Syllabus - Module V

Data Processing: The os and sys modules. Introduction to file I/O - Reading and writing text files, Manipulating binary files. NumPy - Basics, Creating arrays, Arithmetic, Slicing, Matrix Operations, Random numbers. Plotting and visualization. Matplotlib - Basic plot, Ticks, Labels, and Legends. Working with CSV files. – Pandas - Reading, Manipulating, and Processing Data.

OS Module in Python

- The OS module in python provides functions for interacting with the operating system.
- OS, comes under Python's standard utility modules.
- This module provides a portable way of using operating system dependent functionality.

Creating Directory:

We can create a new directory using the mkdir() function from the OS module.

import os

os.mkdir("d:\\tempdir")

Changing the Current Working Directory:

We must first change the current working directory to a newly created one before doing any operations in it. This is done using the chdir() function.

```
import os
os.chdir("d:\\tempdir")
```

- There is a getcwd() function in the OS module using which we can confirm if the current working directory has been changed or not.
- In order to set the current directory to the parent directory use ".." as the argument in the chdir() function.

Removing a Directory

- The rmdir() function in the OS module removes the specified directory either with an absolute or relative path.
- However, we can not remove the current working directory. Also, for a directory to be removed, it should be empty. For example, tempdir will not be removed if it is the current directory.

List Files and Sub-directories:

- The listdir() function returns the list of all files and directories in the specified directory.
- If we don't specify any directory, then list of files and directories in the current working directory will be returned.

Scan all the directories and subdirectories:

- Python method walk() generates the file names in a directory tree by walking the tree either top-down or bottom-up.

sys Module in Python

The sys module provides functions and variables used to manipulate different parts of the Python runtime environment.

sys.argv

returns a list of command line arguments passed to a Python script. The item at index 0 in this list is always the name of the script. The rest of the arguments are stored at the subsequent indices.

sys.exit

This causes the script to exit back to either the Python console or the command prompt. This is generally used to safely exit from the program in case of generation of an exception.

sys.maxsize

Returns the largest integer a variable can take.

sys.path

This is an environment variable that is a search path for all Python modules.

sys.version

This attribute displays a string containing the version number of the current Python interpreter.

Introduction to File I/O

Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory.

When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order:

(1) Open a file

(2) Read or write (perform operation)

(3) Close the file

Opening Files in Python:

Python has a built-in open() function to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

Files are divided into two category:

Text Files - simple texts in human readable format.

Binary Files - have binary data which is understood by the computer.

Syntax: file_object = open(file_name [, access_mode])

```
f = open("test.txt")    # open file in current directory
f = open("C:/Python38/README.txt")  # specifying full path
```

file_name: File name contains a string type value containing the name of the file which we want to acess.

access_mode: The value of access_mode specifies the mode in which we want to open the file, i.e, read, write, append, etc.

Access Modes	Description
r	Opens a file for reading only.
rb	Opens a file for reading only in binary format.
r+	Opens a file for both reading and writing.
rb+	Opens a file for both reading and writing in binary format.
w	Opens a file for writing only.
wb	Opens a file for writing only in binary format.
w+	Opens a file for both writing and reading.
wb+	Opens a file for both writing and reading in binary format.
a	Opens a file for appending.
ab	Opens a file for appending in binary format.
a+	Opens a file for both appending and reading.
ab+	Opens a file for both appending and reading in binary format.

```
f = open("test.txt")           # equivalent to 'r'
f = open("test.txt",'w')       # write in text mode
f = open("img.bmp",'rb+')      # read and write in binary mode
```

Closing a File:

When the operations that are to be performed on an opened file are finished, we have to close the file in order to release the resources. The closing of file is done with a built-in function close()

Syntax: fileobject.close()

Example:

```
# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)

# Close opend file
fo.close()
```

- Most recommended way of using file is along with ‘with’ keyword.
- Because, once the ‘with’ block exits, file is automatically closed and file object is destroyed.

Eg.,

```
with open("test.txt") as f:
print(f.read())
```

- The closing method will close the file instantly, It is not safe.
- When we are performing some operations on a file and an exception occurs, the code exits without closing the file.
- Hence , we should use try...finally block

The file Object Attributes:

file.closed : Returns true if file is closed, false otherwise.

file.mode : Returns access mode with which file was opened.

file.name : Returns name of the file.

file.softspace : Returns false if space explicitly required with print, true otherwise.

Eg.,

```
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not : ", fo.closed
print "Opening mode : ", fo.mode
print "Softspace flag : ", fo.softspace
```

Output:

```
Name of the file: foo.txt
Closed or not : False
Opening mode : wb
Softspace flag :  0
```

Reading and Writing Files

- Writing to a File:

The write() method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write() method does not add a newline character ('\n') to the end of the string.

Syntax : fileObject.write(string)

writelines() method writes the items of a list to the file.

- Reading from a File:

The read() method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

Syntax : fileObject.read([size])

size - This parameter gives the number of bytes to be read from an opened file.

✓readline() - This method is used to read the file contents line by line.

whenever we write fileobject.readline(), it prints one line from that file and continues in this way until the end of the file.

✓readlines() - This method returns a list containing each line in the file as a list item.

Use the hint parameter to limit the number of lines returned. If the total number of bytes returned exceeds the specified number, no more lines are returned.

- We can also use the following method to read the content from the file

```
f = open("test.txt", "r+")
for each in f:
    print(each)
```

- split() using file handling:

This splits the variable when space is encountered. You can also split using any characters as we wish.

```
with open("file.text", "r") as file:
data = file.readlines()
for line in data:
    word = line.split()
    print (word )
```

File positions:

The tell() method returns the current file position in a file stream.

The seek() method is used to change the position of the File Handle to a given specific position

File handle is like a cursor, which defines from where the data has to be read or written in the file.

Syntax: f.seek(offset, from_what), where f is file pointer

Offset: Number of postions to move forward

from_what: It defines point of reference.

The reference point is selected by the from_what argument. It accepts three values:

0: sets the reference point at the beginning of the file

1: sets the reference point at the current file position

2: sets the reference point at the end of the file

Renaming a File:

- This is done by using rename() method.
- This method is passed with two arguments, the current filename and the new filename

Syntax : os.rename(current_filename, new_filename)

Deleting a File:

- This is done with the help of the remove() method.
- It takes the filename as an argument to be deleted.

Syntax : os.remove(filename)

File Related Methods

Sl. No.	Methods with Description
1	file.close() Close the file. A closed file cannot be read or written any more.
2	file.flush() Flush the internal buffer memory.
3	file.fileno() Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.
4	file.__next__() Returns the next line from the file each time it is being called
5	file.read([size]) Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes).
6	file.readline([size]) Reads one entire line from the file. A trailing newline character is kept in the string.
7	file.readlines([sizehint]) It reads until the end of the file using readline. It returns the list of lines read.

File Related Methods

Sl. No.	Methods with Description
8	file.seek(offset) Sets the file's current position
9	file.tell() Returns the file's current position
10	file.truncate([size]) Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.
11	file.write(str) Writes a string to the file. There is no return value.
12	file.writelines(sequence) Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings
13	file.isatty() Returns True if the file is connected to a tty(-like) device, else False.

Programming Exercise:

(1) Write a program to read a file line by line and store it into a list.

(2) Write a program to find the longest word.

(3) Write a program to remove newline characters from a file.

(4)Write a program to read contents from a file and write the content to another file after eliminating duplicates.

(5) Write a Python program to count the frequency of words in a file.

Reading and Writing to a Binary File

- To open a file in binary format, add 'b' to the mode parameter. Hence the "rb" mode opens the file in binary format for reading, while the "wb" mode opens the file in binary format for writing.
- Unlike text mode files, binary files are not human readable. When opened using any text editor, the data is unrecognizable.
- The following code stores a list of numbers in a binary file.

```
f=open("binfile.bin","wb")
num=[5, 10, 15, 20, 25]
arr=bytearray(num)
f.write(arr)
f.close()
```

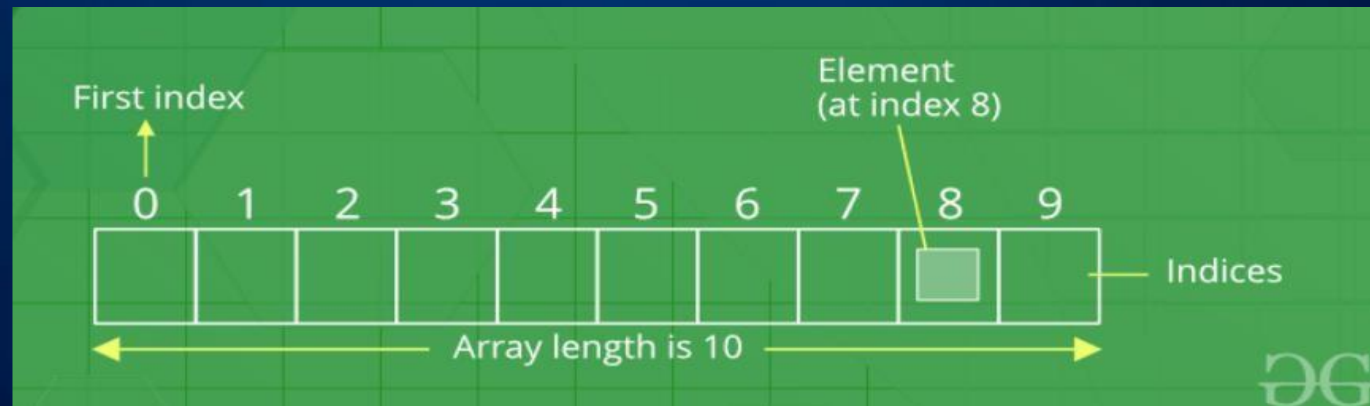
Reading and Writing to a Binary File

- To read the binary file, the output of the read() method is casted to a list using the list() function.

```
f=open("binfile.bin","rb")
num=list(f.read())
print (num)
f.close()
```

Python Array

- An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.
- Array can be handled in Python by a module named array. They can be useful when we have to manipulate only a specific data type values.
- If you create arrays using the array module, all elements of the array must be of the same type.



- Creating a Array

Array in Python can be created by importing array module.

array(data_type, value_list)

The above syntax is used to create an array with data type and value list specified in its arguments.

```
import array as arr
a = arr.array('d', [1.1, 3.5, 4.5])
print(a)
```

output:

```
array('d', [1.1, 3.5, 4.5])
```

Here, we created an array of float type. The letter d is a type code. This determines the type of the array during creation.

Commonly used type codes are listed as follows:

Code	C Type	Python Type	Min bytes
b	signed char	int	1
B	unsigned char	int	1
u	Py_UNICODE	Unicode	2
h	signed short	int	2
H	unsigned short	int	2
i	signed int	int	2
I	unsigned int	int	2
l	signed long	int	4
L	unsigned long	int	4
f	float	float	4
d	double	float	8

Activate Wi
Go to Settings

- Accessing Python Array Elements

We use indices to access elements of an array:

```
import array as arr
a = arr.array('i', [2, 4, 6, 8])

print("First element:", a[0])
print("Second element:", a[1])
print("Last element:", a[-1])
```

Output:

```
First element: 2
Second element: 4
Last element: 8
```

- Slicing Python Arrays

We can access a range of items in an array by using the slicing operator :

```
import array as arr

numbers_list = [2, 5, 62, 5, 42, 52, 48, 5]
numbers_array = arr.array('i', numbers_list)

print(numbers_array[2:5]) # 3rd to 5th
print(numbers_array[:-5]) # beginning to 4th
print(numbers_array[5:])  # 6th to end
print(numbers_array[:])   # beginning to end
```

• Changing and Adding Elements

Arrays are mutable; their elements can be changed in a similar way as lists

```
import array as arr
numbers = arr.array('i', [1, 2, 3, 5, 7, 10])

numbers[0] = 0
print(numbers)     # Output: array('i', [0, 2, 3, 5, 7, 10])

numbers[2:5] = arr.array('i', [4, 6, 8])
print(numbers)     # Output: array('i', [0, 2, 4, 6, 8, 10])
```

We can add one item to the array using the append() method, or add several items using the extend() method.

- Removing Python Array Elements

We can delete one or more items from an array using Python's del statement.

```
import array as arr

number = arr.array('i', [1, 2, 3, 3, 4])
del number[2]  # removing third element
print(number)  # Output: array('i', [1, 2, 3, 4])

del number  # deleting entire array
print(number)  # Error: array is not defined
```

NumPy (Numerical Python)

NumPy is a library consisting of multidimensional array objects and a collection of routines for processing those arrays. Using NumPy, mathematical and logical operations on arrays can be performed.

Using NumPy, a developer can perform the following operations –

- ✓ Mathematical and logical operations on arrays.
- ✓ Fourier transforms and routines for shape manipulation.
- ✓ Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

What’s the difference between a Python list and a NumPy array?

- NumPy gives you an enormous range of fast and efficient ways of creating arrays and manipulating numerical data inside them.
- While a Python list can contain different data types within a single list, all of the elements in a NumPy array should be homogeneous.
- The mathematical operations that are meant to be performed on arrays would be extremely inefficient if the arrays weren’t homogeneous.

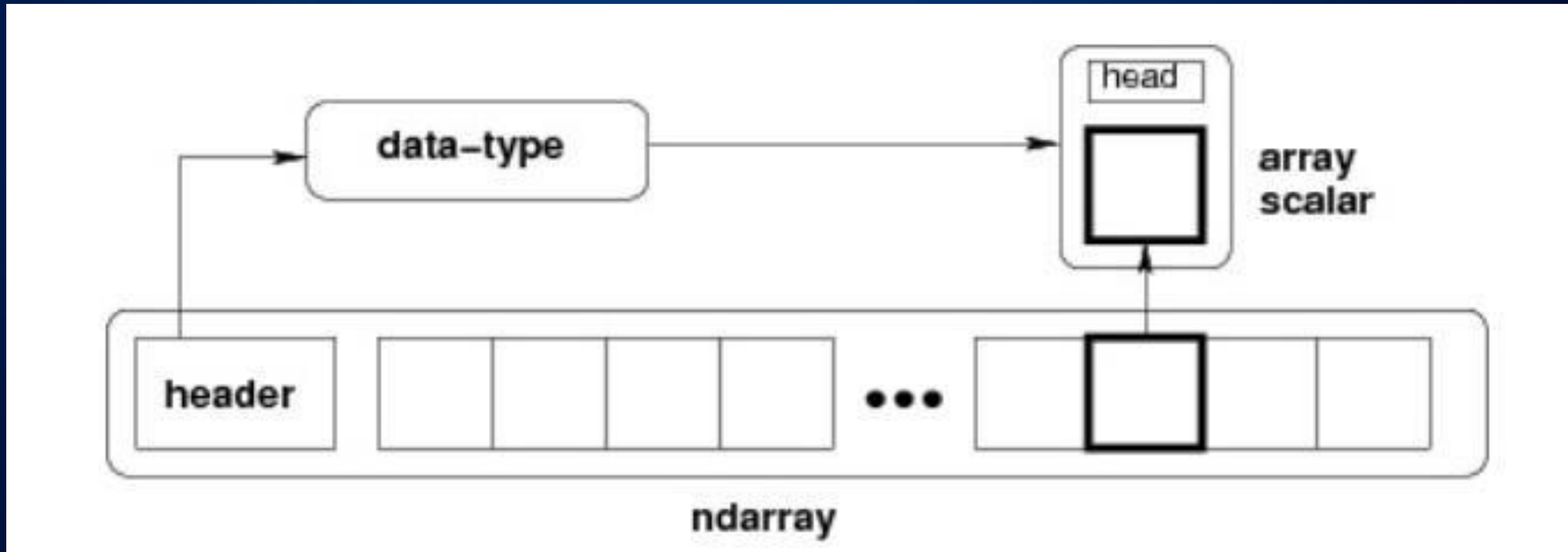
ndarray Object

The most important object defined in NumPy is an N-dimensional array type called ndarray. It describes the collection of items of the same type. Items in the collection can be accessed using a zero-based index.

Every item in an ndarray takes the same size of block in the memory. Each element in ndarray is an object of data-type object (called dtype).

Any item extracted from ndarray object (by slicing) is represented by a Python object of one of array scalar types.

The following diagram shows a relationship between ndarray, data type object (dtype) and array scalar type –



The basic ndarray is created using an array function in NumPy as follows –

numpy.array

Creating Arrays

```
import numpy as np
a = np.array([1,2,3,4])
print(a)
b = np.array([(1,2,3),(4,5,6)], dtype = float)
print(b)
c = np.array([(1,2,3),(4,5,6),(7,8,9)])
print(c)
```

Output:

```
[1 2 3 4]
```

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

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

ndarray Object - Parameters

<u>Some important attributes of ndarray object</u>

(1) ndarray.ndim

ndim represents the number of dimensions (axes) of the ndarray.

(2) ndarray.shape

shape is a tuple of integers representing the size of the ndarray in each dimension.

(3) ndarray.size

size is the total number of elements in the ndarray. It is equal to the product of elements of the shape.

(4) ndarray.dtype

dtype tells the data type of the elements of a NumPy array. In NumPy array, all the elements have the same data type.

(5) ndarray.itemsize

itemsize returns the size (in bytes) of each element of a NumPy array.

```
import numpy as np
a = np.array([[[1,2,3],[4,3,5]],[[3,6,7],[2,1,0]]])
print("The dimension  of array a is:", a.ndim)
print("The size of the array  a is: ", a.shape)
print("The total no: of elements in array a is: ", a.size)
print("The datatype of elements in array a is: ", a.dtype)
print("The size of each element in array a is: ", a.itemsize)
```

Output:

```
The dimension  of array a is: 3
The size of the array a is:  (2, 2, 3)
The total no: of elements in array a is:  12
The datatype of elements in array a is:  int32
The size of each element in array a is:  4
```

ndarray Object - Attributes

Some important attributes of ndarray object

Examples:

#creation of 1D array

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

Output:

[1 2 3 4 5]

#creation of more than 1D array

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

Output:

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

```
#minimum dimensions
import numpy as np
a = np.array([1,2,3,4,5,6], ndmin = 2)
print(a)
```

Output:
[[1,2,3,4,5,6]]

```
#dtype parameter
import numpy as np
a = np.array([1,2,3], dtype = complex)
print(a)
```

Output:
[1.+0.j 2.+0.j 3.+0.j]

NumPy supports a much greater variety of numerical types than Python does. The following table shows different scalar data types defined in NumPy.

Sl. No.	Data Types & Description
1.	bool_ Boolean (True or False) stored as a byte
2.	int_ Default integer type (same as C long; normally either int64 or int32)
3.	intc Identical to C int (normally int32 or int64)
4.	intp Integer used for indexing (same as C ssize_t; normally either int32 or int64)
5.	int8 Byte (-128 to 127)
6.	int16 Integer (-32768 to 32767)

7.	**int32** **Integer (-2147483648 to 2147483647)**
8.	int64 Integer (-9223372036854775808 to 9223372036854775807)
9.	uint8 Unsigned integer (0 to 255)
10.	uint16 Unsigned integer (0 to 65535)
11.	uint32 Unsigned integer (0 to 4294967295)
12.	uint64 Unsigned integer (0 to 18446744073709551615)
13.	float_ Shorthand for float64
14.	float16 Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
15.	float32 Single precision float: sign bit, 8 bits exponent, 23 bits mantiss
16.	float64 Double precision float: sign bit, 11 bits exponent, 52 bits mantissa

Different ways of creating an Array

There are 8 ways for creating arrays using NumPy Package

(1) using array()

(2) using arange()

(3) using linspace()

(4) using logspace()

(5) using zeros()

(6) using ones()

(7) using empty()

(8) using eye()

(1) Using array()

Numpy arrays can be created very easily by passing a list to the function numpy.array().

Eg.,

```
numpy_array = np.array([1,2,3,4)]
print(numpy_array)
```

(2) Using arange()

The arange() function is one of the Numpy's most used method for creating an array within a specified range.

Syntax: arange(start, end, step, dtype)

Among all of the arguments only end is mandatory and by default start=0 and step=1.

```
import numpy as np
a = np.arange(0,11,2)
print(a)

b = np.arange(50,121,4)
print(b)

c = np.arange(15)
print(c)
```

Output:

```
[ 0  2  4  6  8 10]
[ 50  54  58  62  66  70  74  78  82  86  90  94  98 102 106 110 114 118]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

(3) Using linspace()

- In the arange() function, we had control over where to start the Numpy array from, where to stop and step points but with linspace() we can maintain a proper linear stepping or spacing between array elements value while generating the array.
- linspace() function takes arguments: start index, end index and the number of elements to be outputted.
- These number of elements would be linearly spaced in the range mentioned.

Syntax: linspace(start_index, end_index, num_of_elements)

```
import numpy as np
a = np.linspace(15,75,10)
print(a)

b = np.linspace(1,10,10)
print(b)
```

Output:

```
[15.         21.66666667 28.33333333 35.         41.66666667 48.33333333
 55.         61.66666667 68.33333333 75.        ]

[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

(4) Using logspace()

In linspace() the gap between two elements is fixed.
The logspace() function returns number spaces evenly w.r.t interval on a log scale.
In logspace(), the spacing between two numbers depends on the log value.

```
import numpy as np
a = np.logspace(1, 40, 5)
print(a)
```

Output:

```
[1.00000000e+01 5.62341325e+10 3.16227766e+20 1.77827941e+30
 1.00000000e+40]
```

(5) Using zeros()

zeros() returns a new array with zeros.

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

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.

(6) Using ones()

ones() function is used to create a matrix full of ones.

Syntax: numpy.ones(shape, dtype=float, order='C')

```
import numpy as np
a = np.zeros(shape = (2,3), order = 'C' )
print(a)
a = np.ones(shape = (3,3), dtype = int, order = 'C' )
print(a)
```

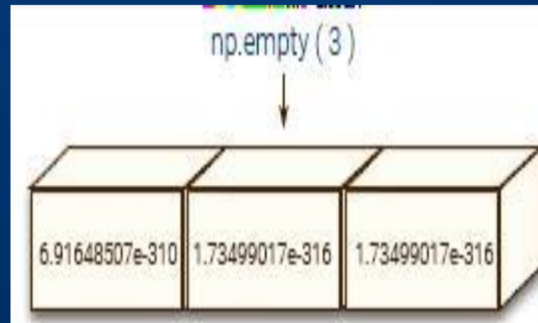
Output:

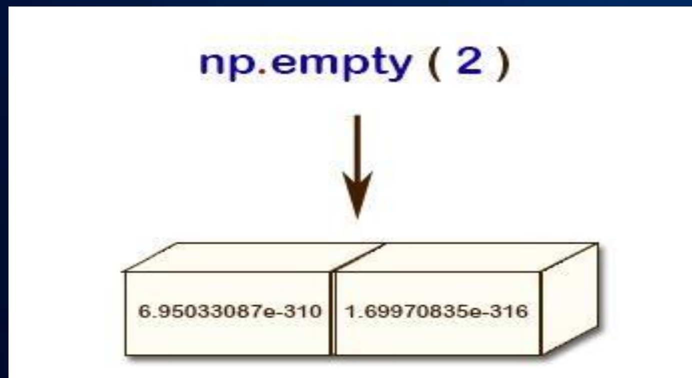
```
[[0. 0. 0.]
 [0. 0. 0.]]
[[1 1 1]
 [1 1 1]
 [1 1 1]]
```

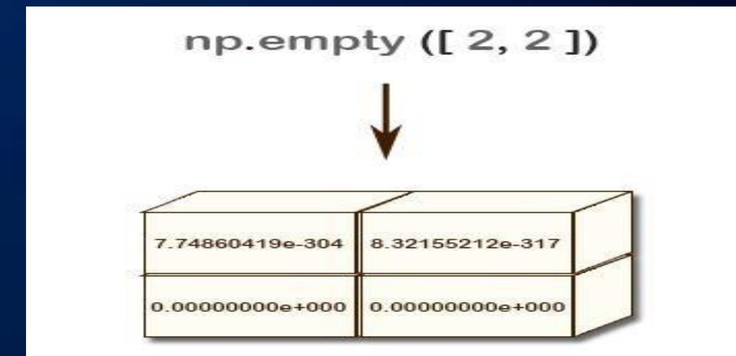
(7) using empty()

The empty() function is used to create a new array of given shape and type, without initializing entries

Syntax: numpy.empty(shape, dtype, order)







(8) Using eye()

The eye() function is used to create a 2-D array with ones on the diagonal and zeros elsewhere.

It returns an array where all elements are equal to zero, except for the k-th diagonal, whose values are equal to one

Syntax: numpy.eye(N, M=None, k=0, dtype=<class 'float'>, order='C')

N	*Number of rows in the output.*
M	*Number of columns in the output. If None, defaults to N.*
k	*Index of the diagonal: 0 (the default) refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.*
dtype	*Data-type of the returned array.*
order	*Whether the output should be stored in row-major or column-major order in memory*

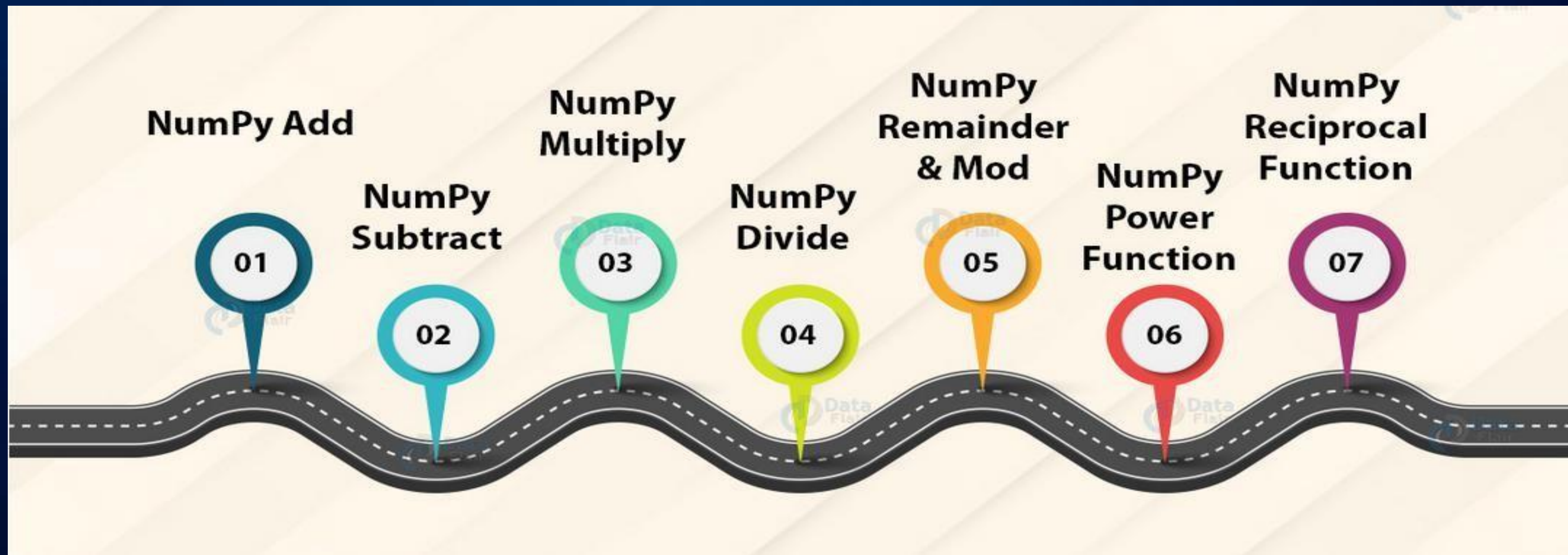
```
import numpy as np
a = np.eye(3, 3, 0, dtype=int, order='C')
print(a)
```

Output:

```
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

Arithmetic Operations with NumPy Array

- The arithmetic operations with NumPy arrays perform element-wise operations, this means the operators are applied only between corresponding elements.
- Arithmetic operations are possible only if the array has the same structure and dimensions.



Basic operations : with scalars

```
import numpy as np
a = np.array([1,2,3,4,5])
b = a+1
print(b)
c = 2**a
print(c)
```

Output:

```
[2 3 4 5 6]
[ 2  4  8 16 32]
```

```
import numpy as np
a = np.array([1,2,3,4,5])
b = np.ones(5) + 1
c = a - b
print(c)
d = (a*b)
print(d)
e = np.arange(5)
print(e)
f = 2 ** (e+1) - e
print(f)
```

Output:

```
[-1.  0.  1.  2.  3.]
[ 2.  4.  6.  8. 10.]
[0 1 2 3 4]
[ 2  3  6 13 28]
```

We have both functions and operators to perform Arithmetic Operations

(1) NumPy add Function & add Operator

This function is used to add two arrays. If we add arrays having dissimilar shapes we get “Value Error”.

We can also use the add operator “+” to perform addition of two arrays.

(2) NumPy subtract Function & subtract Operator

This function to output the difference of two arrays. If we subtract two arrays having dissimilar shapes we get “Value Error”.

We can also use the subtract operator “-” to produce the difference of two arrays.

(3) NumPy Multiply function & Multiply Operator

We use this function to output the multiplication of two arrays. We cannot work with dissimilar arrays.

We can also use the multiplication operator “*” to get the product of two arrays.

(4) NumPy Divide function & Divide Operator

We use this function to output the division of two arrays. We cannot divide dissimilar arrays.

We can also use the divide operator “/” to divide two arrays.

```
import numpy as np
a = np.array([7, 3, 4, 5, 1])
b = np.array([3, 4, 5, 6, 7])
print(a+b)
print(np.add(a, b))
print("---------------------")
print(a-b)
print(np.subtract(a, b))
print("---------------------")
print(np.multiply(a, b))
print(a*b)
print("---------------------")
print(np.divide(a, b))
print(a/b)
```

Output:

```
[10  7  9 11  8]
[10  7  9 11  8]
---------------------
[ 4 -1 -1 -1 -6]
[ 4 -1 -1 -1 -6]
---------------------
[21 12 20 30  7]
[21 12 20 30  7]
---------------------
[2.33333333 0.75       0.8
0.83333333 0.14285714]
[2.33333333 0.75       0.8
0.83333333 0.14285714]
```

(4) NumPy Mod and Remainder function

We use both the functions to output the remainder of the division of two arrays.

(5) NumPy Power Function

This Function treats the first array as base and raises it to the power of the elements of the second array.

(6) NumPy Reciprocal Function

This Function returns the reciprocal of all the array elements.

```
import numpy as np
a = np.array([2, 10, 6, 5, 8])
b = np.array([3, 5, 2, 1, 7])

print(np.remainder(a, b))
print(np.mod(a, b))
print("_______________")
print(np.power(a, b))
print("_______________")
print(np.reciprocal(a))
```

Output:

```
[2 0 0 0 1]
[2 0 0 0 1]
---------------------
[      8  100000      36       5 2097152]
---------------------
[0 0 0 0 0]
```

Trigonometric Functions –

NumPy has standard trigonometric functions which return trigonometric ratios for a given angle in radians.

- ✓ The np.sin() NumPy function help to find sine value of the angle in degree and radian.
- ✓ The np.cos() NumPy function help to find cosine value of the angle in degree and radian.
- ✓ The np.tan() NumPy function help to find tangent value of the angle in degree and radian.

sine value of angle in degree and radians

```
import numpy as np
sin_90 = np.sin(90)
print("sine value of angle 90 in degree: ", sin_90)
sin90 = np.sin(90*(np.pi/180))
print("sine value of angle 90 in radians: ",sin90)
```

Output:

sine value of angle 90 in degree: 0.8939966636005579

sine value of angle 90 in radians: 1.0

cosine value of angle in degree and radians

```
import numpy as np
cos_180 = np.cos(180)
print("cosine value of angle 180 in degree: ", cos_180)
cos180 = np.cos(180*(np.pi/180))
print("cosine value of angle 180 in radians: ", cos180)
```

Output:

```
cosine value of angle 180 in degree:  -0.5984600690578581
cosine value of angle 180 in radians:  -1.0
```

tangent value of angle in degree and radians

```
import numpy as np
tan_60 = np.tan(60)
print("tangent value of angle 60 in degree: ", tan_60)
tan60 = np.tan(60*(np.pi/180))
print("tangent value of angle 60 in radians: ", tan60)
```

Output:

```
tangent value of angle 60 in degree:  0.320040389379563
tangent value of angle 60 in radians:  1.7320508075688767
```

Comparison

Comparing two NumPy arrays determines whether they are equivalent by checking if every element at each corresponding index are the same.

Method 1 :

use the == operator to compare two NumPy arrays to generate a new array object. Call ndarray.all() with the new array object as ndarray to return True if the two NumPy arrays are equivalent.

```
import numpy as np
a = np.array([1, 2, 3, 4])
b = np.array([1, 2, 3, 4])
print("Result of element wise comparison: ")
print(a == b)
print("Result of whole comparison: ")
c = a == b
print(c.all())
```

Output:

```
Result of element wise comparison:
[ True  True  True  True]
Result of whole comparison:
True
```

We can also use greater than, less than and equal to operators to compare. To understand, have a look at the code below.

Syntax : numpy.greater(x1, x2)

Syntax : numpy.greater_equal(x1, x2)

Syntax : numpy.less(x1, x2)

Syntax : numpy.less_equal(x1, x2)

Thank You