Aug-2

16 ⟨ 2×8, 8×2, 4×4 ⟩ 1×16, 16×1

Python-tutorial 16

- reshape( ), extract diagonal elements :-

np.arange(6)
→ array([0,1,2,3,4,5]) → .shape = (6)

np.arange(16).reshape((2,8)) → (must match no. of elements)
→ array([[0,1,2,3,4,5,6,7], (also, 3-dim.)
[8,9,10,11,12,13,14,15]])

np.arange(16).reshape((2,2,3))
→ error (∵ 16 ≠ 2×2×3)

x = np.arange(16).reshape((-1,8)) ← understood default (automatically it will be 2)
→ array([[0,1,2,3,4,5,6,7],
[8,9,10,11,12,13,14,15]]) 2×8

np.arange(16).reshape((-2,5)) (default)
→ error (any -ve) (16/5) ×

np.arange(8).reshape((-1,-1,2))
→ error (∵ only one unknown dim. is allowed).

- extract diagonal :

np.diag(x) → array([0,9])
(∵ by-default one-sq. it chooses).

Note : by default key (k=0)

(16,) = 1x16 ← shape

```
x = np.arange(16).reshape((-1,4))
x
array([[0, 1, 2, 3],
       [4, 5, 6, 7],
       [8, 9,10,11],
       [12,13,14,15]])
np.diag(x) → array([0,5,10,15])
np.diag(x,k=2) → array([2,7])
np.diag(x,k=-1) → array([4,9,13])
```

Note: by-default key(k=0)

```
np.diag(np.diag(x))   (: reconstruct sq. matrix with diag. elem)
array([[0, 0, 0, 0],
       [0, 5, 0, 0],
       [0, 0,10, 0],
       [0, 0, 0,15]])
```

- diagflat:

```
np.diagflat([[1,2],[3,4]])   (k=0 default)
array([[1, 0, 0, 0],   (: first flattens the list then creates sq. matrix.)
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

- triangeral matrices:

sign = np.tri(rows, columns, key, dtype)

```
np.tri(3,5,1, dtype=int)
array([[1 1 0 0 0]
       [1 1 1 0 0]
       [1 1 1 1 0]])
```
3x5

key=1 triangle=1

- lower triangle:

```
np.tril([[1,2,3],[4,5,6],[7,8,9]], 1)
array([[1 2 0]
       [4 5 6]
       [7 8 9]])
```
key=1

- upper triangle:

```
np.triu([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], 1)
array([[0 2 3]
       [0 0 6]
       [0 0 0]
       [0 0 0]])
```

- Random Number Generation:

rand() → uniform distribution b/w [0,1)
randn() → standard Normal distri. (mean=0, std.d=1)
randint() → random data integer

(2 X 3 ke 2 sets)

- rand()

```
print(np.random.rand(2,2,3))
[[[0.1 0.8 0.4
   0.9 0.5 0.8]]
 [[0.4 0.6 0.9]
  [0.2 0.4 0.8]]
 [[0.9 0.4 0.2]
  [0.8 0.2 0.4]]]
```

- randn()

```
print(np.random.randn(4,3))
```

→ data-set of 4X3 with mean=0, std.d.=1.

- randint() (lower, upper, shape/how many)

```
print(np.randint(1,100,(2,2))
[[98 52]
 [6  18]]
```

- Sorting: (by-default = ascending)

```
A = randint(1,100,10) → 1D-array
np.sort(A)
M = randint(1,100,25).reshape(5,5)
or M = randint(1,100,(5,5))
```

row-sorting: np.sort(M, axis=0/1)

or column sorting

Note: row-sorting [0000], column []

- Maximum value of matrix:
  M.max()
- Indices of maximum values:
  M = [20 40 60
  12 100 90
  48 52 89].
  M.argmax(axis=1) → [2 1 2]
  (∵ try argmax with blank axis)

Note: In 3D, argmax first flatters the data & then gives index of maximum value.

- Indexing & slicing:
  also you can do in 3D (its easy)
  mat[0][:,1]

- Subseting:
  mat = np.array(randint(10,100,(3,5))
  mat > 50 → (True/False, matrix)
  mat[mat > 50] → value ~~matrix~~ ↓

- Universal functions:
  *Multiply mat1 * mat2 (∵ not actual matrix multiplication, its index-wise)
  ≠ mat1 @ mat2 (∵ actual multiply)

matrix multiplication : (m,n) * @ (n,p)
same

* → index-wise multiplication
@ → actual matrix-multiplication

- division by zero : $M = \begin{bmatrix} 3 & 2 \\ 5 & 1 \end{bmatrix}$
(possible in numpy, not in python)

M/0 → $\begin{bmatrix} inf & inf \\ inf & inf \end{bmatrix}$

- power:
M**3 or power(M,3)
- Broadcasting:

```
s = np.zeros((3,3))
s = s+100
s
array([[100., 100., 100.],
       [100., 100., 100.],
       [100., 100., 100.]])
```

element-wise broadcasting operation.

row-wise broadcasting

```
add_rows = np.array([1, 0, 2])
y = s+add_rows
y
array([[101., 100., 102.],
       [101., 100., 102.],
       [101., 100., 102.]])
add_cols = np.array([[0,1,2,3]])
add_cols = add_cols.T
y = s+add_cols
```

('transpose for making column')

- Square-root:

  np.sqrt(mat1)

- exponential:

  np.exp(mat1)

- Modulo reminder:

  np.fmod(mat1, mat2)

  or mat1 % mat2

(: remainder of matrix1 / matrix2)