In [1]:

```
import numpy as np
```

1.Array Operations

In [71]:

```
a1 = np.arange(12).reshape(3,4)
a2 = np.arange(12,24).reshape(3,4)
print(a1)
print(a2)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

a. Scalar operations

In [72]:

```
print(a1)
print(a2)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

Arithmetic

In [73]:

```
a1*2
```

Out[73]:

```
array([[ 0,  2,  4,  6],
       [ 8, 10, 12, 14],
       [16, 18, 20, 22]])
```

In [74]:

```
1 a1+10
```

Out[74]:

```
array([[10, 11, 12, 13],
       [14, 15, 16, 17],
       [18, 19, 20, 21]])
```

In [75]:

```
1 a1/3
```

Out[75]:

```
array([[0.        , 0.33333333, 0.66666667, 1.        ],
       [1.33333333, 1.66666667, 2.        , 2.33333333],
       [2.66666667, 3.        , 3.33333333, 3.66666667]])
```

In [76]:

```
1 a1-4
```

Out[76]:

```
array([[-4, -3, -2, -1],
       [ 0,  1,  2,  3],
       [ 4,  5,  6,  7]])
```

relational

In [77]:

```
1 a2 > 16
```

Out[77]:

```
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]])
```

In [78]:

```
1 a2 == 15
```

Out[78]:

```
array([[False, False, False,  True],
       [False, False, False, False],
       [False, False, False, False]])
```

In [79]:

```
a2 !=13
```

Out[79]:

```
array([[ True, False,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

b. Vector operations

In [80]:

```
print(a1)
print(a2)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

arithmetic

In [81]:

```
a1 + a2
# addition will be done item wise
```

Out[81]:

```
array([[12, 14, 16, 18],
       [20, 22, 24, 26],
       [28, 30, 32, 34]])
```

In [82]:

```
a1 * a2
# element wise multiplication
```

Out[82]:

```
array([[  0,  13,  28,  45],
       [ 64,  85, 108, 133],
       [160, 189, 220, 253]])
```

In [83]:

```
a1/a2
```

Out[83]:

```
array([[0.        , 0.07692308, 0.14285714, 0.2       ],
       [0.25      , 0.29411765, 0.33333333, 0.36842105],
       [0.4       , 0.42857143, 0.45454545, 0.47826087]])
```

In [84]:

```
a1-a2
```

Out[84]:

```
array([[-12, -12, -12, -12],
       [-12, -12, -12, -12],
       [-12, -12, -12, -12]])
```

relational

In [85]:

```
a1 > a2
```

Out[85]:

```
array([[False, False, False, False],
       [False, False, False, False],
       [False, False, False, False]])
```

In [86]:

```
a1 != a2
```

Out[86]:

```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

2.Array functions

In [87]:

```
a1 = np.random.random((3,3))
print(a1)
# this will give random number between the range 0 and 1
a1 = np.round(a1*100)
print(a1)
```

```
[[0.27999282 0.49836175 0.93546286]
 [0.30223843 0.17826006 0.44200551]
 [0.49061543 0.42641055 0.88666243]]
[[28. 50. 94.]
 [30. 18. 44.]
 [49. 43. 89.]]
```

max / min / sum / prod

- axis=0 it is column
- axis=1 it is row

In [88]:

```
np.max(a1)
```

Out[88]:

```
94.0
```

In [89]:

```
np.min(a1)
```

Out[89]:

```
18.0
```

In [90]:

```
# if we want minimum value of particular axis then
# if axis=0 then it will give column wise minimum value
np.min(a1, axis=0)
```

Out[90]:

```
array([28., 18., 44.])
```

In [91]:

```
np.min(a1, axis=1)
# here axis = 1 so it wili work on rows
```

Out[91]:

```
array([28., 18., 43.])
```

In [92]:

```
np.sum(a1)
```

Out[92]:

```
445.0
```

In [93]:

```
np.sum(a1, axis=1)
# row wise sum
```

Out[93]:

```
array([172.,  92., 181.])
```

In [94]:

```
np.prod(a1)
# product
```

Out[94]:

```
586349916768000.0
```

In [95]:

```
np.prod(a1, axis=0)
# product of column values
```

Out[95]:

```
array([ 41160.,  38700., 368104.])
```

In [96]:

```
np.prod(a1, axis=1)
# product of row values
```

Out[96]:

```
array([131600.,  23760., 187523.])
```

mean / median / std / var

In [97]:

```
np.mean(a1)
```

Out[97]:

```
49.44444444444444
```

In [98]:

```
np.mean(a1, axis=0)
# mean of each column
```

Out[98]:

```
array([35.66666667, 37.        , 75.66666667])
```

In [99]:

```
np.median(a1)
```

Out[99]:

```
44.0
```

In [100]:

```
np.std(a1)
```

Out[100]:

```
24.604024562891293
```

In [101]:

```
np.var(a1)
```

Out[101]:

```
605.3580246913581
```

trignometry

In [102]:

```
1 np.sin(a1)
```

Out[102]:

```
array([[ 0.27090579, -0.26237485, -0.24525199],
       [-0.98803162, -0.75098725,  0.01770193],
       [-0.95375265, -0.83177474,  0.86006941]])
```

dot product

In [68]:

```
1 a2 = np.arange(12).reshape(3,4)
2 a3 = np.arange(12,24).reshape(4,3)
3 print(a2)
4 print(a3)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[12 13 14]
 [15 16 17]
 [18 19 20]
 [21 22 23]]
```

In [103]:

```
1 np.dot(a2,a3)
```

Out[103]:

```
array([[ 906,  960, 1014],
       [1170, 1240, 1310],
       [1434, 1520, 1606]])
```

In [113]:

```
1 # other way for dot multiplication
2 a2.dot(a3)
```

Out[113]:

```
array([[ 906,  960, 1014],
       [1170, 1240, 1310],
       [1434, 1520, 1606]])
```

log and exponent

In [104]:

```
a1
```

Out[104]:

```
array([[28., 50., 94.],
       [30., 18., 44.],
       [49., 43., 89.]])
```

In [105]:

```
np.log(a1)
```

Out[105]:

```
array([[3.33220451, 3.91202301, 4.54329478],
       [3.40119738, 2.89037176, 3.78418963],
       [3.8918203 , 3.76120012, 4.48863637]])
```

In [107]:

```
np.exp(a1)
# exponent
```

Out[107]:

```
array([[1.44625706e+12, 5.18470553e+21, 6.66317622e+40],
       [1.06864746e+13, 6.56599691e+07, 1.28516001e+19],
       [1.90734657e+21, 4.72783947e+18, 4.48961282e+38]])
```

round / floor / ceil

In [110]:

```
a = np.random.random((3,4))
print(a)
# we will get random values between 0 to 1
print(a*100)
# we are multiplying it with 100
np.round(a*100)
# now we will round off the values
```

```
[[0.61087332 0.22066652 0.04624918 0.13387266]
 [0.12339297 0.65699782 0.29365854 0.31516081]
 [0.92205717 0.66792889 0.93118636 0.89666914]]
[[61.08733172 22.06665223  4.62491831 13.38726636]
 [12.33929698 65.69978205 29.36585435 31.51608135]
 [92.20571747 66.79288904 93.11863643 89.66691424]]
```

Out[110]:

```
array([[61., 22.,  5., 13.],
       [12., 66., 29., 32.],
       [92., 67., 93., 90.]])
```

In [111]:

```
a = np.random.random((3,4))
print(a)
# we will get random values between 0 to 1
print(a*100)
# we are multiplying it with 100
np.floor(a*100)
# now we will floor the values
```

```
[[0.52879632 0.40978892 0.32087545 0.25503579]
 [0.88779447 0.56780666 0.00938096 0.47992876]
 [0.50494063 0.25831264 0.27514974 0.18485574]]
[[52.87963183 40.97889164 32.08754508 25.50357944]
 [88.77944701 56.7806662   0.93809582 47.99287634]
 [50.49406311 25.83126449 27.51497441 18.48557355]]
```

Out[111]:

```
array([[52., 40., 32., 25.],
       [88., 56.,  0., 47.],
       [50., 25., 27., 18.]])
```

In [112]:

```
a = np.random.random((3,4))
print(a)
# we will get random values between 0 to 1
print(a*100)
# we are multiplying it with 100
np.ceil(a*100)
# now we will ceil the values
```

```
[[0.40554253 0.47244245 0.11320834 0.16445815]
 [0.17695094 0.50241432 0.30285826 0.63854692]
 [0.34501671 0.0776014  0.91667775 0.54337987]]
[[40.55425297 47.24424459 11.32083408 16.44581507]
 [17.69509381 50.24143222 30.28582642 63.85469238]
 [34.50167058  7.76014043 91.66777474 54.33798726]]
```

Out[112]:

```
array([[41., 48., 12., 17.],
       [18., 51., 31., 64.],
       [35.,  8., 92., 55.]])
```

- **Note :** after typing np. press tab key to get all available function within that np. syntax

3.Reshaping numpy arrays

1. Reshape
2. Transpose
3. Ravel

1. Reshape

```
np.reshape(a, newshape, order='C)
```

- Gives a new shape to an array without changing its data.

In [120]:

```
a1=np.arange(24)
a1
```

Out[120]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23])
```

In [124]:

```
np.reshape(a1,(8,3))
```

Out[124]:

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17],
       [18, 19, 20],
       [21, 22, 23]])
```

In [125]:

```
np.reshape(a1,(12,2))
```

Out[125]:

```
array([[ 0,  1],
       [ 2,  3],
       [ 4,  5],
       [ 6,  7],
       [ 8,  9],
       [10, 11],
       [12, 13],
       [14, 15],
       [16, 17],
       [18, 19],
       [20, 21],
       [22, 23]])
```

In [126]:

```
np.reshape(a1,(6,4))
```

Out[126]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [128]:

```
a = np.array([[1,2,3], [4,5,6]])
np.reshape(a, 6)
```

Out[128]:

```
array([1, 2, 3, 4, 5, 6])
```

In [129]:

```
np.reshape(a,(3,2))
```

Out[129]:

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In [119]:

```
b = np.arange(16).reshape(2,2,2,2)
b
# 4D array will be created
```

Out[119]:

```
array([[[[ 0,  1],
         [ 2,  3]],

        [[ 4,  5],
         [ 6,  7]]],


       [[[ 8,  9],
         [10, 11]],

        [[12, 13],
         [14, 15]]]])
```

In [131]:

```
np.reshape(b,(2,2,4))
# 3D array
```

Out[131]:

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
```

2. Transpose

> np.transpose(a, axes=None)

Returns an array with axes transposed.

> - ndarray.transpose is Equivalent method.

In [132]:

```
'''For a 1-D array, it returns an unchanged view of the original array,
as a transposed vector is simply the same vector
'''
a = np.array([1, 2, 3, 4])
print(a)
np.transpose(a)
```

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

Out[132]:

```
array([1, 2, 3, 4])
```

In [141]:

```
b = np.arange(8).reshape(2,4)
print(b)
```

```
[[0 1 2 3]
 [4 5 6 7]]
```

In [142]:

```
b.T
```

Out[142]:

```
array([[0, 4],
       [1, 5],
       [2, 6],
       [3, 7]])
```

In [147]:

```
1 b.transpose()
```

Out[147]:

```
array([[0, 4],
       [1, 5],
       [2, 6],
       [3, 7]])
```

In [145]:

```
1 np.transpose(b)
```

Out[145]:

```
array([[0, 4],
       [1, 5],
       [2, 6],
       [3, 7]])
```

3. Ravel

- The numpy.ravel() functions returns contiguous flattened array(1D array with all the input-array elements and with the same type as it). A copy is made only if needed.
- it will bring higehr dimension array to 1D without losing any information

- **Syntax :**
  - np.ravel(array, order = 'C')

In [148]:

```
1 b = np.arange(16).reshape(2,2,2,2)
2 b
3 # 4D array will be created
```

Out[148]:

```
array([[[[ 0,  1],
         [ 2,  3]],

        [[ 4,  5],
         [ 6,  7]]],


       [[[ 8,  9],
         [10, 11]],

        [[12, 13],
         [14, 15]]]])
```

In [153]:

```
print(np.ravel(b))
# it will return 1D array
print(np.ravel(b).ndim)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
1
```

In [154]:

```
b.ravel()
# other way of writing ravel function
```

Out[154]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
```

4.Stacking

1. horizontal stacking
2. vertical stacking

- stack() is used for joining multiple NumPy arrays.
- Unlike, concatenate(), it joins arrays along a new axis.
- If axis is not explicitly passed, it is taken as 0.
- It returns a NumPy array.
- to join 2 arrays, they must have the same shape and dimensions.

- stack() creates a new array which has 1 more dimension than the input arrays. If we stack two 1-D arrays, the resultant array will have 2 dimensions.

In [4]:

```
a4 = np.arange(12).reshape(3,4)
a5 = np.arange(12,24).reshape(3,4)

print(a4)
print(a5)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

1. horizontal stacking

In [5]:

```
1 np.hstack((a4,a5))
```

Out[5]:

```
array([[ 0,  1,  2,  3, 12, 13, 14, 15],
       [ 4,  5,  6,  7, 16, 17, 18, 19],
       [ 8,  9, 10, 11, 20, 21, 22, 23]])
```

In [6]:

```
1 np.hstack((a4,a5,a4))
2 # we can join as many array as we want
```

Out[6]:

```
array([[ 0,  1,  2,  3, 12, 13, 14, 15,  0,  1,  2,  3],
       [ 4,  5,  6,  7, 16, 17, 18, 19,  4,  5,  6,  7],
       [ 8,  9, 10, 11, 20, 21, 22, 23,  8,  9, 10, 11]])
```

2.vertical stacking

In [7]:

```
1 np.vstack((a4,a5))
```

Out[7]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [9]:

```
1 np.vstack((a4,a5,a5))
```

Out[9]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [2]:

```
# input array
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

print(a)
print(b)

print(a.ndim)
print(b.ndim)

# Stacking two 1-d arrays
c = np.stack((a, b),axis=0)
print(c)
c.ndim
```

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

Out[2]:

```
2
```

In [3]:

```
# stack two 1-d arrays column - wise
np.stack((a,b),axis=1)
```

Out[3]:

```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

In [10]:

```
# input arrays
x=np.array([[1,2,3],
            [4,5,6]])

y=np.array([[7,8,9],
            [10,11,12]])
print(x)
print(x.ndim)
print(y)
print(y.ndim)
```

```
[[1 2 3]
 [4 5 6]]
2
[[ 7  8  9]
 [10 11 12]]
2
```

In [11]:

```
# stacking with axis=0

a=np.stack((x,y),axis=0)
print(a)
print(a.ndim)
```

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

 [[ 7  8  9]
  [10 11 12]]]
3
```

In [14]:

```
# stacking with axis=1

b=np.stack((x,y),axis=1)
print(b)
print(b.ndim)
```

```
[[[ 1  2  3]
  [ 7  8  9]]

 [[ 4  5  6]
  [10 11 12]]]
3
```

In [12]:

```
'''NumPy provides a helper function: dstack()
to stack along height, which is the same as depth.'''

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

arr = np.dstack((arr1, arr2))
print(arr)
```

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

5. Splitting

- Splitting is reverse operation of Joining.
- Joining merges multiple arrays into one and Splitting breaks one array into multiple.

1. hsplit()

In [18]:

```
a4 = np.arange(12).reshape(3,4)
print(a4)

```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

In [19]:

```
np.hsplit(a4,2)
```

Out[19]:

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

2.vsplit()

In [21]:

```
a5 = np.arange(12,24).reshape(3,4)
print(a5)
```

```
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

In [22]:

```
np.vsplit(a5,3)
```

Out[22]:

```
[array([[12, 13, 14, 15]]),
 array([[16, 17, 18, 19]]),
 array([[20, 21, 22, 23]])]
```

In [26]:

```
'''Use the hsplit() method to split the 2-D array
into three 2-D arrays along rows.'''

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9],
                [10, 11, 12], [13, 14, 15], [16, 17, 18]])

newarr = np.hsplit(arr, 3)
print(newarr)
```

```
[array([[ 1],
       [ 4],
       [ 7],
       [10],
       [13],
       [16]]), array([[ 2],
       [ 5],
       [ 8],
       [11],
       [14],
       [17]]), array([[ 3],
       [ 6],
       [ 9],
       [12],
       [15],
       [18]])]
```

In [27]:

```
print(newarr[0])
print(newarr[1])
print(newarr[2])
```

```
[[ 1]
 [ 4]
 [ 7]
 [10]
 [13]
 [16]]
[[ 2]
 [ 5]
 [ 8]
 [11]
 [14]
 [17]]
[[ 3]
 [ 6]
 [ 9]
 [12]
 [15]
 [18]]
```

In [28]:

```
'''Use the vsplit() method to split the 2-D array
into three 2-D arrays along columns.'''

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9],
                [10, 11, 12], [13, 14, 15], [16, 17, 18]])

newarr = np.vsplit(arr, 3)
print(newarr)
```

```
[array([[1, 2, 3],
       [4, 5, 6]]), array([[ 7,  8,  9],
       [10, 11, 12]]), array([[13, 14, 15],
       [16, 17, 18]])]
```

In [29]:

```
print(newarr[0])
print(newarr[1])
print(newarr[2])
```

```
[[1 2 3]
 [4 5 6]]
[[ 7  8  9]
 [10 11 12]]
[[13 14 15]
 [16 17 18]]
```

In [30]:

```
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 3)
print(newarr)
#  The return value is an array containing three arrays.
```

```
[array([1, 2]), array([3, 4]), array([5, 6])]
```

- The return value of the array_split() method is an array containing each of the split as an array.
- If you split an array into 3 arrays, you can access them from the result just like any array element:

In [32]:

```
arr = np.array([1, 2, 3, 4, 5, 6])

newarr = np.array_split(arr, 3)
print(newarr)

print(newarr[0])
print(newarr[1])
print(newarr[2])
```

```
[array([1, 2]), array([3, 4]), array([5, 6])]
[1 2]
[3 4]
[5 6]
```

In [33]:

```
# If the array has less elements than required, it will adjust from the end accordin
arr = np.array([1, 2, 3, 4, 5, 6])

newarr = np.array_split(arr, 4)

print(newarr)
```

```
[array([1, 2]), array([3, 4]), array([5]), array([6])]
```

In [34]:

```
# Split the 2-D array into three 2-D arrays.

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

newarr = np.array_split(arr, 3)

print(newarr)
```

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

In [35]:

```
print(newarr[0])
print(newarr[1])
print(newarr[2])
```

```
[[1 2]
 [3 4]]
[[5 6]
 [7 8]]
[[ 9 10]
 [11 12]]
```

- In addition, you can specify which axis you want to do the split around.
- The example below also returns three 2-D arrays, but they are split along the row (axis=1).

In [38]:

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17

newarr = np.array_split(arr, 3, axis=1)

print(newarr)
```

```
[array([[ 1],
       [ 4],
       [ 7],
       [10],
       [13],
       [16]]), array([[ 2],
       [ 5],
       [ 8],
       [11],
       [14],
       [17]]), array([[ 3],
       [ 6],
       [ 9],
       [12],
       [15],
       [18]])]
```

In [39]:

```
print(newarr[0])
print(newarr[1])
print(newarr[2])
```

```
[[ 1]
 [ 4]
 [ 7]
 [10]
 [13]
 [16]]
[[ 2]
 [ 5]
 [ 8]
 [11]
 [14]
 [17]]
[[ 3]
 [ 6]
 [ 9]
 [12]
 [15]
 [18]]
```

In [40]:

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17

newarr = np.array_split(arr, 3, axis=0)

print(newarr)
```

```
[array([[1, 2, 3],
       [4, 5, 6]]), array([[ 7,  8,  9],
       [10, 11, 12]]), array([[13, 14, 15],
       [16, 17, 18]])]
```