

Overview:

Numpy is a multidimensional array library.

Lists are very slow.

**Why is NumPy Faster?**

NumPy is very fast. Because NumPy uses fixed type i.e it stores as Int32.

Lists are slow because they keep references. And an integer in list takes a hell lot of memory.

NumPy has no type checking.

NumPy has contiguous memory.

We can multiply arrays in NumPy. a\*b

**Applications of NumPy:**

Replacement of MATLAB for Mathematics.

Backend of Pandas.

Photography.

Machine Learning e.g. Tensors.

# NumPy import:

| import numpy as np |
| --- |

Basics

# Basic array:

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

[1 2 3]

# 2-D array:

| b = np.array([[1,2,3],[4,5,6]])  #Should have equal elements in all rows |
| --- |

[[1 2 3]

[4 5 6]]

# Get dimension of NumPy array:

| b.ndim |
| --- |

2

# Get shape:

| b.shape |
| --- |

(2, 3)

# Get Type:

| a.dtype |
| --- |

dtype('int32')

# Specifying type of array while initialization:

| a = np.array([[1,2,3],[4,5,6]],dtype='int16') a.dtype |
| --- |

dtype('int16')

# Size of a single item for int16 array:

| a.itemsize |
| --- |

2

# Total number of elements in array:

| a.size |
| --- |

6

# Total number of bytes in array:

| a.nbytes |
| --- |

12

Accessing/Changing specific elements, rows, columns,etc

# To get a specific element (row, column):

| a = np.array([[1,2,3,4,5,6,7],[8,9,10,11,12,13,14]]) a[0,0] |
| --- |

1

# Other stuff::

| a[0,-1] |
| --- |

7

| a[0,:] |
| --- |

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

| a[:,2] |
| --- |

array([ 3, 10])

| a[:,2:4] |
| --- |

array([[ 3, 4],

[10, 11]])

General Syntax - a[start\_index:end\_index:step\_size**,** start\_index:end\_index:step\_size**,** …..n dimensions]

# Accessing multiple indices (Indices (1,1), (1,2) and (1,3)):

| file\_data[[1,1,1],[1,2,3]] |
| --- |

# Setting values:

| a[0,3]=10 |
| --- |

# Setting values for entire row:

| a = np.array([[1,2,3,4,5,6,7],[8,9,10,11,12,13,14]]) a[1,:]=99 print(a) |
| --- |

[[ 1 2 3 4 5 6 7]

[99 99 99 99 99 99 99]]

# 3-D array:

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

[[[1 2]

[3 4]]

[[5 6]

[7 8]]]

# Accessing a 3-D element:

| b[1,1,0] |
| --- |

7

# Using a range function:

| np.arange(start = 0, stop = 5, step = 1) |
| --- |

array([0, 1, 2, 3, 4])

# linspace (which we give start and end point and intervals in between):

| np.linspace(start = 0, stop = 100, num = 5) |
| --- |

array([ 0., 25., 50., 75., 100.])

# To ravel an array:

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

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

Initializing different types of arrays

# To create an array of zeros using dimensions:

| b = np.zeros((2,3)) print(b) |
| --- |

[[0. 0. 0.]

[0. 0. 0.]]

# To create an array of ones using dimensions:

| b = np.ones((2,3), dtype='int32') print(b) |
| --- |

[[1 1 1]

[1 1 1]]

# Any other number (we use full):

| b = np.full((2,2), 99) |
| --- |

[[99 99]

[99 99]]

# If we want it to have the same dimensions as another array:

| b = np.full\_like(a, 4) or b = np.full(a.shape(), 4) |
| --- |

# Generating a random number array:

| b = np.random.rand(4,2) |
| --- |

[[0.85754961 0.91127152]

[0.45251941 0.12162455]

[0.50694868 0.05315512]

[0.70002409 0.30481215]]

# Generating a random number array using shape of other array:

| b = np.random.random\_sample(a.shape) |
| --- |

# Generating a random number array using randn(includes negative numbers):

| array = np.random.randn(4) |
| --- |

[ 0.50302852 -1.24692807 1.04556022 -0.05267173]

# Array with random integers:

| b = np.random.randint(4,7, size=(3,3))  #Array with random numbers from 4 to 7 where 7 is exclusive |
| --- |

[[5 4 5]

[6 4 5]

[5 4 4]]

# Identity matrix:

| b = np.identity(3) |
| --- |

[[1. 0. 0.]

[0. 1. 0.]

[0. 0. 1.]]

# Repeating an array:

| arr = np.array([[1,2,3]]) b = np.repeat(arr, 3, axis = 0) # Where 3 is no. of times to repeat print(b) |
| --- |

[[1 2 3]

[1 2 3]

[1 2 3]]

# Repeating an array horizontally:

| arr = np.array([[1,2,3]]) b = np.repeat(arr, 3, axis = 1) # Where 3 is no. of times to repeat print(b) |
| --- |

[[1 1 1 2 2 2 3 3 3]]

# To print a design:

| import numpy as np  n = int(input()) arr = np.full((n,n),99) for i in range(0,n):  arr[i:-i,i:-i] = i print(arr) |
| --- |

[[99 99 99 99 99 99 99 99 99 99]

[99 1 1 1 1 1 1 1 1 99]

[99 1 2 2 2 2 2 2 1 99]

[99 1 2 3 3 3 3 2 1 99]

[99 1 2 3 4 4 3 2 1 99]

[99 1 2 3 4 4 3 2 1 99]

[99 1 2 3 3 3 3 2 1 99]

[99 1 2 2 2 2 2 2 1 99]

[99 1 1 1 1 1 1 1 1 99]

[99 99 99 99 99 99 99 99 99 99]]

# Beware of copying an array. It usually copies the link rather than values:

| a = np.array([1,2,3]) b = a b[0]=100 print(a) |
| --- |

[100 2 3]

# To prevent linking of arrays, we use copy():

| a = np.array([1,2,3]) b = a.copy() b[0]=100 print(a) |
| --- |

[1 2 3]

Mathematics

# Basic math on arrays:

| a = np.array([1,2,3])  print(a + 2)  print(a - 2)  print(a \* 2)  print(a / 2)  print(a \*\* 2) |
| --- |

a + 2

[3 4 5]

a - 2

[-1 0 1]

a \* 2

[2 4 6]

a / 2

[0.5 1. 1.5]

a \*\* 2

[1 4 9]

# Sin on the array:

| a = np.array([1,2,3]) np.sin(a) |
| --- |

array([0.84147098, 0.90929743, 0.14112001])

# Square root on the array:

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

array([[1. , 1.41421356, 1.73205081],

[2. , 2.23606798, 2.44948974]])

# Standard deviation on the array:

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

1.707825127659933

# Exponent ex:

| np.exp(a) |
| --- |

# Logarithm log x:

| np.log(a) |
| --- |

# Logarithm log10 x:

| np.log10(a) |
| --- |

Linear Algebra

# Matrix multiplication (It should have compatible dimensions):

| a = np.ones((2,3)) b = np.full((3,2),2) np.matmul(a,b) |
| --- |

array([[6., 6.],

[6., 6.]])

# Finding the determinant of a matrix:

| a = np.identity(3) np.linalg.det(a) |
| --- |

1.0

For more functions on linear algebra.

<https://docs.scipy.org/doc/numpy/reference/routines.linalg.html>

Statistics

# Finding minimum in data:

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

1

# Finding maximum in data:

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

6

# Using axis for min, max:

| np.min(a, axis=0) #Finds minimum in each column |
| --- |

array([1, 2, 3])

| np.min(a, axis=1) #Finds minimum in each row |
| --- |

array([1, 4])

Reorganizing Arrays

# Changing dimensions (we can change to any compatible dimension):

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

array([[1, 2],

[3, 4],

[5, 6]])

If we use -1 for dimensions in reshape, e.g. (2,-1), (-1,4), except (-1,-1) python will automatically find the right dimension number to replace the -1’s place for that dataset.

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

array([[1],

[2],

[3],

[4],

[5],

[6]])

# Vertical stacking:

| v1 = np.array([1,2,3,4,5]) v2 = np.array([6,7,8,9,10])  np.vstack([v1,v2]) |
| --- |

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

[ 6, 7, 8, 9, 10]])

# Horizontal stacking:

| h1 = np.ones((2,4),dtype='int32') h2 = np.zeros((2,2),dtype='int32')  np.hstack([h1,h2]) |
| --- |

array([[1, 1, 1, 1, 0, 0],

[1, 1, 1, 1, 0, 0]])

Miscellaneous

GitHub link for data:

<https://github.com/KeithGalli/NumPy/blob/master/data.txt>

# Load data from a File:

| file\_data = np.genfromtxt('data.txt',delimiter=',') |
| --- |

array([[ 1., 13., 21., 11., 196., 75., 4., 3., 34., 6., 7.,

8., 0., 1., 2., 3., 4., 5.],

[ 3., 42., 12., 33., 766., 75., 4., 55., 6., 4., 3.,

4., 5., 6., 7., 0., 11., 12.],

[ 1., 22., 33., 11., 999., 11., 2., 1., 78., 0., 1.,

2., 9., 8., 7., 1., 76., 88.]])

# To change array data type:

| file\_data = file\_data.astype('int32') |
| --- |

Boolean Masking and Advanced Indexing

File data:

array([[ 1, 13, 21, 11, 196, 75, 4, 3, 34, 6, 7, 8, 0,

1, 2, 3, 4, 5],

[ 3, 42, 12, 33, 766, 75, 4, 55, 6, 4, 3, 4, 5,

6, 7, 0, 11, 12],

[ 1, 22, 33, 11, 999, 11, 2, 1, 78, 0, 1, 2, 9,

8, 7, 1, 76, 88]])

# Searching for data which is greater than 50 in the file\_data:

| file\_data > 50 |
| --- |

array([[False, False, False, False, True, True, False, False, False,

False, False, False, False, False, False, False, False, False],

[False, False, False, False, True, True, False, True, False,

False, False, False, False, False, False, False, False, False],

[False, False, False, False, True, False, False, False, True,

False, False, False, False, False, False, False, True, True]])

# Indexing the condition:

| file\_data[file\_data > 50] |
| --- |

array([196, 75, 766, 75, 55, 999, 78, 76, 88])

# We can specify elements of array using list of index:

| a = np.array([0,10,20,30,40,50,60,70,80]) a[[0,5,8]] # We can specify using index list |
| --- |

array([ 0, 50, 80])

# Find if **any** of thevalues in **ROW** or **COLUMN** has a condition:

| np.any(file\_data > 50, axis=0) |
| --- |

array([False, False, False, False, True, True, False, True, True,

False, False, False, False, False, False, False, True, True])

# Find if **all** values in **ROW** or **COLUMN** has a condition:

| np.all(file\_data > 50, axis=0) |
| --- |

array([False, False, False, False, True, False, False, False, False,

False, False, False, False, False, False, False, False, False])

# Combining conditions:

| ((file\_data > 50) & (file\_data <100)) |
| --- |

array([[False, False, False, False, False, True, False, False, False,

False, False, False, False, False, False, False, False, False],

[False, False, False, False, False, True, False, True, False,

False, False, False, False, False, False, False, False, False],

[False, False, False, False, False, False, False, False, True,

False, False, False, False, False, False, False, True, True]])

# Not condition:

| (~(file\_data > 50) & (file\_data <100)) |
| --- |

array([[ True, True, True, True, False, False, True, True, True,

True, True, True, True, True, True, True, True, True],

[ True, True, True, True, False, False, True, False, True,

True, True, True, True, True, True, True, True, True],

[ True, True, True, True, False, True, True, True, False,

True, True, True, True, True, True, True, False, False]])

​

# Mesh in numpy:

| import numpy as np  # from matplotlib import pyplot as plt  # pyplot imported for plotting graphs    x = np.linspace(-4, 4, 9)    # numpy.linspace creates an array of  # 9 linearly placed elements between  # -4 and 4, both inclusive  y = np.linspace(-5, 5, 11)    # The meshgrid function returns  # two 2-dimensional arrays  x\_1, y\_1 = np.meshgrid(x, y)    print("x\_1 = ")  print(x\_1)  print("y\_1 = ")  print(y\_1) |
| --- |

x\_1 =

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

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

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

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

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

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

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

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

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

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

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

y\_1 =

[[-5. -5. -5. -5. -5. -5. -5. -5. -5.]

[-4. -4. -4. -4. -4. -4. -4. -4. -4.]

[-3. -3. -3. -3. -3. -3. -3. -3. -3.]

[-2. -2. -2. -2. -2. -2. -2. -2. -2.]

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

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

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

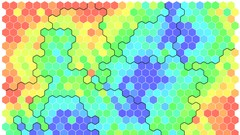
[ 2. 2. 2. 2. 2. 2. 2. 2. 2.]

[ 3. 3. 3. 3. 3. 3. 3. 3. 3.]

[ 4. 4. 4. 4. 4. 4. 4. 4. 4.]

[ 5. 5. 5. 5. 5. 5. 5. 5. 5.]]

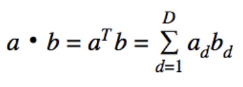
## [www.udemy.com/course/deep-learning-prerequisites-the-numpy-stack-in-python](http://www.udemy.com/course/deep-learning-prerequisites-the-numpy-stack-in-python)



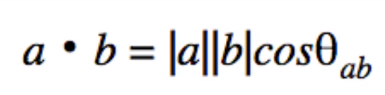
Deep Learning Prerequisites: The Numpy Stack in Python by Lazy Programmer Inc.

## Dot Product

### Method 1



### Method 2



|a| is the magnitude. Meaning it is the square root of the sum of squares of its individual components.

E.g. vector a = 5i + 8j -3k

|a| = sqrt(5\*5 + 8\*8 + (-3)\*(-3))

### Dot product with Numpy

| a = np.array([1,2]) b = np.array([2,1])  np.dot(a,b) |
| --- |

4

## Calculating the magnitude of a vector in Numpy

| amag = np.linalg.norm(a) |
| --- |

### Finding the angle between two vectors

| cosangle = a.dot(b) / (np.linalg.norm(a) \* np.linalg.norm(b)) angle = np.arccos(cosangle) |
| --- |

## Inverse cosine function on a cosine value

| arccos(angle) |
| --- |

## Matrix Data Type

| M = np.matrix([[1,2],[3,4]]) |
| --- |

But don’t use it since official documentation is against using it.

## np.random.randn()

Return a sample (or samples) from the “standard normal” distribution.

| G = np.random.randn(10,10) |
| --- |

## Calculating Mean of an array

| G.mean() |
| --- |

## Calculating variance of an array

| G.var() |
| --- |

## Calculating inverse of a matrix

| A = np.array([[1,2],[3,4]]) Ainv = np.linalg.inv(A) Ainv |
| --- |

## Calculating determinant of a matrix

| np.linalg.det(A) |
| --- |

## Getting the diagonal elements of a matrix

| np.diag(A) |
| --- |

## Creating a matrix with a diagonal by a given array of elements

It creates a matrix of zeros but with diagonal elements replaced by the array passed in the arguments.

If we pass a 1-D array, we get a matrix like this.

If we pass a 2-D array, we get its diagonal elements like the previous example.

| np.diag([1,2]) |
| --- |

array([[1, 0],

[0, 2]])

## To do a matrix multiplication

We do a dot product instead of a \*.

| np.array([[1,2],[3,4]]).dot(np.array([[1,2],[3,4]])) |
| --- |

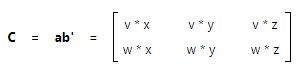
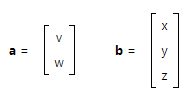
array([[ 7, 10],

[15, 22]])

## Inner product and outer product

The inner product is the same as the dot product.

The outer product of two vectors produces a rectangular matrix, not a scalar. This is illustrated below.



Notice that the elements of Matrix C consist of the product of elements from Vector A crossed with elements from Vector B. Thus, Matrix C winds up being a matrix of cross products from the two vectors.

### Outer product and inner product in Numpy

| A = np.array([[1],[2]]) B = np.array([[3],[4]]) np.outer(A,B) |
| --- |

array([[3, 4],

[6, 8]])

| A = np.array([1,2]) B = np.array([3,4]) np.inner(A,B) |
| --- |

11

## Matrix trace (Sum of diagonals of the matrix)

| np.diag(A).sum() |
| --- |

Or

| np.trace() |
| --- |

## Calculating Covariance

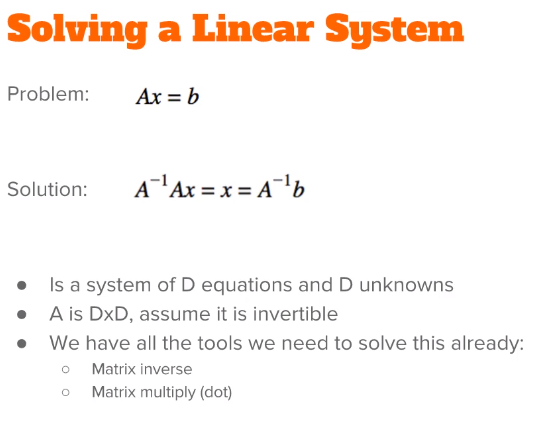
| X = np.random.randn(100,3) cov = np.cov(X) |
| --- |

## Calculating Eigenvalues and Eigenvectors

| eigenvalues, eigenvectors = np.eig(C)  eigenvalues, eigenvectors = np.eigh(C) |
| --- |

eigh is for symmetric and Hermitian matrices

## Solving a Linear System



### Conventional way to solve linear system

| A = np.array([[1,2],[3,4]]) b = np.array([1,2]) x = np.linalg.inv(A).dot(b) x |
| --- |

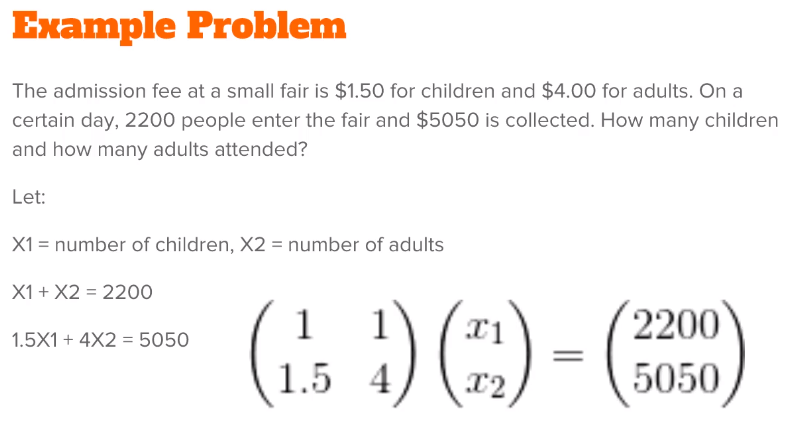
array([0. , 0.5])

Better way in Numpy

| x = np.linalg.solve(A,b) |
| --- |

array([0. , 0.5])

## Solving an actual word problem



### Program to solve it

| A = np.array([[1,1],[1.5,4]]) b = np.array([2200,5050])  np.linalg.solve(A,b) |
| --- |

array([1500., 700.])

Extras

# flatten()

It converts it to a 1D array.

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

[1,2,3,4]

Flatten can take 2 parameters i.e ‘C’(row-major), ‘F’(column-major).

# np.mean()

Whenever we compare two Numpy arrays, we get a resultant array of 0’s(true) and 1’s(false).

np.mean(A == B) will give the mean of the array of 0’s and 1’s.

# np.allclose

Returns True if two arrays are element-wise equal within a tolerance.

| np.allclose([1e10,1e-7], [1.00001e10,1e-8]) |
| --- |

False

# np.where

Returns the index values where the index is true.

| a = np.where(p\_test != y\_test) |
| --- |

# numpy.expand\_dims

Expand the shape of an array.

Insert a new axis that will appear at the axis position in the expanded array shape.

| numpy.expand\_dims(a, axis) |
| --- |

| a = np.expand\_dims(X,-1) |
| --- |

Here, we are adding an extra dimension.

# np.asarray

Converts PIL image to numpy array.

| from PIL import Image  pic = Image.open('C:/Users/tejas/Desktop/car.jpg') pic\_arr = np.asarray(pic) type(pic\_arr) |
| --- |

numpy.ndarray

## numpy.power

Increases the power of all values by the specified value.

| x1 = range(6) |
| --- |

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

| np.power(x1, 3) |
| --- |

array([ 0, 1, 8, 27, 64, 125])

## numpy.append

It appends to the end of the array.

| np.append(arr = X, values = np.ones((50,1)).astype(int), axis = 1) |
| --- |

Here we are appending ones to the end.

# Changing to any data type

## numpy.float32()

We pass in a numpy array, it will just convert it to that data type and return.

| a = np.float32(b) |
| --- |

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

| np.empty([2, 2]) |
| --- |

array([[ -9.74499359e+001, 6.69583040e-309],

[ 2.13182611e-314, 3.06959433e-309]])

# numpy.fromstring(string, dtype=float, count=-1, sep='')

A new 1-D array initialized from text data in a string.

| np.fromstring('1 2', dtype=int, sep=' ') |
| --- |

array([1, 2])

| np.fromstring('1, 2', dtype=int, sep=',') |
| --- |

array([1, 2])

# np.concatenate

| import numpy as np a = np.array([1, 2, 3]) b = np.array([5, 6]) c=np.concatenate([a,b,a]) print(c) |
| --- |

[1 2 3 5 6 1 2 3]