**Roll No : B-05**

**Name : Anuj Somani**

**Batch : TC1**

**Assignment Title : To study and use of Numpy library to store and manipulate data using Numpy array functions.**

**Problem Statement : Study and use of Numpy library.**

**Theory :**

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the *ndarray* object. This encapsulates *n*-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

* NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an *ndarray* will create a new array and delete the original.
* The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
* NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python’s built-in sequences.
* A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today’s scientific/mathematical Python-based software, just knowing how to use Python’s built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

The points about sequence size and speed are particularly important in scientific computing. As a simple example, consider the case of multiplying each element in a 1-D sequence with the corresponding element in another sequence of the same length. If the data are stored in two Python lists, a and b, we could iterate over each element:

c **=** **[]**

**for** i **in** range**(**len**(**a**)):**

c**.**append**(**a**[**i**]\***b**[**i**])**

This produces the correct answer, but if a and b each contain millions of numbers, we will pay the price for the inefficiencies of looping in Python. We could accomplish the same task much more quickly in C by writing (for clarity we neglect variable declarations and initializations, memory allocation, etc.)

**for** **(**i **=** **0;** i **<** rows**;** i**++):** **{**

c**[**i**]** **=** a**[**i**]\***b**[**i**];**

**}**

This saves all the overhead involved in interpreting the Python code and manipulating Python objects, but at the expense of the benefits gained from coding in Python. Furthermore, the coding work required increases with the dimensionality of our data. In the case of a 2-D array, for example, the C code (abridged as before) expands to

**for** **(**i **=** **0;** i **<** rows**;** i**++):** **{**

**for** **(**j **=** **0;** j **<** columns**;** j**++):** **{**

c**[**i**][**j**]** **=** a**[**i**][**j**]\***b**[**i**][**j**];**

**}**

**}**

NumPy gives us the best of both worlds: element-by-element operations are the “default mode” when an *ndarray* is involved, but the element-by-element operation is speedily executed by pre-compiled C code. In NumPy

c **=** a **\*** b

does what the earlier examples do, at near-C speeds, but with the code simplicity we expect from something based on Python. Indeed, the NumPy idiom is even simpler! This last example illustrates two of NumPy’s features which are the basis of much of its power: vectorization and broadcasting.

**Usage :**

To create an array:

* import numpy as np a = np.array([1,2,3])

**Multi-Dimensional Array**

A multidimensional array has more than one column.

*We can consider a multi-dimensional array to be an Excel Spreadsheet — it has columns and rows. Each column can be considered as a dimension.*





Example 2D Array

We can instantiate an array object:

* numpy.array([,.,.,.,])e.g.numpy.array([1,2]) #1Dnumpy.array([[1,2],[10,20]]) #2D#For complex typesnumpy.array([1,2], dtype=complex) #1D complex

**If you want to create a 3-D Array:**

* This will create 3 arrays with 4 rows and 5 columns each with random integers.
* 3DArray = np.random.randint(10, size=(3, 4, 5))

**There are also other types available such as:**

* Boolean
* Integer (signed and unsigned)
* Float
* Complex

**Before I go any further, let’s quickly understand what lists are and when to use arrays over lists**

Just like an array data structure, a list in Python is also a data structure.

A list is mutable and is an ordered sequence of elements. It is flexible and can hold any arbitrary data. Additionally, we can append items to a list efficiently. However, lists take more space than an array.

An array is a thin wrapper around C arrays. You should use a Numpy array if you want to perform mathematical operations. Additionally, we can perform arithmetic functions on an array which we cannot do on a list.

**4. Different Ways To Create An Array**

**Note: numpy and np both refer to the Numpy package here:**

* import numpy as npimport numpy

There are a number of different ways to create an array. This section will provide an overview of the most common methodologies:

* **If you want to create an array without any element:**
* numpy.empty(2) #this will create 1D array of 2 elementsnumpy.empty([2,3]) #this will create 2D array (2 rows, 3 columns each)

**2. If you want to create an array with 0s:**

* numpy.zeros(2) #it will create an 1D array with 2 elements, both 0#Note the parameter of the method is shape, it could be int or a tuple

**3. If you want to create an array with 1s:**

* numpy.ones(2) # this will create 1D array with 2 elements, both 1

**4. If you want to create a Numpy array from a sequence of elements, such as from a list:**

* numpy.asarray([python sequence]) #e.g. numpy.asarray([1,2])

**5. From a buffer in memory:**

We can make a copy of the string in memory:

* x = np.fromstring(‘hi’, dtype=’int8')

Then we can refer to the buffer of the string directly which is memory efficient:

* a = np.frombuffer(x, dtype=’int8')

We can pass in dtype parameter, default is float.

**6. If you want to create a range of elements:**

* import numpy as nparray = np.arange(3)#array will contain 0,1,2

**7. If you want to create an array with values that are evenly spaced:**

* numpy.arange(first, last, step, type)e.g. to create 0-5, 2 numbers apartnumpy.arange(0,6,2) will return [0,2,4]

**8. If you want to create an array where the values are linearly spaced between an interval then use:**

* numpy.linspace(first, last, number)e.g.numpy.linspace(0,10,5) will return [0,2.5,5,7.5,10]

**9. If you want to create an array where the values are log spaced between an interval then use:**

* numpy.logspace(first, end, number)e.g.a= numpy.logspace(1, 15, 4)#results in [1.00000000e+01 4.64158883e+05 2.15443469e+10 1.00000000e+15]

Any base can be specified, Base10 is the default.

**10. Random number generation**

Use the random module of numpy for uniformly distributed numbers:

* np.random.rand(3,2) #3 rows, 2 cols

**5. Adding/Removing/Sorting Elements**

We can perform a number of fast operations on a Numpy array. This makes Numpy a desirable library for the Python users.

**To add elements:**

* a = [0]np.append(a, [1,2]) #adds 1,2 at the end#insert can also be used if we want to insert along a given indexThis will return [0,1,2]

**To delete elements:**

* np.delete(array, 1) #1 is going to be deleted from the arraye.g. a = np.delete([0,1,2], 1) #results in [0,2]

**Sorting**

To sort an array, call the sort(array, axis, kind, orderby) function:

* np.sort(array1, axis=1, kind = 'quicksort')e.g. a = np.sort([[0,3,2],[1,2,3]], axis=1, kind = 'quicksort')results in [[0 2 3] [1 2 3]]

**6. NumPy Array Functions And Attributes**

A ndarray object has a number of attributes, such as:

* **shape: To find the dimensions (numbers of column/row) of an array:**
* array = np.array([[..],[..]])array.shapee.g.a = np.array([[1,2],[3,4]])print(a.shape)It will return (2,2) # rows, columns

We can change the shape (resize) an array by setting the shape property:

* array.shape = (1,2) #1 row, 2 columns

We can also use the reshape() method if you want to change the shape of an array without copying any data:

* array = np.arange(10) # This returns 1d array of 10 elementsarray.reshape(2,5) # This will return an array of 2 rows, 5 columns

We can also set the dimension value to -1 which will let the Numpy infer the dimension from the data.

If we want to flatten an array without returning a copy, we can use the ravel() function:

* array.ravel() # this will reshape the above array as 1d with 10 elements

If we want to flatten an array and produce a copy then we can use the flatten() method:

* a = array.flatten() #this will return an 1d array

**2. resize(x,y) can also be used to resize an array**

**3. If we want to find the number of dimensions of an array:**

* array.ndime.g.a = np.array([[1,2],[3,4]])print(a.ndim)shows 2

**4. If we want to find the length of each element of an array:**

* a = np.array([0,1,2]).itemsizeprint(a)shows 4

**5. If we want to slice a subset of an array:**

* array = np.arange(100)#Get 3rd element:array[2] #prints 2#Get items within indexesarray[3:5] #3 is start, 5 is end, prints [3 4]#Get 3-10 element, step size 4 increments:array[2:9:4] #prints [2 6]#Get all elements from 2nd element onwardsarray[1:] #prints [1-99]#Can also pass in N-Dimensional Indexarray = np.array([[0,1,3],[1,2,4]])print(array[[0,1],[1,2]]) #prints [1 4]

We can split an array along an axis:

* array = np.random.randon(3,5)return [[0.1, 0.5, 1.4, 4.5, 1.2],[8.1, 5.5, 1.4, 6.8, 1.7],[5.3, -0.5, 5.4, 8.5, 6.2]]#Splitfirst, second, third, fourth, fifth = np.split(array, [1,5])

**6. Conditions In Array Slicing**

We can pass in boolean operators e.g.

Get all NAN elements

* array[np.isnan(array)]

**where()** can be used to pass in boolean expressions:

* np.where(array > 2) # will return all elements that meet the criteria

**7. Broadcasting an array**

When a mathematical operation is performed on two arrays of different sizes then the smaller array is broadcasted to the size of the larger array:

* bigger\_array = np.arange(15).reshape(5,3) #5 rows, 3 columns arraysmaller\_array = np.arange(5).reshape(5,1) #5 rows, 1 column arrayfinal\_array = smaller\_array \* bigger\_arrayprint(final\_array)This prints multiplied broadcasted array of 5 rows, 3 columns[[ 0 0 0] [ 3 4 5] [12 14 16] [27 30 33] [48 52 56]]

The key to note is that the broadcasting is compatible with two arrays where the number of columns of the first array is the same as the number of rows of the second array, or if any of the arrays has a length of 1.

**8. Transposing Array**

* array.Trollaxis, swapaxes, transpose are also available transpose functions.

**9.**

**To join arrays:**

Concatenate: Arrays are joined based on the axis

We can also stack them using vstack or hstach methods.

* a = [1,2]b= [3,4]c = [a,b]#input: [[1, 2], [3, 4]]np.concatenate(c)#output: [1 2 3 4]np.stack(c)#output:#[[1 2]#[3 4]]np.hstack(c)#ouput:#[1 2 3 4]np.vstack(c)#output:#[[1 2]#[3 4]]

**10. String Operations**

A large number of string operations can be utilised e.g.

* add(), upper(), lower(), replace() etc.

**11. To create a deep copy of numpy array:**

* new\_array = np.copy(array)

To repeat an array, we can use the repeat() or tile() functions.

The repeat(n) will simply repeat each element n times. The n could also be an array whereby each element will be repeated differently based on the value of n e.g. [1,5] means we need to repeat the first element once and the second element 5 times.

For multidimensional arrays, we can pass in the axis attribute.

tile(array, (n,m)) is slightly different because along with repeating the elements, it also tiles/stacks the items for n number of rows and m number of columns.

**12. ufunc: Reduce and Accumulate**

We can also provide our own vectorised operations.

reduce() takes a single array and aggregates its values.

Accumulate() aggregates the values and preserves the intermediate aggregate results.

We can also write our own ufuncs as long as the function takes in array(s) and returns a value.

We will have to use np.fromnpfunc(my\_new\_ufunc, elements) to create the new func and then execute it on NumPy arrays.

**7. Structured Arrays**

If we want to create an array with elements of multiple data types then we can create a structured array. We can set the dtype which is a list of tuples containing the name and the type of the elements.

* type = [('column\_1', np.int32, 'column\_2', np.float64])array = np.array([1,2], [2.4, -1], dtype=type)

Structured arrays are faster than pandas DataFrame because they consume lower memory as each element is represented as a fixed number of bytes, they are lean and hence efficient low-level arrays, and also can be seen as a tabular structure.

**8. Mathematical Functions**

Numpy offers a range of powerful Mathematical functions. This is one of the reasons why the library is popular in quantitative fields. Additionally, a number of libraries are built on top of Numpy due to the fact that it has a rich set of mathematical features.

**Add, Subtract, Multiply, Divide, Power, Mod**

To perform basic arithmetic functions on two arrays a and b:

* a = [1,2]b= [3,4]c = np.add(a, b)c = np.subtract(a, b)c = np.multiply(a, b)c = np.divide(a, b)c = np.power(a, b) c = np.power(a, 2)#to get remainderc= np.mod(a, b)c = np.remainder(a, b)

**Rounding, Ceil, Floor**

To change the precision of all elements of an array:

* np.around(array, 4) # 4dpnp.ceil(array) #1.8 will become 2np.floor(array) #1.8 will become 1

**Trigonometric**

* array = [0, 1]np.sin(array)np.cos(array)np.tan(array)np.arcsin(array)np.arccos(array)np.arctan(array)

A number of complex number functions can also be applied such as getting real or imaginary parts of an array with complex numbers.

**Statistical**

There are also a large number of statistical functions available:

* a = [1,2]np.amin(a, 0) #min in the axisnp.amax(a, 0) #max in the axisnp.percentile(a, 10)#Additionally, following functions are available:np.median(a)np.std(a)np.average(a)np.mean(a)np.var(a)

**Applications :**

Numpy Applications with Other Libraries

1. NumPy with Pandas

Pandas is one of the most important libraries in python for data analysis. Pandas provide high performance, fast analysis, and data cleaning. We use it to manipulate data structures and have data analysis tools.

It consists of a data frame object. It interoperates with NumPy for faster computations. When we use both the libraries together it is a very helpful resource for scientific computations.

2. NumPy with Matplotlib

Matplotlib is a module in NumPy. It is a very helpful tool to work with graphical representations. It consists of a wide range of functions to plot graphs and also manipulate them.

This combination can replace the functionalities of MatLab. It is used to generate the graphs of the results. We enhance it further with the use of graphic toolkits like PyQt and wxPython.

3. NumPy with SciPy

Scipy is an open-source library in Python. It is the most important scientific library in python. It has been built upon the functionalities of NumPy.There are advanced functionalities in SciPy for scientific computations.

We can combine it with NumPy for greater mathematical performance. The combination helps in the implementation of complex scientific operations.

4. NumPy with Tkinter

Tkinter is a standard library for GUI. We use Tkinter for the GUI representation of the NumPy data. Its combination with NumPy can implement fast and easy GUIs. The use of Tkinter along with NumPy is user friendly. We can easily convert the array objects into image objects.

**Code with Output :**

**Sample :**

import numpy as np

arr = np.array([1,4,6,8])

print(arr)

Output : [1 4 6 8]

import numpy as np

brr = np.array([(1,5,4,2,7),(7,8,3,5,1)],dtype=float)

print(brr)

output :

[[1.,5.,4.,2.,7.]

[7.,8.,3.,5.,1.]]

**Conclusion :**

we studied use of Numpy library to store and manipulate data using Numpy array functions.