NumPy.

Fundamental package for scientific computing in Python

→ It provides

- multidimensional array.
- various derived objects (masked arrays & matrices)
- 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.

→ At the core of the NumPy package is the ndarray object.

Difference b/w NumPy array & Python sequences

Numpy ndarray	Python lists, seq
① They have fixed size at creation	① can grow dynamically.
② All the elements must be of same data type	② opposite
③ facilitate advanced mathematical ops can be executed with less code	③ Py seq takes more code.

→ The point about seq size and speed are particularly important in scientific computing.

eg: consider 2 lists a & b.

```
→
c = []
for i in range(len(a)):
    c.append(a[i] * b[i])
```

```
→ for (i=0; i<rows; i++) {
    c[i] = a[i] * b[i];
}
```

→ OR for nd array

```
for (i=0; i<rows; i++) {
    for (j=0; j<columns; j++) {
        c[i][j] = a[i][j] * b[i][j];
    }
}
```

→ Numpy gives the best soln

In above loops takes more time for huge data.

so in NumPy element-by-element operations are the "default mode"

But here in NumPy execution happens at high speed bcz of pre-compiled C code.

$\boxed{C = a * b}$

Why NumPy is Fast

① vectorization describes the absence of any explicit looping, indexing etc.

② vectorized code is more concise & easier to read. reducing bugs as well

③ The code will be resembling like standard mathematical notation.

④ without vectorization code could be "difficult to read for loops"

→ Broadcasting common word for implicit element-by-element behaviour of operations. All ops of Numpy is this.

→ Numpy fully supports OOPS
→ ndarray is a class having numerous methods & attributes.

Numpy Installation

→ using conda
→ using pip

Numpy

→ Numpy main object is the homogeneous multidimensional array.
→ In Numpy dimensions are called axes

→ ndarray is also aliased as array class
→ numpy.array is not same as array.array
↓
Python library which only handles 1D array.

1. ndarray.ndim
   no of axes of the array.
2. ndarray.shape.
   tuple of integers indicating the size of each axes
   for matrix (n×m) shape will be (n, m)
3. ndarray.size
   if (n, m) = (3×4)
   size = 12 → i.e no of elements in a matrix
4. ndarray.dtype
   eg: var.dtype.name
   o/p numpy.int32
   or
   numpy.int16
   .float64.
5. ndarray.itemsize
   the size of bytes of each element of the array.
6. ndarray.data
   o/p actual elements of array
   No need to use this we access using indexing.

Eg:

```
import numpy as np
a=np.arange(15).reshape
                    (3,5)
a
o/p array([[ 0, 1, 2, 3, 4],
           [5, 6, 7, 8, 9],
           [10, 11, 12, 13, 14]])

a.shape        #(3,5)
a.ndim         # 2
a.dtype.name   # 'int64'
a.itemsize     # 8
a.size         # 15
type(a)        # <class 'numpy.ndarray'>
b = np.array([6, 7, 8])
```

Eg
→ b = np.array([1.2, 3.5, 5.1])
b.dtype # dtype('float64')

Eg
→ b = np.array([(1.5, 2, 3),
(4, 5, 6)])

Eg
→ c = np.array([[1, 2],
[3, 4]], dtype = complex)

array([[1+0.j, 2+0.j],
[3 . . . .]])

***
Eg: np.zeros((3, 4))

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot \end{bmatrix}$$

np.ones((2, 3, 4), dtype = np.int16)

$$\begin{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{bmatrix}$$

np.empty((2, 3))
→ random exponential nos like 3.73..e-262

Eg: np.arange(10, 30, 5)
array([10, 15, 20, 25])
np.arange (0, 2, 0.3)
array([0, 0.3, 0.6, 0.9, 1.2,
1.5, 1.8])

Eg: from numpy import pi
np.linspace (0, 2, 9)
# array([0., 0.25, ... 2.])
x = np.linspace (0, 2*pi, 100)
= np.sin(x)

Routines

① Array creation routines
→ np.zeros_like(x)
x = np.arange(6)
x = x.reshape((2, 3))

Note: lly we have
→ zeros → arange
→ ones → linspace
→ ones_like → array
→ empty
→ empty_like

Basic Ops

Eg: c = a-b # subtraction of matrix
b**2 # sqr of a matrix
10*np.sin(a)

2 Types of Matrix Product

① element wise → * operator
② matrix product → @, dot operator

$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ $B = \begin{bmatrix} 2 & 0 \\ 3 & 4 \end{bmatrix}$

C = A*B
$\begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}$

C = A@B
or
A.dot(B)
$\begin{bmatrix} 5 & 4 \\ 3 & 4 \end{bmatrix}$

lly we have
+= a*=3
*= b+=a

sum, min, max, cumulative sum
sqrt, add, exponent
cumulative sum ↓ a, a+b, a+b+c...

→ b.sum(axis=0)
→ b.min(axis=1)
→ b.cumsum(axis=1)
→ np.exp(B)
→ np.sqrt(A)
→ np.add(B, C) #adds elements by element

Indexing, Slicing & Iterating.

→ a[2:5] # Elements with index 2, 3 & 4 will be taken.

→ a[:6:2] = 1000

From start to 6th element set every 2nd element to 1000.

→ a[::-1] # reversed a

→ b[2,3] # m=2 & n=3 element is picked

→ b[-1] or b[-1:]

last row

→ consider a

array([[3., 7., 3., 4.],
[1., 4., 2., 2.],
[7., 2., 4., 9.]])

→ a.ravel()

flattened.

([3. . . . . . 9.])

→ a.reshape(6,2)

→ a.resize((2,6))

Stacking

→ a = np.floor(10*rg.random((2,2)))

→ np.vstack((a,b))

stacks a & b vertically

np.hstack((a,b))

horizontally.

→ column-stack

→ row-stack

Automatic Reshaping

b = a.reshape((2, -1, 3))

-1 means whatever is needed

Splitting

a = np.hsplit(b, 3)

c = np.vsplit(d, 3)

splits a matrix array to 3 different matrix array.

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

o/p

[array([[6., 7., 6., 9.],
...]),
array([[0., 5., 4., 0.]
array([[ ]])]

Sorting.

→ np.sort(arr)

ascending order sorted. arr is 1D array.

- argsort
- lexsort
- searchsorted
- partition.

different types of sorting.

Concatenate.

np.concatenate((a,b))

Unique items & Count

→ unique-ota = np.unique(a)

→ To get the indices ~~values~~ of the unique values.

```
→ unique_values, indices_list
  = np.unique(a, return_index
              = True)
print(indices_list)
```

→ Occurance count

```
unique_values, occurence_count
  = np.unique(a, return_counts
              = True)
```

Plotting arrays with Matplotlib.

Eg.

```
import matplotlib.pyplot
    as plt

>> plt.plot(a)
   plt.show()

>> plt.plot(x, y, 'purple')
        # line

>> plt.plot(x, y, 'o')
        # dots.
```

Eg:

```
fig = plt.figure()
ax = fig.add_subplot
     (projection = '3d')
X = np.arange(-5, 5, 0.15)
Y = np.arange(-5, 5, 0.15)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 +
            Y**2)
Z = np.sin(R)

ax.plot_surface(X, Y, Z,
    rstride=1, cstride=1,
    cmap='viridis')
```

Advanced Indexing

```
→ x = np.arange(10, 1, -1)
  x[np.array([3, 3, 1, 8])]
  x   # array([7, 7, 9, 2])
```

***

```
→ y = np.arange(35).
        reshape(5, 7)
  y[np.array([0, 2, 4]),
    np.array([0, 1, 2])]
  # (0,0) (2,1) (4,2) → elements
    are picked
```

np.newaxis → adjusts your matrix as per requirement.

→ 3D Array accessed.

```
a[0, :, 3].shape
    → row   → column
(3,)
a[0, :, 3, np.newaxis]
            .shape
(3, 1)
a[0, :, 3, np.newaxis,
      np.newaxis].shape
(3, 1, 1)
```

```
>> a > 14
[False False True
 True False
```

Replacing values after filtering.

```
P = np.arange(-10,10)
     .reshape(2, 2, 5)

P

q = P<0

q

P[q] = 0

P
```

so all values less than 0 is replaced with 0.

Create arrays with regularly spaced values

① np.arange(0, 10, 2)
#even nos from 0 to 10

② np.linspace(0.1, 0.2, num=5, endpoint=False, dtype=np.int)

0.1 to 0.18

③ geomspace & logspace
#same like linespace but on logscale or (geometric progression

④ meshgrid → used to create rectangular grid out of 2 1D arrays
mgrid
ogrid.
→ return open ndgrid

→ np.meshgrid(x, y) # one array must have more elements
→ np.mgrid[0:4, 0:6]
→ np.ogrid[0:4, 0:6]

Ufunc - Universal functions
→ functions that operate element by element on whole arrays.