What is NumPy

NumPy stands for numeric python which is a python package for the computation and processing of the multidimensional and single dimensional array elements.

**Travis Oliphant** created NumPy package in 2005 by injecting the features of the ancestor module Numeric into another module Numarray.

It is an extension module of Python which is mostly written in C. It provides various functions which are capable of performing the numeric computations with a high speed.

NumPy provides various powerful data structures, implementing multi-dimensional arrays and matrices. These data structures are used for the optimal computations regarding arrays and matrices.

The need of NumPy

With the revolution of data science, data analysis libraries like NumPy, SciPy, Pandas, etc. have seen a lot of growth. With a much easier syntax than other programming languages, python is the first choice language for the data scientist.

NumPy provides a convenient and efficient way to handle the vast amount of data. NumPy is also very convenient with Matrix multiplication and data reshaping. NumPy is fast which makes it reasonable to work with a large set of data.

There are the following advantages of using NumPy for data analysis.

1. NumPy performs array-oriented computing.
2. It efficiently implements the multidimensional arrays.
3. It performs scientific computations.
4. It is capable of performing Fourier Transform and reshaping the data stored in multidimensional arrays.
5. NumPy provides the in-built functions for linear algebra and random number generation.

# **NumPy Environment Setup**

NumPy doesn't come bundled with Python. We have to install it using **the python pip** installer. Execute the following command.

1. $ pip install numpy

It is best practice to install NumPy with the full SciPy stack. The binary distribution of the SciPy stack is specific to the operating systems.

## Windows

On the Windows operating system, The SciPy stack is provided by the Anaconda which is a free distribution of the Python SciPy package.

It can be downloaded from the official website: https://www.anaconda.com/. It is also available for Linux and Mac.

The CanoPy also comes with the full SciPy stack which is available as free as well as commercial license. We can download it by visiting the link: https://www.enthought.com/products/canopy/

The Python (x, y) is also available free with the full SciPy distribution. Download it by visiting the link: https://python-xy.github.io/

Linux

In Linux, the different package managers are used to install the SciPy stack. The package managers are specific to the different distributions of Linux. Let's look at each one of them.

Ubuntu

Execute the following command on the terminal.

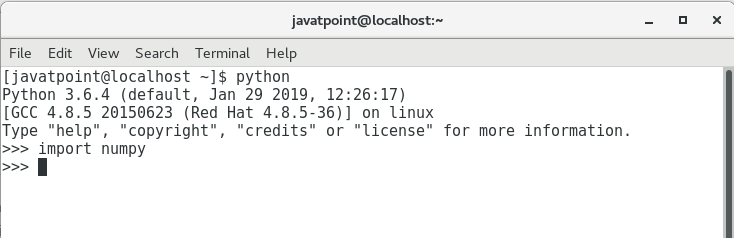
1. $ sudo apt-get install python-numpy
3. $ python-scipy python-matplotlibipythonipythonnotebook python-pandas
5. $ python-sympy python-nose

Redhat

On Redhat, the following commands are executed to install the Python SciPy package stack.

1. $ sudo yum install numpyscipy python-matplotlibipython
3. $ python-pandas sympy python-nose atlas-devel

To verify the installation, open the Python prompt by executing python command on the terminal (cmd in the case of windows) and try to import the module NumPy as shown in the below image. If it doesn't give the error, then it is installed successfully.



# **numpy.array() in Python**

The homogeneous multidimensional array is the main object of **NumPy**. It is basically a table of elements which are all of the same type and indexed by a tuple of positive integers. The dimensions are called axis in NumPy.

The NumPy's array class is known as **ndarray** or **alias array**. The numpy.array is not the same as the standard Python library class **array.array**. The array.array handles only one-dimensional arrays and provides less functionality.

### **Syntax**

1. numpy.array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)

### **Parameters**

There are the following parameters in numpy.array() function.

**1) object: array\_like**

Any object, which exposes an array interface whose \_\_array\_\_ method returns any nested sequence or an array.

**2) dtype : optional data-type**

This parameter is used to define the desired parameter for the array element. If we do not define the data type, then it will determine the type as the minimum type which will require to hold the object in the sequence. This parameter is used only for upcasting the array.

**3) copy: bool(optional)**

If we set copy equals to true, the object is copied else the copy will be made when an object is a nested sequence, or a copy is needed to satisfy any of the other requirements such as dtype, order, etc.

**4) order : {'K', 'A', 'C', 'F'}, optional**

The order parameter specifies the memory layout of the array. When the object is not an array, the newly created array will be in C order (row head or row-major) unless 'F' is specified. When F is specified, it will be in Fortran order (column head or column-major). When the object is an array, it holds the following order.

| **order** | **no copy** | **copy=True** |
| --- | --- | --- |
| 'K' | Unchanged | F and C order preserved. |
| 'A' | Unchanged | When the input is F and not C then F order otherwise C order |
| 'C' | C order | C order |
| 'F' | F order | F order |

When copy=False or the copy is made for the other reason, the result will be the same as copy= True with some exceptions for A. The default order is 'K'.

**5) subok : bool(optional)**

When subok=True, then sub-classes will pass-through; otherwise, the returned array will force to be a base-class array (default).

**6) ndmin : int(optional)**

This parameter specifies the minimum number of dimensions which the resulting array should have. Users can be prepended to the shape as needed to meet this requirement.

### **Returns**

The numpy.array() method returns an ndarray. The ndarray is an array object which satisfies the specified requirements.

## Create a NumPy ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called ndarray.

We can create a NumPy ndarray object by using the array() function.

import numpy as np

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

print(arr)

print(type(arr))

**type():** This built-in Python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy.ndarray type.

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

### **Example**

Use a tuple to create a NumPy array:

import numpy as np  
  
arr = np.array((1, 2, 3, 4, 5))  
  
print(arr)

## Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

**nested array:** are arrays that have arrays as their elements.

## 0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

### **Example**

Create a 0-D array with value 42

import numpy as np  
  
arr = np.array(42)  
  
print(arr)

## 1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

These are the most common and basic arrays.

### **Example**

Create a 1-D array containing the values 1,2,3,4,5:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5])  
  
print(arr)

## 2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.

These are often used to represent matrix or 2nd order teansors.

NumPy has a whole sub module dedicated towards matrix operations called numpy.mat

### **Example**

Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

import numpy as np  
  
arr = np.array([[1, 2, 3], [4, 5, 6]])  
  
print(arr)

## Reshaping arrays

Reshaping means changing the shape of an array.

The shape of an array is the number of elements in each dimension.

By reshaping we can add or remove dimensions or change number of elements in each dimension.

## Reshape From 1-D to 2-D

### **Example**

Convert the following 1-D array with 12 elements into a 2-D array.

The outermost dimension will have 4 arrays, each with 3 elements:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])  
  
newarr = arr.reshape(4, 3)  
  
print(newarr)

## Reshape From 1-D to 3-D

### **Example**

Convert the following 1-D array with 12 elements into a 3-D array.

The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])  
  
newarr = arr.reshape(2, 3, 2)  
  
print(newarr)

### **Example**

Try converting 1D array with 8 elements to a 2D array with 3 elements in each dimension (will raise an error):

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])  
  
newarr = arr.reshape(3, 3)  
  
print(newarr)

## Filtering Arrays

Getting some elements out of an existing array and creating a new array out of them is called *filtering*.

In NumPy, you filter an array using a *boolean index list*.

A *boolean index list* is a list of booleans corresponding to indexes in the array.

If the value at an index is True that element is contained in the filtered array, if the value at that index is False that element is excluded from the filtered array.

### **Example**

Create an array from the elements on index 0 and 2:

import numpy as np  
  
arr = np.array([41, 42, 43, 44])  
  
x = [True, False, True, False]  
  
newarr = arr[x]  
  
print(newarr)

## Creating the Filter Array

In the example above we hard-coded the True and False values, but the common use is to create a filter array based on conditions.

### **Example**

Create a filter array that will return only values higher than 42:

import numpy as np  
  
arr = np.array([41, 42, 43, 44])  
  
# Create an empty list  
filter\_arr = []  
  
# go through each element in arr  
for element in arr:  
  # if the element is higher than 42, set the value to True, otherwise False:  
  if element > 42:  
    filter\_arr.append(True)  
  else:  
    filter\_arr.append(False)  
  
newarr = arr[filter\_arr]  
  
print(filter\_arr)  
print(newarr)

### **Example**

Create a filter array that will return only even elements from the original array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
# Create an empty list  
filter\_arr = []  
  
# go through each element in arr  
for element in arr:  
  # if the element is completely divisble by 2, set the value to True, otherwise False  
  if element % 2 == 0:  
    filter\_arr.append(True)  
  else:  
    filter\_arr.append(False)  
  
newarr = arr[filter\_arr]  
  
print(filter\_arr)  
print(newarr)

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

Indexing:

## Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

### **Example**

Get the first element from the following array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4])  
  
print(arr[0])

### **Example**

Get third and fourth elements from the following array and add them.

import numpy as np  
  
arr = np.array([1, 2, 3, 4])  
  
print(arr[2] + arr[3])

## Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

### **Example**

Access the element on the first row, second column:

import numpy as np  
  
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])  
  
print('2nd element on 1st row: ', arr[0, 1])

## Negative Indexing

Use negative indexing to access an array from the end.

### **Example**

Print the last element from the 2nd dim:

import numpy as np  
  
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])  
  
print('Last element from 2nd dim: ', arr[1, -1])

# **NumPy Array Slicing**

## Slicing arrays

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: [*start*:*end*].

We can also define the step, like this: [*start*:*end*:*step*].

If we don't pass start its considered 0

If we don't pass end its considered length of array in that dimension

If we don't pass step its considered 1

### **Example**

Slice elements from index 1 to index 5 from the following array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[1:5])

Slice elements from index 4 to the end of the array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[4:])

Slice elements from the beginning to index 4 (not included):

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[:4])

import numpy as np

a = np.arange(10)

s = slice(2,7,2)

print a[s]

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.

# slice items starting from index

import numpy as np

a = np.arange(10)

print(a[2:])

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:]

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]]

## Negative Slicing

Use the minus operator to refer to an index from the end:

Slice from the index 3 from the end to index 1 from the end:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[-3:-1])

## STEP

Use the step value to determine the step of the slicing:

### **Example**

Return every other element from index 1 to index 5:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[1:5:2])

### **Example**

Return every other element from the entire array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[::2])

# **NumPy - Broadcasting**

The term **broadcasting** refers to the ability of NumPy to treat arrays of different shapes during arithmetic operations. Arithmetic operations on arrays are usually done on corresponding elements. If two arrays are of exactly the same shape, then these operations are smoothly performed.

import numpy as np

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

b = np.array([10,20,30,40])

c = a \* b

print c

If the dimensions of two arrays are dissimilar, element-to-element operations are not possible. However, operations on arrays of non-similar shapes is still possible in NumPy, because of the broadcasting capability. The smaller array is **broadcast** to the size of the larger array so that they have compatible shapes.

Broadcasting is possible if the following rules are satisfied −

* Array with smaller **ndim** than the other is prepended with '1' in its shape.
* Size in each dimension of the output shape is maximum of the input sizes in that dimension.
* An input can be used in calculation, if its size in a particular dimension matches the output size or its value is exactly 1.
* If an input has a dimension size of 1, the first data entry in that dimension is used for all calculations along that dimension.

A set of arrays is said to be **broadcastable** if the above rules produce a valid result and one of the following is true −

* Arrays have exactly the same shape.
* Arrays have the same number of dimensions and the length of each dimension is either a common length or 1.
* Array having too few dimensions can have its shape prepended with a dimension of length 1, so that the above stated property is true.

The following program shows an example of broadcasting.

## Example 2

import numpy as np

a = np.array([[0.0,0.0,0.0],[10.0,10.0,10.0],[20.0,20.0,20.0],[30.0,30.0,30.0]])

b = np.array([1.0,2.0,3.0])

print ('First array:' )

print(a)

print ('\n)'

print('Second array:' )

print (b)

print( '\n')

print ('First Array + Second Array')

print(a + b)

Statistics is concerned with collecting and then analyzing that data. It includes methods for collecting the samples, describing the data, and then concluding that data. NumPy is the fundamental package for scientific calculations and hence goes hand-in-hand for NumPy statistical Functions.

NumPy contains various statistical functions that are used to perform statistical data analysis. These statistical functions are useful when finding a maximum or minimum of elements. It is also used to find basic statistical concepts like standard deviation, variance, etc.

## NumPy Statistical Functions

NumPy is equipped with the following statistical functions:

**1. np.amin()-** This function determines the minimum value of the element along a specified axis.  
**2. np.amax()-** This function determines the maximum value of the element along a specified axis.  
**3. np.mean()-** It determines the mean value of the data set.  
**4. np.median()-** It determines the median value of the data set.  
**5. np.std()-** It determines the standard deviation

**6. np.average()-** It determines the weighted average

### **Finding maximum and minimum of array in NumPy**

NumPy **np.amin()**and **np.amax()**functions are useful to determine the minimum and maximum value of array elements along a specified axis.

import numpy as np

arr= np.array([[1,23,78],[98,60,75],[79,25,48]])

print(arr)

#Minimum Function

print(np.amin(arr))

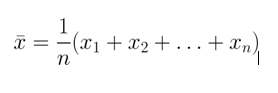
#Maximum Function

print(np.amax(arr))

### **Finding Mean, Median, Standard Deviation and Variance in NumPy**

#### **Mean**

Mean is the sum of the elements divided by its sum and given by the following formula:



It calculates the mean by adding all the items of the arrays and then divides it by the number of elements. We can also mention the axis along which the mean can be calculated.

import numpy as np

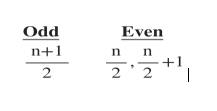
a = np.array([5,6,7])

print(a)

print(np.mean(a))

#### **Median**

Median is the middle element of the array. The formula differs for odd and even sets.



It can calculate the median for both one-dimensional and multi-dimensional arrays. Median separates the higher and lower range of data values.

import numpy as np

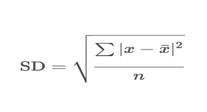
a = np.array([5,6,7])

print(a)

print(np.median(a))

#### **Standard Deviation**

Standard deviation is the square root of the average of square deviations from mean. The formula for standard deviation is:



import numpy as np

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

print(a)

print(np.std(a))

### **NumPy Average Function**

NumPy **np.average()**function determines the weighted average along with the multi-dimensional arrays. The weighted average is calculated by multiplying the component by its weight, the weights are specified separately. If weights are not specified it produces the same output as mean.

import numpy as np

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

print(a)

#without weight same as mean

print(np.average(a))

#with weight gives weighted average

wt = np.array([8,2,3])

print(np.average(a, weights=wt))

# NumPy where() Multiple Conditions

Python NumPy where() is used to get an array with selected elements from the existing array by checking single or multiple conditions. It returns the indices of the array for with each condition being True. Using & Operator, | Operator, NumPy.logical\_and() and, numpy.logical\_or() functions along with where() function based on multiple conditions.

# Create a numpy array

import numpy as np

arr = np.array([5,10,15,20,25])

# Get the indices of array elements

arr1=np.where(arr)

print(arr1)

----------------------------------------------------------------------------

## NumPy where() Multiple Conditions With the & Operator

## To select the NumPy array elements from the existing array-based on multiple conditions using & operator along with where() function. You can specify multiple conditions inside the where() function by enclosing each condition inside a pair of parenthesis and using an & operator.

import numpy as np

arr = np.array([5,10,15,20,25])

# Use numpy.where() multiple conditions with the & operator

arr2 = arr[np.where((arr >5) & (arr <25))]

print(arr2)

----------------------------------------------------------------------------

## where() Multiple Conditions With the | Operator

You can also use the | operator to specify multiple conditions inside the numpy.where() function. You can specify multiple conditions inside this function by enclosing each condition inside a pair of parenthesis and using an | operator.

import numpy as np

arr = np.array([5,10,15,20,25])

# Use numpy.where() multiple conditions with the | operator

arr2 = arr[np.where((arr >5) | (arr % 5 == 0))]

print(arr2)

==============================================

import numpy as np

arr = np.array([5,10,15,20,25])

# Find number of values meet in a condition

arr2 = (arr[np.where((arr >5) | (arr % 5 == 0))]).size

print(arr2)

================================================

## Use where() Multiple Conditions With the logical\_and()

numpy.logical\_and() the function is used to calculate the element-wise truth value of AND gate in Python. Using this function inside the where() function to specify multiple conditions and get the selected elements from an existing array.

import numpy as np

arr = np.array([5,10,15,20,25])

# Use numpy.where() multiple conditions with the numpy.logical\_and()

arr2 = arr[np.where(np.logical\_and(arr >5, arr <25))]

print(arr2)

==============================================

## Use where() Multiple Conditions With the logical\_or()

The numpy.logical\_or() function is used to calculate the element-wise truth value of OR gate in Python. You can use this function inside the where() function to specify multiple conditions.

import numpy as np

arr = np.array([5,10,15,20,25])

# Use numpy.where() multiple conditions with the .logical\_or()

arr2 = arr[np.where(np.logical\_or(arr >5, arr % 5 == 0))]

print(arr2)

============================================

# **NumPy - Arithmetic Operations**

import numpy as np

a = np.arange(9, dtype = np.float\_).reshape(3,3)

print(a)

print('First array:')

print(a)

print('\n')

print('Second array:')

b = np.array([10,10,10])

print(b)

print('\n')

print('Add the two arrays:')

print(np.add(a,b))

print('\n')

print('Subtract the two arrays:')

print(np.subtract(a,b))

print('\n')

print('Multiply the two arrays:')

print(np.multiply(a,b))

print('\n')

print('Divide the two arrays:')

print(np.divide(a,b))

==========================================

# Numpy numpy.transpose()

With the help of **Numpy numpy.transpose()**, We can perform the simple function of transpose within one line by using **numpy.transpose()** method of Numpy. It can transpose the 2-D arrays on the other hand it has no effect on 1-D arrays. This method transpose the 2-D numpy array.

***Parameters:******axes :****[None, tuple of ints, or n ints] If anyone wants to pass the parameter then you can but it’s not all required. But if you want than remember only pass****(0, 1)****or****(1, 0)****. Like we have array of shape (2, 3) to change it (3, 2) you should pass (1, 0) where 1 as 3 and 0 as 2.****Returns:****ndarray*

**Example #1 :**   
In this example we can see that it’s really easy to transpose an array with just one line.

# importing python module named numpy

**import** numpy as np

# making a 3x3 array

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

                [4, 5, 6],

                [7, 8, 9]])

# before transpose

**print**(ans, end **=**'\n\n')

# after transpose

print(ans.transpose())

=================================================

**Example #2 :**   
In this example we demonstrate the use of tuples in numpy.transpose().

# importing python module named numpy

import numpy as np

# making a 3x3 array

gfg = np.array([[1, 2],

[4, 5],

[7, 8]])

# before transpose

print(gfg, end ='\n\n')

# after transpose

print(gfg.transpose(1,0))

===============================================

**Method 2:**Using Numpy ndarray.T object.

| # importing python module named numpy  **import** numpy as np    # making a 3x3 array  gfg **=** np.array([[1, 2, 3],                  [4, 5, 6],                  [7, 8, 9]])    # before transpose  **print**(gfg, end **=**'\n\n')    # after transpose  print(gfg.T) |
| --- |

========================================================

# importing python module named numpy

import numpy as np

# making a 3x3 array

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

print(a)

print(np.transpose(a))

print("==========================")

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

print(a)

print(np.transpose(a))

print("======================")

a = np.ones((100, 12, 30))

print(np.transpose(a, (1, 0, 2)).shape)

print("===========================")

a = np.ones((2, 3, 4, 5))

print(np.transpose(a).shape)

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

# **numpy.linalg.inv:Inverse**

**linalg.inv(*a)***

Compute the (multiplicative) inverse of a matrix.

Given a square matrix *a*, return the matrix *ainv* satisfying dot(a, ainv) = dot(ainv, a) = eye(a.shape[0]).

**Parameters:**

**a*(…, M, M) array\_like***

Matrix to be inverted.

**Returns:**

**ainv*(…, M, M) ndarray or matrix***

(Multiplicative) inverse of the matrix *a*.

**Raises:**

**LinAlgError**

If *a* is not square or inversion fails.

import numpy as np

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

print(a)

b = np.linalg.inv(a)

print(b)

print("==============================")

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

[[3, 7], [4, 1]]])

b = np.linalg.inv(a)

print(b)

========================================

import numpy as np

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

print(a)

b = np.linalg.inv(a)

print(b)

print(np.allclose(np.dot(a,b), np.eye(2)))