Python Programming

Numpy

Agenda :

- Introduction
- Installation
- Numpy array
- Creation of Numpy arrays
- Operations
- Reading from file
- Saving to file

Introduction :

NumPy stands for ‘Numerical Python’.

It is a package for data analysis and scientific computing with Python.

NumPy uses a multidimensional array object, and has functions and tools for working with these arrays.

The powerful n-dimensional array in NumPy speeds-up data processing.

Installation :

pip install numpy

pip - Package installer for Python.

pip is a package-management system written in Python and is used to install and manage software packages.

NumPy Array :

NumPy arrays are used to store lists of numerical data, vectors and matrices.

The NumPy library has a large set of routines (built-in functions) for creating, manipulating, and transforming NumPy arrays.

Python language also has an array data structure, but it is not as versatile, efficient and useful as the NumPy array.

The NumPy array is officially called **ndarray** but commonly known as array.

Numpy Array vs Python List

Numpy arrays	Python List
Allocates a fixed size when we create it	Lists grows dynamically
It memory efficient	List do not store efficiently
Elements of np array are of the same data type resulting same size in memory	Elements of python list can be of different data type resulting different size in memory
Advanced mathematical in little time through vectorization	Advanced mathematics takes time

Creation of NumPy Arrays from List :

```
array1 = np.array([10,20,30])
>>> array1
array([10, 20, 30])
```

Creating 1-D array -

```
>>> array2 = np.array([5,-7.4,'a',7.2])
>>> array2
array(['5', '-7.4', 'a', '7.2'], dtype='<U32') #U32 means Unicode-32 data type.
```

Creating 2-D array -

```
array3 = np.array([[2.4,3], [4.91,7],[0,-1]])
>>> array3
array([[ 2.4 , 3. ], [ 4.91, 7. ], [ 0. , -1. ]])
```

Attributes of NumPy Array :

i) **ndarray.ndim**: gives the number of dimensions of the array as an integer value.

ii) **ndarray.shape**: It gives the sequence of integers indicating the size of the array for each dimension. (row, column) format for 2D.

iii) **ndarray.size**: It gives the total number of elements of the array.

iv) **ndarray.dtype**: is the data type of the elements of the array. Common data types are int32, int64, float32, float64, U32, etc.

v) **ndarray.itemsize**: It specifies the size in bytes of each element of the array.

Other Ways of Creating NumPy Arrays :

1. We can specify data type (integer, float, etc.) while creating array using dtype as an argument to array().

Eg - array4 = np.array([[1,2], [3,4]], dtype=float)

```
>>> array4
array([[1., 2.], [3., 4.]])
```

2. We can create an array with all elements initialised to 0 using the function zeros().(float datatype by default)

Eg - array5 = np.zeros((3,4))

```
>>> array5
array([[0., 0., 0., 0.], [0., 0., 0., 0.], [0., 0., 0., 0.]])
```

3. We can create an array with all elements initialised to 1 using the function ones().(by default float)

Eg - array6 = np.ones((3,2))

>>> array6

array([[1., 1.], [1., 1.], [1., 1.]])

4. We can create an array with numbers in a given range and sequence using the arange() function.

Eg - array7 = np.arange(6)

>>> array7

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

Indexing :

Each element of 1-D array is identified or referred using the name of the Array along with the index of that element, which is unique for each element.

For 2-D arrays indexing for both dimensions starts from 0, and each element is referenced through two indexes i and j, where i represents the row number and j represents the column number.

Eg -

>>>array[0,2] ## first row, third column

Slicing :

Slicing is done by defining which part of the array to be sliced by specifying the start and end index values using [start : end] along with the array name.

Eg -

```
>> array8 = np.array([-2, 2, 6, 10, 14, 18, 22])
>>> array8[3:5]    ## Similar to lists in python
array([10, 14])
```

For 2-D Arrays -

```
>>> array9 = np.array([[ -7, 0, 10, 20], [ -5, 1, 40, 200], [ -1, 1, 4, 30]])
>>> array9[1:3, 0:2]     ## row, column
array([[-5, 1], [-1, 1]])
```

Operations on Arrays :

Arithmetic Operations :

Arithmetic operations on NumPy arrays are fast and simple.

When we perform a basic arithmetic operation like addition, subtraction, multiplication, division etc. on two arrays, the **operation is done on each corresponding pair of elements**.

```
>>> array1 = np.array([[3,6],[4,2]])
>>> array2 = np.array([[10,20],[15,12]])
>>> array1 + array2
array([[13, 26], [19, 14]])
```

#Matrix Multiplication

```
>>> array1 @ array2
array([[120, 132], [ 70, 104]]
```

#Exponentiation

```
>>> array1 ** 3
array([[ 27, 216], [ 64, 8]], dtype=int32)
```

#Division

```
>>> array2 / array1
array([[3.33333333, 3.33333333],
 [3.75 , 6. ]])
```

#(Modulo)

```
>>> array2 % array1
array([[1, 2], [3, 0]], dtype=int32)
```

for element-wise operations, size of both arrays must be same.

Transpose :

turns its rows into columns and columns into rows

Eg - >>> array3 = np.array([[10,-7,0, 20], [-5,1,200,40],[30,1,-1,4]])

>>> array3.transpose() **# the original array does not change**

array([[10, -5, 30], [-7, 1, 1], [0, 200, -1], [20, 40, 4]])

Sorting :

>>> array4 = np.array([1,0,2,-3,6,8,4,7])

>>> array4.sort() **## ascending order by default**

>>> array4

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

In 2-D array, sorting can be done along either of the axes i.e., row-wise or column-wise. By default, sorting is done row-wise (i.e., on axis = 1).

It means to arrange elements in each row in ascending order.

```
>>> array5 = np.array([[10,-7,0, 20], [-5,1,200,40],[30,1,-1,4]])
#axis =0 means column-wise sorting
>>> array5.sort(axis=0)
>>> array5
array([[ -5, -7, -1, 4], [ 10, 1, 0, 20], [ 30, 1, 200, 40]])
```

Concatenating Arrays :

Concatenation means joining two or more arrays. Concatenating 1-D arrays means appending the sequences one after another.

NumPy.concatenate() function can be used to concatenate two or more 2-D arrays either row-wise or column-wise.

All the dimensions of the arrays to be concatenated must match exactly except for the dimension or axis along which they need to be joined.

```
>>> array1
array([[ 10, 20], [-30, 40]])
>>> array2
array([[0, 0, 0], [0, 0, 0]])
>>> np.concatenate((array1,array2), axis=1)
array([[ 10, 20, 0, 0, 0], [-30, 40, 0, 0, 0]])
```

Reshaping Arrays :

We can modify the shape of an array using the **reshape()** function.

Attempting to change the number of elements in the array using reshape() results in an error.

Eg - >>> array3 = np.arange(10,22)

>>> array3.reshape(3,4)

array([[10, 11, 12, 13], [14, 15, 16, 17], [18, 19, 20, 21]])

Splitting Arrays :

We can split an array into two or more subarrays.

numpy.split() splits an array along the specified axis. (axis = 0, by default)

```
>>> array([[ 10, -7, 0, 20], [ -5, 1, 200, 40], [ 30, 1, -1, 4], [ 1, 2, 0, 4], [ 0, 1, 0, 2]])
# [1,3] indicate the row indices on which # to split the array
>>> first, second, third = numpy split(array4, [1, 3])
>>> first
array([[10, -7, 0, 20]])
>>> second
array([[ -5, 1, 200, 40], [ 30, 1, -1, 4]])
```

Statistical Operations on Arrays :

1. The max() function finds the maximum element from an array.
2. The min() function finds the minimum element from an array
3. The sum() function finds the sum of all elements of an array.
4. The mean() function finds the average of elements of the array.
5. The std() function is used to find standard deviation of an array of elements.

Loading Arrays from Files :

Using NumPy.loadtxt() :

```
>>> studentdata = np.loadtxt('C:\Users\atalb\Documents\Python training notebooks/marksheet.txt', skiprows=1, delimiter=',', dtype = int)

>>> studentdata

array([[ 1, 36, 18, 57], [ 2, 22, 23, 45],
                         [ 3, 43, 51, 37], [ 4, 41, 40, 60], [ 5, 13, 18, 27]])
```

The parameter skiprows=1 indicates that the first row need to be skipped.

The delimiter specifies whether the values are separated by comma, semicolon, tab or space(default is space).

Specify the data type of the array to be created by specifying through the dtype argument. (By default, dtype is float.)

Using NumPy.genfromtxt() :

As compared to loadtxt(), genfromtxt() can also **handle missing values** in the data file.

dataarray = np.genfromtxt('C:\Users\atalb\Documents\Python training notebooks/marksheet_missing.txt',skip_header=1, delimiter = ',')

>>> dataarray

array([[1., 36., 18., 57.], [2., nan, 23., 45.], [3., 43., 51., nan], [4., 41., 40., 60.], [5., 13., 18., 27.]])

The genfromtxt() function converts missing values and character strings in numeric columns to **nan**.(if we specify dtype as int, convert to -1)

Convert to some specific value using the parameter filling_ values.

Saving NumPy Arrays in Files on Disk :

The **savetxt()** function is used to save a NumPy array to a text file.

Eg -

>>> np.savetxt('C:\Users\atalb\Documents\Python training notebooks/save.txt', studentdata, delimiter=',', fmt='%i')

We have used parameter fmt to specify the format in which data are to be saved. The default is float.

References and Links :

Python Documentation : https://docs.python.org/3/

Numpy Documentation : https://numpy.org/doc/

GitHub link : https://github.com/lunatic-bot/PythonTraining

Thank You