a)

7.745966692414834

**>>>** LA.norm(b)

7.745966692414834

**>>>** LA.norm(b, 'fro')

7.745966692414834

**>>>** LA.norm(a, np.inf)

4.0

**>>>** LA.norm(b, np.inf)

9.0

**>>>** LA.norm(a, -np.inf)

0.0

**>>>** LA.norm(b, -np.inf)

2.0

>>>

**>>>** LA.norm(a, 1)

20.0

**>>>** LA.norm(b, 1)

7.0

**>>>** LA.norm(a, -1)

-4.6566128774142013e-010

**>>>** LA.norm(b, -1)

6.0

**>>>** LA.norm(a, 2)

7.745966692414834

**>>>** LA.norm(b, 2)

7.3484692283495345

>>>

**>>>** LA.norm(a, -2)

0.0

**>>>** LA.norm(b, -2)

1.8570331885190563e-016 # may vary

**>>>** LA.norm(a, 3)

5.8480354764257312 # may vary

**>>>** LA.norm(a, -3)

0.0

Using the axis argument to compute vector norms:

>>>

**>>>** c = np.array([[ 1, 2, 3],

**...**  [-1, 1, 4]])

**>>>** LA.norm(c, axis=0)

array([ 1.41421356, 2.23606798, 5. ])

**>>>** LA.norm(c, axis=1)

array([ 3.74165739, 4.24264069])

**>>>** LA.norm(c, ord=1, axis=1)

array([ 6., 6.])

Using the axis argument to compute matrix norms:

>>>

**>>>** m = np.arange(8).reshape(2,2,2)

**>>>** LA.norm(m, axis=(1,2))

array([ 3.74165739, 11.22497216])

**>>>** LA.norm(m[0, :, :]), LA.norm(m[1, :, :])

(3.7416573867739413, 11.224972160321824