Numpy → Homogeneous Multidimensional array.

Numpy dimensions → called as ~~axes~~ axes.

Numpy array class → ndarray(or) array

numpy.array is not same as array.array

ndarray.ndim → No of axis of the array.

ndarray.shape → size of array of each dim

shape → (n, m).

ndarray.size ⇒ equal to product of element of shape.

ndarray.dtype → describing type of elements in the array.

numpy.int32

numpy.int16

numpy.float64.

ndarray.itemsize ⇒ size in bytes of each element of the array.

↓ equivalent to ndarray.dtype.itemsize

ndarray.data → memory at that loc.

np.arrange(10) → prints from 0 to 10.

np.arrang(15).reshape(3,5)

print('a=\n{}'.format(a)) → [[0, 1 2 3 4
[5 6 7 8 9
10 11 12 13 14]]

print("a.shape = {}".format(a.shape)).

print("a.ndim = {}".format(a.ndim))

array transforms seq of seq into 2D arrays.

seq of seq of seq into 3D arrays.

b = [[(1, 2), (3, 4)]])
print ("b = \n { }".format (b))

↳ b = ~~[[1 2] [3 4]]~~ o/p → b = [[1 2]
[3 4]]

a = np.array ([(6,7), (5,7), (7,6)], dtype = complex)
print ("a = \n { }".format (a))

↓

o/p → a = [[6.+0.j 7.+0.j]
[5.+0.j 7.+0.j]
[7.+0.j 6.+0.j]]

fun → zeros → creates array full of zeroes.
funct → 1's → " " " 1's.
" → Empty → " " whose initial content is random and depends on state of memory.

↓

By default dtype created is float 64.

print ("np.zeros ((3,4)) = \n { }".format (np.zeros ((3,4))))

↓

np.zeros((3,4)) = [[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]]

print ("np.ones ((2,(3,4)) = \n { }".format (np.ones (2,(3,4))))

rows x columns

~~Prints for twice~~ prints as 2 lists of matrices

np.ones((2,3,4)) = [[[1. 1. 1. 1.]
[1. 1. 1. 1.]
[1. 1. 1. 1.]]

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

print ("np.empty((2,3)) = \n {}". format (np.empty ((2,3))).

↓

np.empty ((2,3)) = [[3.0e-323 3.5e-323 2.5e-323]
[3.5e-323 3.5e-323 3.0e-323]]

o/p is like this bcz it creates random numbers.

→ To create seq of numbers → numpy provides arange funct.

returns an array. ← analogues to range built-in funct.

print (np.arange ~~[10,35~~ [10,30,5] = {}". format (np.arange (10,30,5

↓

o/p → np.arange (10,30,5) = [10, 15, 20, 25]

starts at 10 with 5 diff until 30 it prints all numbers

It also works for decimals.

np. arrange (0, 2, 0.3)

start at 0 with a diff of 0.3 points

all numbers until number 2.

~~0, 0.3.~~

O/P → [0 0.3 0.6 0.9 1.2 1.5 1.8]

Due to finite floating point precision it is not possible to predict the no of elements obtained so we use the funct. linspace.

Print Arrays

One dim arrays are printed as rows.

2 dim " " " " matrices.

3 " " " " " lists of matrices.

If array is too large to be printed, Numpy skips the central part of array and only prints corners.

Basic Operations.

matrix product → @ is done using symbol.

```
c = np.array([[1,1], [0,1]])
d = np.array([[2,0], [2,1]])
print("c@d = \n{}".format(c@d))
```

↓

```
c@d = [[4 1]
       [2 1]]
```

Element wise Prod.

```
print("c*d = \n{}".format(c*d))
```

↓

```
c*d = [[2 0]
       [0,1]]
```

we can also do matrix product by using dot funct.

print ("c.dot(d) = \n {}". format (c.dot(d)))

↓

c.dot(d) = [[4 1]
[2 1]]

use of += and *=

rg = np.random.default_rng(1)

a = np.ones((2,3), dtype = int)

b = rg.random((2,3))

a*=3

print ("a = \n {}". format(a))

↓

a = [[3 3 3]
[3 3 3]]

Axes in Matrix

Axis 0 → moving through matrix from top to bottom

Axis 1 → moving through down from each row from top to bottom

Axis 2

↓

moving across columns from left to right

Universal functions

ufunc. → sin, cos, exp.

↓

operate element wise on an array → o/p → arr

Indexing, Slicing and Iterating

all these can be done on 1D arrays

a = np.arrange (10)**3.

O/P ⇒ $0^3, 1^3, 2^3$.

⇒ [0 1 8 27 64 125 216 343 512 729]

Multi dim arrays have one index per axis.

Shape Manipulation

↳ changing the shape of an array.

No. of elements on each axis

a = np.floor(10* np.random.random(3,4)))

a = [[9. 3. 2. 8.]
[1. 2. 9. 4.]
[7. 7. 2. 9.]]

a.ravel() → flattens the array

a.ravel() = [9. 3. 2. 8. 1. 2. 9. 4. 7. 7. 2. 9.]

a.reshape (6,2) → reshape the array.

[[9. 3.]
[2. 8.]
[1. 2.]
[9. 4.]
[7. 7.]
[2. 9.]]

Transpose

print ("a.T = \n {}".format (a.T))

a.T = [[9. 1. 7.]
[3. 2. 7.]
[2. 9. 2.]
[8. 4. 9.]]

a.T.shape ⇒ (4,3)
before it was (3,4) after transpose shape got changed to (4,3)

→ The order of element in array resulting from ravel is (C-style) → rightmost changes faster

→ ravel and reshape can also be instructed using optional arg to use (FORTRAN-style)
↓
leftmost changes fastest.

reshape()
↓
Returns its arg with modified shape

resize ()
↓
modifies array itself.

a = np.array([4., 2.])
b = np.array([3., 8.])
np.column_stack((a,b)) = [[4. 3.]
[2. 8.]]

np.hstack((a,b)) = [4. 2. 3. 8.]
a[:, newaxis] = ~~[4. 2.]~~ [[4.]
[2.]]

np.column_stack ((a[:, newaxis], b[:, newaxis]))

↓

[[4. 3.]
 [2. 8.]]

↓

o/p is same as o/p for

np.hstack ((a[:, newaxis],
b[:, newaxis]))

⇒ for arrays with more than 2D → hstack stacks along second axes.

→ vstack stacks along 1st axes.

r_ and c_ ⇒ useful for creating arrays by stacking Nos along one axis.

↓ use of range literals.

np.r_[1:4, 0, 4] ⇒ [1 2 3 0 4]

Copies and Views.

③ Cases → No copy at all.

↳ View or shallow copy.

↳ Deep copy.

1) No copy at all

↳ Simple assignments make no copy of array objects or of their data

↓

Just gives the o/p as True/False

→ Python makes mutable objs as ref; so funct. calls make no copy.

② View or Shallow Copy.

↓

This method creates a new array obj.
↳ looks at same data.

c.base is a ⇒ c is a view of data owned by a.
c.flags-owndata ⇒ c doesnot own the data.

c = c.reshape((2,6)) ⇒ a's shape will not change
c [0,4] = 1234 ⇒ a's shape changes.

Deep Copy.

↳ copy method makes a complete copy of array and its data.