AI@Amrita- Session I Introduction to Numpy

Numpy Tutorial

In [3]:
```
import numpy as np
```

Simplest way to create an array in Numpy is to use Python List

In [1]:
```
myPythonList = [1,9,8,3]
print(myPythonList)
```

```
[1, 9, 8, 3]
```

To convert python list to a numpy array by using the object np.array.

In [4]:
```
numpy_array_from_list = np.array(myPythonList)
numpy_array_from_list
```

Out[4]:
```
array([1, 9, 8, 3])
```

In [7]:
```
a  = np.array([1,9,8,3])
print(a)
```

```
[1 9 8 3]
```

In [8]:
```
x=numpy_array_from_list + 10
print(x)
```

```
[11 19 18 13]
```

Shape of Array You can check the shape of the array with the object shape preceded by the name of the array. In the same way, you can check the type with dtypes.

In [9]:
```
a  = np.array([1,2,3])
print(a.shape)
print(a.dtype)
```

```
(3,)
int64
```

In [10]:
```
#### Different type
b  = np.array([1.1,2.0,3.2])
print(b.dtype)
```

```
float64
```

In [12]:
```
### 2 dimension
c = np.array([(1,2,3),
              (4,5,6)])
print(c)
```

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

In [13]:
```
### 3 dimension
d = np.array([
    [[1, 2,3],
        [4, 5, 6]],
    [[7, 8,9],
        [10, 11, 12]]
])
print(d.shape)
(2, 2, 3)
```

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

Out[13]:
```
(2, 2, 3)
```

What is numpy.zeros()? numpy.zeros() or np.zeros Python function is used to create a matrix full of zeroes. numpy.zeros() in Python can be used when you initialize the weights during the first iteration in TensorFlow and other statistic tasks.

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

Shape: is the shape of the numpy zero array Dtype: is the datatype in numpy zeros. It is optional. The default value is float64 Order: Default is C which is an essential row style for numpy.zeros() in Python.

In [14]:
```
import numpy as np
x=np.zeros((2,2), dtype=np.int16)
print(x)
```

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

In [15]:
```
import numpy as np
np.ones((1,2,3), dtype=np.int16)
```

Out[15]:
```
array([[[1, 1, 1],
        [1, 1, 1]]], dtype=int16)
```

Reshape Data In some occasions, you need to reshape the data from wide to long. You can use the reshape function for this. The syntax is

numpy.reshape(a, newShape, order='C') Here,

a: Array that you want to reshape

newShape: The new desires shape

Order: Default is C which is an essential row style.

In [16]:
```
import numpy as np
e  = np.array([(1,2,3), (4,5,6)])
print(e)
e.reshape(3,2)
```

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

Out[16]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Flatten Data When you deal with some neural network like convnet, you need to flatten the array. You can use flatten(). The syntax is

numpy.flatten(order='C') Here,

Order: Default is C which is an essential row style.

In [17]:
```
e.flatten()
```

Out[17]:
```
array([1, 2, 3, 4, 5, 6])
```

What is hstack? With hstack you can appened data horizontally. This is a very convenient function in Numpy

In [18]:
```
## Horitzontal Stack
import numpy as np
f = np.array([1,2,3])
g = np.array([4,5,6])

print('Horizontal Append:', np.hstack((f, g)))
```

```
Horizontal Append: [1 2 3 4 5 6]
```

In [ ]:
```

```

What is vstack? With vstack you can appened data vertically. Lets study it with an example:

In [19]:
```
## Vertical Stack
import numpy as np
f = np.array([1,2,3])
g = np.array([4,5,6])

print('Vertical Append:', np.vstack((f, g)))
```

```
Vertical Append: [[1 2 3]
 [4 5 6]]
```

Generate Random Numbers To generate random numbers for Gaussian distribution use

numpy.random.normal(loc, scale, size)

Here

Loc: the mean. The center of distribution scale: standard deviation. Size: number of returns

In [21]:
```
## Generate random nmber from normal distribution
normal_array = np.random.normal(5, 0.5, 10)
print(normal_array)
```

```
[5.78943125 4.7591597  5.99314678 6.11610238 5.95480875 4.58438977
 5.57644342 5.14760086 4.32919408 5.81119806]
```

Asarray The asarray()function is used when you want to convert an input to an array. The input could be a lists, tuple, ndarray, etc.

Syntax:

numpy.asarray(data, dtype=None, order=None)[source] Here,

data: Data that you want to convert to an array

dtype: This is an optional argument. If not specified, the data type is inferred from the input data

Order: Default is C which is an essential row style. Other option is F (Fortan-style)

In [ ]:
```

```

In [23]:
```
A = np.matrix(np.ones((4,4)))
np.array(A)[2]=2  #immutable
print(A)
```

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

In [24]:
```
np.asarray(A)[2]=2
print(A)
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [2. 2. 2. 2.]
 [1. 1. 1. 1.]]
```

What is numpy.arange()? numpy.arange() is an inbuilt numpy function that returns an ndarray object containing evenly spaced values within a defined interval. For instance, you want to create values from 1 to 10; you can use numpy.arange() function.

Syntax:

numpy.arange(start, stop,step) Start: Start of interval Stop: End of interval Step: Spacing between values. Default step is 1

In [26]:
```
np.arange(1, 11)
```

Out[26]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [28]:
```
np.arange(1, 14, 4)
```

Out[28]:
```
array([ 1,  5,  9, 13])
```

Linspace Linspace gives evenly spaced samples.

Syntax:

numpy.linspace(start, stop, num, endpoint) Here,

Start: Starting value of the sequence Stop: End value of the sequence Num: Number of samples to generate. Default is 50 Endpoint: If True (default), stop is the last value. If False, stop value is not included.

In [33]:
```
import numpy as np
np.linspace(1.0, 10.0, num=20)
```

Out[33]:
```
array([ 1.        ,  1.47368421,  1.94736842,  2.42105263,  2.89473684,
        3.36842105,  3.84210526,  4.31578947,  4.78947368,  5.26315789,
        5.73684211,  6.21052632,  6.68421053,  7.15789474,  7.63157895,
        8.10526316,  8.57894737,  9.05263158,  9.52631579, 10.        ])
```

In [34]:
```
np.linspace(1.0, 5.0, num=5, endpoint=False)
```

Out[34]:
```
array([1. , 1.8, 2.6, 3.4, 4.2])
```

LogSpace LogSpace returns even spaced numbers on a log scale. Logspace has the same parameters as np.linspace.

Syntax:

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

In [35]:
```
np.logspace(3.0, 4.0, num=4)
```

Out[35]:
```
array([ 1000.        ,  2154.43469003,  4641.58883361, 10000.        ])
```

In [36]:
```
x = np.array([1,2,3], dtype=np.complex128)
x.itemsize
```

Out[36]:
```
16
```

Indexing and slicing Slicing data is trivial with numpy. We will slice the matrice "e". Note that, in Python, you need to use the brackets to return the rows or columns

In [38]:
```
## Slice
import numpy as np
e  = np.array([(1,2,3), (4,5,6)])
print(e)
```

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

In [39]:
```
## First column
print('First row:', e[0])

## Second col
print('Second row:', e[1])
```

```
First row: [1 2 3]
Second row: [4 5 6]
```

In [40]:
```
print('Second column:', e[:,1])
```

```
Second column: [2 5]
```

In [42]:
```
## Second Row, two values
print(e[1, :2])
```

```
[4 5]
```

NumPy has quite a few useful statistical functions for finding minimum, maximum, percentile standard deviation and variance, etc from the given elements in the array. The functions are explained as follows −

Statistical function Numpy is equipped with the robust statistical function as listed below

Function Numpy Min np.min() Max np.max() Mean np.mean() Median np.median() Standard deviation np.std()

In [43]:
```
import numpy as np
normal_array = np.random.normal(5, 0.5, 10)
print(normal_array)
```

```
[5.38157118 5.37186171 5.05123951 4.32336104 4.66653566 5.81183913
 4.63195366 5.2945789  5.55464569 5.5953062 ]
```

In [44]:
```
### Min
print(np.min(normal_array))

### Max
print(np.max(normal_array))

### Mean
print(np.mean(normal_array))

### Median
print(np.median(normal_array))

### Sd
print(np.std(normal_array))
```

```
4.323361037578353
5.811839127089547
5.168289266004939
5.333220303215558
0.4599639204216421
```

What is numpy dot product? Numpy.dot product is a powerful library for matrix computation. For instance, you can compute the dot product with np.dot. Numpy.dot product is the dot product of a and b. numpy.dot() in Python handles the 2D arrays and perform matrix multiplications.

Syntax

numpy.dot(x, y, out=None) Parameters

Here,

x,y: Input arrays. x and y both should be 1-D or 2-D for the np.dot() function to work

out: This is the output argument for 1-D array scalar to be returned. Otherwise ndarray should be returned.

Returns

The function numpy.dot() in Python returns a Dot product of two arrays x and y. The dot() function returns a scalar if both x and y are 1-D; otherwise, it returns an array. If 'out' is given then it is returned.

Raises

Dot product in Python raises a ValueError exception if the last dimension of x does not have the same size as the second last dimension of y.

In [45]:
```
## Linear algebra
### Dot product: product of two arrays
f = np.array([1,2])
g = np.array([4,5])
### 1*4+2*5
np.dot(f, g)
```

Out[45]:
```
14
```

Matrix Multiplication The Numpu matmul() function is used to return the matrix product of 2 arrays. Here is how it works

1) 2-D arrays, it returns normal product

2) Dimensions > 2, the product is treated as a stack of matrix

3) 1-D array is first promoted to a matrix, and then the product is calculated

numpy.matmul(x, y, out=None) Here,

x,y: Input arrays. scalars not allowed

In [46]:
```
### Matmul: matruc product of two arrays
h = [[1,2],[3,4]]
i = [[5,6],[7,8]]
### 1*5+2*7 = 19
np.matmul(h, i)
```

Out[46]:
```
array([[19, 22],
       [43, 50]])
```

How to inverse a matrix using NumPy

In [49]:
```
# Taking a 3 * 3 matrix
A = np.array([[6, 1, 1],
              [4, -2, 5],
              [2, 8, 7]])

# Calculating the inverse of the matrix
print(np.linalg.inv(A))
```

```
[[ 0.17647059 -0.00326797 -0.02287582]
 [ 0.05882353 -0.13071895  0.08496732]
 [-0.11764706  0.1503268   0.05228758]]
```

How to Calculate the determinant of a matrix using NumPy?

In [50]:
```
# creating a 2X2 Numpy matrix
n_array = np.array([[50, 29], [30, 44]])

# Displaying the Matrix
print("Numpy Matrix is:")
print(n_array)

# calculating the determinant of matrix
det = np.linalg.det(n_array)

print("\nDeterminant of given 2X2 matrix:")
print(int(det))
```

```
Numpy Matrix is:
[[50 29]
 [30 44]]

Determinant of given 2X2 matrix:
1330
```

Rank of a Matrix

In [ ]:

In [51]:
```
# Let's create a square matrix (NxN matrix)
mx = np.array([[1,1,1],[0,1,2],[1,5,3]])

# Let's get rank of matrix
np.linalg.matrix_rank(mx)
```

Out[51]:
```
3
```

How To Calculate Eigenvectors And Eigenvalues With Numpy

In [54]:
```
example_matrix = np.array([
  [1,2,3,4],
  [5,6,7,8],
  [9,10,11,12],
  [13,14,15,16]
])
eigenvalues, eigenvectors = np.linalg.eig(example_matrix)
print(eigenvalues, eigenvectors)
```

```
[ 3.62093727e+01 -2.20937271e+00 -5.14138333e-16 -2.08943529e-15] [[-0.15115432
-0.72704996  0.01790738 -0.26832672]
 [-0.34923733 -0.28320876  0.38415354  0.71367217]
 [-0.54732033  0.16063243 -0.82202921 -0.62236419]
 [-0.74540333  0.60447363  0.41996829  0.17701873]]
```

Get diagonal of a matrix

In [56]:
```
# make matrix with numpy
gfg = np.matrix('[6, 2; 3, 4]')

# applying matrix.diagonal() method
diag = gfg.diagonal()

print(diag)
```

```
[[6 4]]
```

Matrix transpose

In [57]:
```
# make matrix with numpy
gfg = np.matrix('[4, 1; 12, 3]')

# applying matrix.transpose() method
trans = gfg.transpose()

print(trans)
```

```
[[ 4 12]
 [ 1  3]]
```

Trace of a matrix

In [58]:
```
# make matrix with numpy
gfg = np.matrix('[4, 1; 12, 3]')

# applying matrix.trace() method
tracem = gfg.trace()

print(tracem)
```

```
[[7]]
```