9.3 Arrays -Creating a NumPy ndarray Object, 1D, 2D and 3D Arrays (https://www.w3schools.com/python/numpy/numpy_creating_arrays.asp)

DO NOT refer higher dimensional arrays (greater than 3D) from this link

---

9.4 Array

- Indexing (https://www.w3schools.com/python/numpy/numpy_array_indexing.asp)
- Slicing (https://www.w3schools.com/python/numpy/numpy_array_slicing.asp)
- Shape (https://www.w3schools.com/python/numpy/numpy_array_shape.asp)
- Reshaping (https://www.w3schools.com/python/numpy/numpy_array_reshape.asp)
- Iteration (https://www.w3schools.com/python/numpy/numpy_array_iterating.asp)

9.5 Built-in functions

- Concatenate (https://www.w3schools.com/python/numpy/numpy_array_join.asp) refer only concatenate function from this link
- array_split (https://www.w3schools.com/python/numpy/numpy_array_split.asp) refer only array_split function from this link
- where (https://www.w3schools.com/python/numpy/numpy_array_search.asp) refer only where function from this link
- sort (https://www.w3schools.com/python/numpy/numpy_array_sort.asp) refer only sort function from this link

¶

In [ ]:
```
1
```

Numpy_VHA

February 11, 2024

1 Numpy

2 What is numpy?

- NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an 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 and much more.
- At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types

3 Numpy Arrays Vs Python Sequences

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
- A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays.

[56]:
```
%timeit sum(range(100000))
```

```
3.21 ms ± 86.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[57]:
```
%timeit np.sum(np.arange(100000))
```

```
133 µs ± 1.26 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

[182]:
```
# speed
# list
a = [i for i in range(10000000)]
```

```
b = [i for i in range(10000000,20000000)]

c = []
import time

start = time.time()
for i in range(len(a)):
  c.append(a[i] + b[i])
print(time.time()-start)
```

3.823643684387207

[183]:
```
# numpy
import numpy as np
a = np.arange(10000000)
b = np.arange(10000000,20000000)

start = time.time()
c = a + b
print(time.time()-start)
```

0.1406252384185791

[180]:
```
# memory
a = [i for i in range(10000000)]
import sys

sys.getsizeof(a)
```

[184]: 81528048

[181]:
```
a = np.arange(10000000,dtype=np.int8)
sys.getsizeof(a)
```

[181]: 10000096

4 Creating Numpy Arrays

- np.array
- np.array with dtype
- np.arange
- with reshape
- np.ones and np.zeros
- np.random
- np.linspace
- np.identity

[4]:
```
# np.array
import numpy as np

a = np.array([1,2,3])
print(a)
print(type(a))
```

```
[1 2 3]
<class 'numpy.ndarray'>
```

[3]:
```
a = np.array((1,2,3))
print(a)
```

```
[1 2 3]
```

[6]:
```
# 2D and 3D
b = np.array([[1,2,3],[4,5,6]])
print(b)
print()
c = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
print(c)
```

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

[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
```

[7]:
```
# dtype
np.array([1,2,3],dtype=float)
```

[7]:
```
array([1., 2., 3.])
```

[8]:
```
np.array([1,2,3],dtype=str)
```

[8]:
```
array(['1', '2', '3'], dtype='<U1')
```

[10]:
```
np.array([1,2,3],dtype=tuple)
```

[10]:
```
array([1, 2, 3], dtype=object)
```

[11]:
```
np.array([1,2,0],dtype=bool)
```

[11]:
```
array([ True,  True, False])
```

[12]:
```
np.array([1,2,0],dtype=complex)
```

[12]:
```
array([1.+0.j, 2.+0.j, 0.+0.j])
```

[13]:
```
# np.arange
np.arange(1,11,1)
```

[13]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

[14]:
```
np.arange(1,11,2)
```

[14]:
```
array([1, 3, 5, 7, 9])
```

[15]:
```
np.arange(11,1,-1)
```

[16]:
```
array([11, 10,  9,  8,  7,  6,  5,  4,  3,  2])
```

[17]:
```
# with reshape
np.arange(1,11).reshape(5,2)
```

[17]:
```
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10]])
```

[18]:
```
np.arange(1,11).reshape(2,5)
```

[18]:
```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
```

[19]:
```
np.arange(1,13).reshape(5,2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-19-737dfc36139a> in <module>
----> 1 np.arange(1,13).reshape(5,2)

ValueError: cannot reshape array of size 12 into shape (5,2)
```

[20]:
```
np.arange(1,13).reshape(6,2)
```

[20]:
```
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10],
       [11, 12]])
```

[21]:
```
np.arange(1,17).reshape(2,2,2,2)
```

[21]:
```
array([[[[ 1,  2],
         [ 3,  4]],

        [[ 5,  6],
         [ 7,  8]]],


       [[[ 9, 10],
         [11, 12]],

        [[13, 14],
         [15, 16]]]])
```

[37]:
```
np.array([1,4,7,8,9,4,5,2,1,0,3,13]).reshape(6,2)
```

[37]:
```
array([[ 1,  4],
       [ 7,  8],
       [ 9,  4],
       [ 5,  2],
       [ 1,  0],
       [ 3, 13]])
```

[22]:
```
# np.ones and np.zeros
np.ones((3,4))
```

[22]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

[23]:
```
np.zeros((3,4))
```

[23]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

[24]:
```
np.ones((3,4),dtype=int)
```

[24]:
```
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
```

[25]:
```
# np.random
np.random.random((3,4))
```

[25]:
```
array([[7.66790583e-01, 2.78562999e-01, 2.17458415e-01, 5.15394925e-01],
       [1.46550894e-01, 2.49777909e-01, 5.39589866e-01, 1.27481946e-01],
```

```
       [4.82561439e-01, 2.14434094e-01, 5.39000369e-05, 8.57024103e-01]])
```

[28]:
```
np.random.randint(10,size=(5,2))
```

[28]:
```
array([[6, 5],
       [4, 4],
       [9, 9],
       [7, 9],
       [7, 1]])
```

[30]:
```
np.random.randint(20,size=(5,2,2))
```

[30]:
```
array([[[ 8,  7],
        [12, 13]],

       [[12, 12],
        [14, 11]],

       [[13, 12],
        [ 4, 18]],

       [[ 4, 17],
        [ 5, 14]],

       [[ 8,  3],
        [ 6, 16]]])
```

[32]:
```
# np.linspace
np.linspace(-10,10,10,dtype=int)
```

[32]:
```
array([-10,  -7,  -5,  -3,  -1,   1,   3,   5,   7,  10])
```

[33]:
```
np.linspace(-100,100,10,dtype=int)
```

[33]:
```
array([-100,  -77,  -55,  -33,  -11,   11,   33,   55,   77,  100])
```

[34]:
```
# np.identity
np.identity(3)
```

[34]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

5 Array Attributes

- ndim
- shape
- size

- itemsize
- dtype

[39]:
```
a1 = np.arange(10,dtype=np.int32)
a2 = np.arange(12,dtype=float).reshape(3,4)
a3 = np.arange(8).reshape(2,2,2)
print(a1)
print()
print(a2)
print()
print(a3)
```

```
[0 1 2 3 4 5 6 7 8 9]

[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]

[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

[41]:
```
# ndim
print(a1.ndim)
print(a2.ndim)
print(a3.ndim)
```

```
1
2
3
```

[42]:
```
# shape
print(a1.shape)
print(a1)
print()
print(a2.shape)
print(a2)
print()
print(a3.shape)
print(a3)
```

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

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

```
 [ 8.  9. 10. 11.]]

(2, 2, 2)
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

[43]:
```
# size
print(a1.size)
print(a1)
print()
print(a2.size)
print(a2)
print()
print(a3.size)
print(a3)
```

```
10
[0 1 2 3 4 5 6 7 8 9]

12
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]

8
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

[48]:
```
# itemsize
print(a1.itemsize)
print(a1)
print()
print(a2.itemsize)
print(a2)
print()
print(a3.itemsize)
print(a3)
```

```
4
[0 1 2 3 4 5 6 7 8 9]

8
[[ 0.  1.  2.  3.]
```

```
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]

4
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
4
['1']
```

[49]:
```
# dtype
print(a1.dtype)
print(a2.dtype)
print(a3.dtype)
```

```
int32
float64
int32
```

6 Changing Datatype

[53]:
```
# astype
a3=a3.astype(np.int16)
```

[54]:
```
a3.dtype
```

[54]:
```
dtype('int16')
```

7 Array Operations

- reshape
- scalar operations
- relational
- vector operations

[59]:
```
a1 = np.arange(12).reshape(3,4)
a2 = np.arange(12,24).reshape(3,4)
print(a1)
print()
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]]
```

[63]:
```
# scalar operations

# arithmetic
print("addition",a1+2)
print()
print("subtraction",a1-2)
print()
print("multipication",a1 * 2)
print()
print("divison",a1/2)
print()
print("power",a1 ** 2)
print()
print("f-divison",a1//2)
```

```
addition [[ 2  3  4  5]
 [ 6  7  8  9]
 [10 11 12 13]]

subtraction [[-2 -1  0  1]
 [ 2  3  4  5]
 [ 6  7  8  9]]

multipication [[ 0  2  4  6]
 [ 8 10 12 14]
 [16 18 20 22]]

divison [[0.  0.5 1.  1.5]
 [2.  2.5 3.  3.5]
 [4.  4.5 5.  5.5]]

power [[  0   1   4   9]
 [ 16  25  36  49]
 [ 64  81 100 121]]

f-divison [[0 0 1 1]
 [2 2 3 3]
 [4 4 5 5]]
```

[64]:
```
# relational
a2 == 15
```

[64]:
```
array([[False, False, False,  True],
       [False, False, False, False],
       [False, False, False, False]])
```

[65]:
```
a2>5
```

[65]:
```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

[66]:
```
a2>17
```

[66]:
```
array([[False, False, False, False],
       [False, False,  True,  True],
       [ True,  True,  True,  True]])
```

[67]:
```
a1>a2
```

[67]:
```
array([[False, False, False, False],
       [False, False, False, False],
       [False, False, False, False]])
```

[68]:
```
a1<a2
```

[68]:
```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

[69]:
```
# vector operations
# arithmetic
print("addition",a1+a2)
print()
print("subtraction",a1-a2)
print()
print("multipication",a1 * a2)
print()
print("divison",a1/a2)
print()
print("power",a1 ** a2)
print()
print("f-divison",a1//a2)
```

```
addition [[12 14 16 18]
 [20 22 24 26]
 [28 30 32 34]]

subtraction [[-12 -12 -12 -12]
 [-12 -12 -12 -12]
 [-12 -12 -12 -12]]

multipication [[  0  13  28  45]
 [ 64  85 108 133]
```

```
 [160 189 220 253]]

divison [[0.         0.07692308 0.14285714 0.2       ]
 [0.25       0.29411765 0.33333333 0.36842105]
 [0.4        0.42857143 0.45454545 0.47826087]]

power [[          0           1       16384    14348907]
 [          0 -1564725563  1159987200   442181591]
 [          0  1914644777 -1304428544  -122979837]]

f-divison [[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```


8 Array Functions

- max/ min/sum/ prod
- mean/ median/ std/ var/sort
- trigonomoetric functions
- dot product
- log and exponents
- round/floor/ceil

[89]:
```
a1 = np.random.random((3,3))
print(a1)
```

```
[[0.13943692 0.24849347 0.09540113]
 [0.11492873 0.50514922 0.57084673]
 [0.32226955 0.61744354 0.85962115]]
```

[90]:
```
a1 = np.round(a1*100)
print(a1)
```

```
[[14. 25. 10.]
 [11. 51. 57.]
 [32. 62. 86.]]
```

[91]:
```
# max
# 0 -> col and 1 -> row
np.max(a1,axis=0)
```

[91]:
```
array([32., 62., 86.])
```

[92]:
```
np.max(a1,axis=1)
```

[92]:
```
array([25., 57., 86.])
```

[93]:
```
np.max(a1)
```

[93]: 86.0

[94]:
```
#min
np.min(a1,axis=0)
```

[94]: array([11., 25., 10.])

[95]:
```
np.min(a1,axis=1)
```

[95]: array([10., 11., 32.])

[96]:
```
np.min(a1)
```

[96]: 10.0

[97]:
```
#sum
np.sum(a1,axis=0)
```

[97]: array([57., 138., 153.])

[98]:
```
np.sum(a1,axis=1)
```

[98]: array([49., 119., 180.])

[99]:
```
np.sum(a1)
```

[99]: 348.0

[100]:
```
np.prod(a1)
```

[100]: 19096152768000.0

[101]:
```
#prod
np.prod(a1,axis=0)
```

[101]: array([4928., 79050., 49020.])

[79]:
```
np.prod(a1,axis=1)
```

[79]: array([30912., 10472., 116928.])

9 np.sort

- Return a sorted copy of an array.

[4]:
```
# code
import numpy as np
a = np.array([58, 69, 15, 43, 66, 72, 88, 44, 84, 68, 93, 77, 18, 89, 93])
```

```
a
```

[4]:
```
array([58, 69, 15, 43, 66, 72, 88, 44, 84, 68, 93, 77, 18, 89, 93])
```

[71]:
```
np.sort(a)[::-1]
```

[71]:
```
array([93, 93, 89, 88, 84, 77, 72, 69, 68, 66, 58, 44, 43, 18, 15])
```

[72]:
```
b = np.random.randint(1,100,24).reshape(6,4)
b
```

[72]:
```
array([[82, 39, 65, 97],
       [ 4, 48, 81, 39],
       [38, 26, 47, 94],
       [ 6, 42,  9, 45],
       [52, 75, 79, 94],
       [53, 68, 60, 67]])
```

[73]:
```
np.sort(b,axis=0)
```

[73]:
```
array([[ 4, 26,  9, 39],
       [ 6, 39, 47, 45],
       [38, 42, 60, 67],
       [52, 48, 65, 94],
       [53, 68, 79, 94],
       [82, 75, 81, 97]])
```

[74]:
```
np.sort(b,axis=1)
```

[74]:
```
array([[39, 65, 82, 97],
       [ 4, 39, 48, 81],
       [26, 38, 47, 94],
       [ 6,  9, 42, 45],
       [52, 75, 79, 94],
       [53, 60, 67, 68]])
```


10 np.append

- The numpy.append() appends values along the mentioned axis at the end of the array

[78]:
```
print(a)
print()
print(np.append(a,200))
```

```
[58 69 15 43 66 72 88 44 84 68 93 77 18 89 93]

[ 58  69  15  43  66  72  88  44  84  68  93  77  18  89  93 200]
```

[80]:
```
print(b)
print()
print(np.append(b,np.ones((b.shape[0],1)),axis=1))
```

```
[[82 39 65 97]
 [ 4 48 81 39]
 [38 26 47 94]
 [ 6 42  9 45]
 [52 75 79 94]
 [53 68 60 67]]

[[82. 39. 65. 97.  1.]
 [ 4. 48. 81. 39.  1.]
 [38. 26. 47. 94.  1.]
 [ 6. 42.  9. 45.  1.]
 [52. 75. 79. 94.  1.]
 [53. 68. 60. 67.  1.]]
```


[103]:
```
print(b)
print()
print(np.append(b,np.random.random((b.shape[0],1)),axis=1))
```

```
[[82 39 65 97]
 [ 4 48 81 39]
 [38 26 47 94]
 [ 6 42  9 45]
 [52 75 79 94]
 [53 68 60 67]]

[[82.         39.         65.         97.         0.89319894]
 [ 4.         48.         81.         39.         0.36925185]
 [38.         26.         47.         94.         0.43374653]
 [ 6.         42.          9.         45.         0.60719884]
 [52.         75.         79.         94.         0.22009929]
 [53.         68.         60.         67.         0.13865466]]
```

[104]:
```
b.shape[0]
```

[104]: 6

11 np.concatenate

- numpy.concatenate() function concatenate a sequence of arrays along an existing axis.

[105]:
```
# code
c = np.arange(6).reshape(2,3)
d = np.arange(6,12).reshape(2,3)
```

```
print(c)
print(d)
```

```
[[0 1 2]
 [3 4 5]]
[[ 6  7  8]
 [ 9 10 11]]
```

[106]:
```
np.concatenate((c,d),axis=0)
```

[106]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

[107]:
```
np.concatenate((c,d),axis=1)
```

[107]:
```
array([[ 0,  1,  2,  6,  7,  8],
       [ 3,  4,  5,  9, 10, 11]])
```

[108]:
```
e=d = np.arange(13,19).reshape(2,3)
```

[110]:
```
np.concatenate((c,d,e),axis=1)
```

[110]:
```
array([[ 0,  1,  2, 13, 14, 15, 13, 14, 15],
       [ 3,  4,  5, 16, 17, 18, 16, 17, 18]])
```

[111]:
```
np.concatenate((c,d,e),axis=0)
```

[111]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [13, 14, 15],
       [16, 17, 18],
       [13, 14, 15],
       [16, 17, 18]])
```

[4]:
```
import numpy as np
a=np.arange(12).reshape(3,4)
b=np.arange(13,19).reshape(3,2)
np.concatenate((a,b),axis=0)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_18816\1903385360.py in <module>
      2 a=np.arange(12).reshape(3,4)
      3 b=np.arange(13,19).reshape(3,2)
----> 4 np.concatenate((a,b),axis=0)
```

```
<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input array dimensions for the concatenation axis must matc
  ↪exactly, but along dimension 1, the array at index 0 has size 4 and the array
  ↪at index 1 has size 2
```

[5]:
```
import numpy as np
a=np.arange(12).reshape(3,4)
b=np.arange(13,19).reshape(3,2)
np.concatenate((a,b),axis=1)
```

[5]:
```
array([[ 0,  1,  2,  3, 13, 14],
       [ 4,  5,  6,  7, 15, 16],
       [ 8,  9, 10, 11, 17, 18]])
```

12 np.unique

- With the help of np.unique() method, we can get the unique values from an array given as parameter in np.unique() method.

[112]:
```
e = np.array([1,1,2,2,3,3,4,4,5,5,6,6])
np.unique(e)
```

[112]:
```
array([1, 2, 3, 4, 5, 6])
```

13 np.expand_dims

With the help of Numpy.expand_dims() method, we can get the expanded dimensions of an array

[114]:
```
a
```

[114]:
```
array([58, 69, 15, 43, 66, 72, 88, 44, 84, 68, 93, 77, 18, 89, 93])
```

[113]:
```
a.shape
```

[113]:
```
(15,)
```

[115]:
```
np.expand_dims(a,axis=0)
```

[115]:
```
array([[58, 69, 15, 43, 66, 72, 88, 44, 84, 68, 93, 77, 18, 89, 93]])
```

[116]:
```
np.expand_dims(a,axis=0).shape
```

[116]:
```
(1, 15)
```

[117]:
```
np.expand_dims(a,axis=1)
```

[117]:
```
array([[58],
       [69],
       [15],
       [43],
       [66],
       [72],
       [88],
       [44],
       [84],
       [68],
       [93],
       [77],
       [18],
       [89],
       [93]])
```

[118]:
```
np.expand_dims(a,axis=1).shape
```

[118]:
```
(15, 1)
```


14 np.where

- The numpy.where() function returns the indices of elements in an input array where the given condition is satisfied.-

[120]:
```
print(a)
```

```
[58 69 15 43 66 72 88 44 84 68 93 77 18 89 93]
```

[121]:
```
# find all indices with value greater than 50
np.where(a>50)
```

[121]:
```
(array([ 0,  1,  4,  5,  6,  8,  9, 10, 11, 13, 14], dtype=int64),)
```

[122]:
```
# replace all values > 50 with 0#whwere(condition,true,false)
np.where(a>50,0,a)
```

[122]:
```
array([ 0,  0, 15, 43,  0,  0,  0, 44,  0,  0,  0,  0, 18,  0,  0])
```

[123]:
```
np.where(a%2 == 0,0,a)
```

[123]:
```
array([ 0, 69, 15, 43,  0,  0,  0,  0,  0,  0, 93, 77,  0, 89, 93])
```

[8]:
```
a=np.random.randint(1,100,24).reshape(6,4)
a
```

[8]:
```
array([[36, 61, 24, 43],
       [46, 83, 48, 64],
```

```
       [ 9, 69, 45, 49],
       [39, 76, 56, 77],
       [72, 93, 57, 15],
       [23, 29, 77, 37]])
```

[12]:
```
print(np.where(a%2==0))
```

```
(array([0, 0, 1, 1, 1, 3, 3, 4], dtype=int64), array([0, 2, 0, 2, 3, 1, 2, 0],
dtype=int64))
```

[14]:
```
print(np.where(a%2==0,a,0))
```

```
[[36  0 24  0]
 [46  0 48 64]
 [ 0  0  0  0]
 [ 0 76 56  0]
 [72  0  0  0]
 [ 0  0  0  0]]
```

15 np.argmax

- The numpy.argmax() function returns indices of the max element of the array in a particular axis.

[124]:
```
# code
a
```

[124]: array([58, 69, 15, 43, 66, 72, 88, 44, 84, 68, 93, 77, 18, 89, 93])

[125]:
```
np.argmax(a)
```

[125]: 10

[134]:
```
np.argmin(a)
```

[134]: 2

[132]:
```
h=np.array([[1,2,3],[4,7,6]])
```

[128]:
```
np.argmax(h)
```

[128]: 5

[133]:
```
np.argmax(h,axis=1)
```

[133]: array([2, 1], dtype=int64)

[135]:
```
np.argmin(h,axis=1)
```

[135]:
```
array([0, 0], dtype=int64)
```

16 np.cumsum

- numpy.cumsum() function is used when we want to compute the cumulative sum of array elements over a given axis.

[136]:
```
a
```

[136]:
```
array([58, 69, 15, 43, 66, 72, 88, 44, 84, 68, 93, 77, 18, 89, 93])
```

[137]:
```
np.cumsum(a)
```

[137]:
```
array([ 58, 127, 142, 185, 251, 323, 411, 455, 539, 607, 700, 777, 795,
       884, 977], dtype=int32)
```

[138]:
```
b
```

[138]:
```
array([[82, 39, 65, 97],
       [ 4, 48, 81, 39],
       [38, 26, 47, 94],
       [ 6, 42,  9, 45],
       [52, 75, 79, 94],
       [53, 68, 60, 67]])
```

[139]:
```
np.cumsum(b,axis=1)
```

[139]:
```
array([[ 82, 121, 186, 283],
       [  4,  52, 133, 172],
       [ 38,  64, 111, 205],
       [  6,  48,  57, 102],
       [ 52, 127, 206, 300],
       [ 53, 121, 181, 248]], dtype=int32)
```

[140]:
```
np.cumsum(b,axis=0)
```

[140]:
```
array([[ 82,  39,  65,  97],
       [ 86,  87, 146, 136],
       [124, 113, 193, 230],
       [130, 155, 202, 275],
       [182, 230, 281, 369],
       [235, 298, 341, 436]], dtype=int32)
```

[141]:
```
np.cumsum(b)
```

[141]:
```
array([  82,  121,  186,  283,  287,  335,  416,  455,  493,  519,  566,
        660,  666,  708,  717,  762,  814,  889,  968, 1062, 1115, 1183,
       1243, 1310], dtype=int32)
```

[148]:
```
# np.cumprod
print(a)
np.cumprod(a,dtype="int64")
```

```
[58 69 15 43 66 72 88 44 84 68 93 77 18 89 93]
```

[148]:
```
array([                  58,                 4002,                60030,
                    2581290,            170365140,          12266290080,
                1079433527040,       47495075189760,     3989586315939840,
              271291869483909120,  6783399788293996544,  5812949634770288640,
             -6047371016392114176, -3260442321321164800, -8073230703515500544],
            dtype=int64)
```

[145]:
```
print(b)
np.cumprod(b,axis=0)
```

```
[[82 39 65 97]
 [ 4 48 81 39]
 [38 26 47 94]
 [ 6 42  9 45]
 [52 75 79 94]
 [53 68 60 67]]
```

[145]:
```
array([[        82,         39,         65,         97],
       [       328,       1872,       5265,       3783],
       [     12464,      48672,     247455,     355602],
       [     74784,    2044224,    2227095,   16002090],
       [   3888768,  153316800,  175940505, 1504196460],
       [ 206104704, 1835607808, 1966495708, 1996915012]], dtype=int32)
```

[146]:
```
print(b)
np.cumprod(b,axis=1)
```

```
[[82 39 65 97]
 [ 4 48 81 39]
 [38 26 47 94]
 [ 6 42  9 45]
 [52 75 79 94]
 [53 68 60 67]]
```

[146]:
```
array([[      82,     3198,   207870, 20163390],
       [       4,      192,    15552,   606528],
       [      38,      988,    46436,  4364984],
       [       6,      252,     2268,   102060],
       [      52,     3900,   308100, 28961400],
       [      53,     3604,   216240, 14488080]], dtype=int32)
```

17 np.percentile

- numpy.percentile()function used to compute the nth percentile of the given data (array elements) along the specified axis.

[149]:
```
a
```

[149]:
```
array([58, 69, 15, 43, 66, 72, 88, 44, 84, 68, 93, 77, 18, 89, 93])
```

[152]:
```
np.percentile(a,100)
```

[152]:
```
93.0
```

[153]:
```
np.percentile(a,0)
```

[153]:
```
15.0
```

[150]:
```
np.percentile(a,50)
```

[150]:
```
69.0
```

[151]:
```
np.median(a)
```

[151]:
```
69.0
```


18 np.histogram

Numpy has a built-in numpy.histogram() function which represents the frequency of data distribution in the graphical form.

[154]:
```
a
```

[154]:
```
array([58, 69, 15, 43, 66, 72, 88, 44, 84, 68, 93, 77, 18, 89, 93])
```

[156]:
```
np.histogram(a,bins=[0,10,20,30,40,50,60,70,80,90,100])
```

[156]:
```
(array([0, 2, 0, 0, 2, 1, 3, 2, 3, 2], dtype=int64),
 array([  0,  10,  20,  30,  40,  50,  60,  70,  80,  90, 100]))
```

[157]:
```
np.histogram(a,bins=[0,50,100])
```

[157]:
```
(array([ 4, 11], dtype=int64), array([  0,  50, 100]))
```

19 np.corrcoef

- Return Pearson product-moment correlation coefficients.

[158]:
```
salary = np.array([20000,40000,25000,35000,60000])
experience = np.array([1,3,2,4,2])

np.corrcoef(salary,experience)
```

[158]:
```
array([[1.        , 0.25344572],
       [0.25344572, 1.        ]])
```

[159]:
```
python_mark=np.array([100,90,75,57,89,45,30,10])
python_attendance=np.array([100,95,85,40,98,52,40,25])
np.corrcoef(python_mark,python_attendance)
```

[159]:
```
array([[1.        , 0.95150493],
       [0.95150493, 1.        ]])
```


20 np.isin

With the help of numpy.isin() method, we can see that one array having values are checked in a different numpy array having different elements with different sizes.

[6]:
```
a=np.array([58, 69, 15, 43, 66, 72, 88, 44, 84, 68, 93, 77, 18, 89 ,93])
```

[7]:
```
a
```

[7]:
```
array([58, 69, 15, 43, 66, 72, 88, 44, 84, 68, 93, 77, 18, 89, 93])
```

[8]:
```
items = [58,20,30,40,50,60,70,80,90,100]

a[np.isin(a,items)]
```

[8]:
```
array([58])
```

[10]:
```
items = [50,20,30,40,58,60,70,88,90,100]

a[np.isin(a,items)]
```

[10]:
```
array([58, 88])
```

21 np.flip

- The numpy.flip() function reverses the order of array elements along the specified axis, preserving the shape of the array.

[11]:
```
a
```

[11]:
```
array([58, 69, 15, 43, 66, 72, 88, 44, 84, 68, 93, 77, 18, 89, 93])
```

[12]:
```
np.flip(a)
```

[12]:
```
array([93, 89, 18, 77, 93, 68, 84, 44, 88, 72, 66, 43, 15, 69, 58])
```

[16]:
```
b=np.array([[82, 39, 65, 97],
 [ 4, 48, 81, 39],
 [38, 26, 47, 94],
 [ 6, 42,  9, 45],
 [52,75, 79, 94],
 [53, 68, 60, 67]])
```

[17]:
```
b
```

[17]:
```
array([[82, 39, 65, 97],
       [ 4, 48, 81, 39],
       [38, 26, 47, 94],
       [ 6, 42,  9, 45],
       [52, 75, 79, 94],
       [53, 68, 60, 67]])
```

[18]:
```
np.flip(b,axis=1)
```

[18]:
```
array([[97, 65, 39, 82],
       [39, 81, 48,  4],
       [94, 47, 26, 38],
       [45,  9, 42,  6],
       [94, 79, 75, 52],
       [67, 60, 68, 53]])
```

[19]:
```
np.flip(b,axis=0)
```

[19]:
```
array([[53, 68, 60, 67],
       [52, 75, 79, 94],
       [ 6, 42,  9, 45],
       [38, 26, 47, 94],
       [ 4, 48, 81, 39],
       [82, 39, 65, 97]])
```

[20]:
```
np.flip(b)
```

[20]:
```
array([[67, 60, 68, 53],
       [94, 79, 75, 52],
       [45,  9, 42,  6],
       [94, 47, 26, 38],
       [39, 81, 48,  4],
       [97, 65, 39, 82]])
```

22 np.put

- The numpy.put() function replaces specific elements of an array with given values of p_array. Array indexed works on flattened array.

[21]:
```
a
```

[21]:
```
array([58, 69, 15, 43, 66, 72, 88, 44, 84, 68, 93, 77, 18, 89, 93])
```

[22]:
```
np.put(a,[0,1],[110,530])
```

[23]:
```
a
```

[23]:
```
array([110, 530,  15,  43,  66,  72,  88,  44,  84,  68,  93,  77,  18,
        89,  93])
```

[30]:
```
b
```

[30]:
```
array([[100, 200, 300,  97],
       [  4,  48,  81,  39],
       [ 38,  26,  47,  94],
       [  6,  42,   9,  45],
       [ 52,  75,  79,  94],
       [ 53,  68,  60,  67]])
```

[31]:
```
np.put(b,[0,1],[110,530])
```

[32]:
```
b
```

[32]:
```
array([[110, 530, 300,  97],
       [  4,  48,  81,  39],
       [ 38,  26,  47,  94],
       [  6,  42,   9,  45],
       [ 52,  75,  79,  94],
       [ 53,  68,  60,  67]])
```

[33]:
```
np.put(b,[7,8],[110,530])
```

[34]:
```
b
```

[34]:
```
array([[110, 530, 300,  97],
       [  4,  48,  81, 110],
       [530,  26,  47,  94],
       [  6,  42,   9,  45],
       [ 52,  75,  79,  94],
       [ 53,  68,  60,  67]])
```

23 np.delete

The numpy.delete() function returns a new array with the deletion of sub-arrays along with the mentioned axis.

[38]:
```
a
```

[38]:
```
array([110, 530,  15,  43,  66,  72,  88,  44,  84,  68,  93,  77,  18,
        89,  93])
```

[39]:
```
np.delete(a,[0])
```

[39]:
```
array([530,  15,  43,  66,  72,  88,  44,  84,  68,  93,  77,  18,  89,
        93])
```

[40]:
```
np.delete(a,[0,2,4])
```

[40]:
```
array([530,  43,  72,  88,  44,  84,  68,  93,  77,  18,  89,  93])
```

24 Set functions

- np.union1d
- np.intersect1d
- np.setdiff1d
- np.setxor1d
- np.in1d

[42]:
```
m = np.array([1,2,3,4,5])
n = np.array([3,4,5,6,7])

np.union1d(m,n)
```

[42]:
```
array([1, 2, 3, 4, 5, 6, 7])
```

[43]:
```
np.intersect1d(m,n)
```

[43]:
```
array([3, 4, 5])
```

[44]:
```
np.setdiff1d(n,m)
```

[44]:
```
array([6, 7])
```

[45]:
```
np.setxor1d(m,n)
```

[45]:
```
array([1, 2, 6, 7])
```

[50]:
```
m[np.in1d(m,5)]
```

[50]:
```
array([5])
```

25 np.clip

- numpy.clip() function is used to Clip (limit) the values in an array.

[51]:
```
a
```

[51]:
```
array([110, 530,  15,  43,  66,  72,  88,  44,  84,  68,  93,  77,  18,
        89,  93])
```

[54]:
```
np.clip(a,a_min=20,a_max=90)
```

[54]:
```
array([90, 90, 20, 43, 66, 72, 88, 44, 84, 68, 90, 77, 20, 89, 90])
```

26 np.swapaxes

- The swapaxes() function is used to interchange two axes of an array.
- I nterchange two axes of an array.
- Syntax : numpy.swapaxes(arr, axis1, axis2)
- Parameters :
- arr : [array_like] input array.
- axis1 : [int] First axis.
- axis2 : [int] Second axis.
- Return : [ndarray]

[56]:
```
x = np.array([[1,2,3],[4,5,6]])
print(x)
print(x.shape)
print("Swapped")
x_swapped = np.swapaxes(x,0,1)
print(x_swapped)
print(x_swapped.shape)
```

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

[57]:
```
y = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
```

[58]:
```
y
```

[58]:
```
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

[59]:
```
np.swapaxes(y,1,2)
```

[59]:
```
array([[[1, 3],
        [2, 4]],

       [[5, 7],
        [6, 8]]])
```

[60]:
```
np.swapaxes(y,0,2)
```

[60]:
```
array([[[1, 5],
        [3, 7]],

       [[2, 6],
        [4, 8]]])
```


27 np.tile(A, reps)

- Construct an array by repeating A the number of times given by reps. If reps has length d, the result will have dimension of max(d, A.ndim).
- Parameters:

  A: array_like

  ```
  The input array.
  ```

  reps: array_like

  ```
  The number of repetitions of A along each axis.
  ```

- Returns

  c: ndarray

  ```
  The tiled output array.
  ```

[71]:
```
# np.tile - Example
a = np.array([0, 1, 2])
print(a)
print("Tiled")
print(np.tile(a, 2))
# Reps is given as 2 so whole array will get repeted 2 times
```

```
[0 1 2]
Tiled
[0 1 2 0 1 2]
```

[72]:
```
np.tile(a, (2, 2))
# Reps is given as (2, 2)
# means along axis-0, 2 time repetition and
# along axis-1, 2 times repetition
# Axis-0 downward along the rows
# Axis -1 rightward along columns -> or inside each rows.
```

[72]:
```
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])
```

[73]:
```
np.tile(a, (2, 3)) # Along axis-1 3 times repetition
```

[73]:
```
array([[0, 1, 2, 0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2, 0, 1, 2]])
```

28 np.repeat(a, repeats, axis=None)

- Repeat elements of an array. repeats parameter says no of time to repeat
- Parameters:

```
a: array_like

        Input array.

repeats: int or array of ints

        The number of repetitions for each element. repeats is broadcasted to fit th

axis: int, optional

        The axis along which to repeat values. By default, use the flattened input a
```

- Returns:

repeated_array: ndarray

```
        Output array which has the same shape as a, except along the given axis.
```

[75]:
```
x = np.array([[1,2],[3,4]])
print(x)
print(np.repeat(x, 2)) # Every element is getting repeted 2 times.
```

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

[76]:
```
print(x)
print(np.repeat(x, 3, axis=1)) # Alog axis-1 means rightward inside rows/ along␣
 ↪columns
# Along axis-1 columns will increase
```

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

[77]:
```
print(np.repeat(x, 3, axis=0)) # Alog axis-0 means downward to rows/ along␣
 ↪inside a column
# Along axis-0 rows will increase
```

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


29 np.allclose

- Returns True if two arrays are element-wise equal within a tolerance.
- The tolerance values are positive, typically very small numbers. The relative difference (rtol * abs(b)) and the absolute difference atol are added together to compare against the absolute difference between a and b.
- If the following equation is element-wise True, then allclose returns True.
- absolute(a - b) <= (atol + rtol * absolute(b))
- Syntax : numpy.allclose(arr1, arr2, rtol, atol, equal_nan=False)
- Parameters :

  arr1 : [array_like] Input 1st array.

  arr2 : [array_like] Input 2nd array.

  rtol : [float] The relative tolerance parameter.

  atol : [float] The absolute tolerance parameter.

  equal_nan : [bool] Whether to compare NaN's as equal.

  ```
  If True, NaN's in arr1 will be considered equal to NaN's in arr2 in the output array
  ```

Return : [bool] Returns True if the two arrays are equal within the given tolerance, otherwise it returns False.

[78]:
```
#np.allclose example
#Comparing -
a = np.array([1.1, 1.2, 1.0001])
b = np.array([1., 1.02, 1.001])
print("a",a)
print()
print("b",b)
print()
print("abs",np.abs(a-b))
print()

print("allclose",np.allclose(a,b)) # will return false
print()
print("allclose-0.2",np.allclose(a,b, atol=0.2)) # will return true, as i␣
 ↪incearse the absolute tolerance value
```

```
a [1.1    1.2    1.0001]

b [1.    1.02  1.001]

abs [0.1    0.18   0.0009]

allclose False

allclose-0.2 True
```

[79]:
```
print(np.allclose([1.0, np.nan], [1.0, np.nan])) # Nan will be taken as␣
 ↪different

print(np.allclose([1.0, np.nan], [1.0, np.nan], equal_nan=True)) # Nan will be␣
 ↪treated as same
```

```
False
True
```

[102]:
```
a1
```

[102]:
```
array([[14., 25., 10.],
       [11., 51., 57.],
       [32., 62., 86.]])
```

[84]:
```
#mean
np.mean(a1)
```

[84]:
```
43.0
```

[85]:
```
np.mean(a1,axis=0)
```

[85]:
```
array([44.33333333, 32.        , 52.66666667])
```

[87]:
```
np.mean(a1,axis=1)
```

[87]:
```
array([36.        , 37.33333333, 55.66666667])
```

[88]:
```
#median
np.median(a1)
```

[88]:
```
24.0
```

[89]:
```
np.median(a1,axis=0)
```

[89]:
```
array([24., 23., 64.])
```

[90]:
```
np.median(a1,axis=1)
```

[90]:
```
array([23., 17., 56.])
```

[91]:
```
#std
np.std(a1)
```

[91]:
```
29.416548630546945
```

[92]:
```
np.std(a1,axis=0)
```

[92]:
```
array([30.90127649, 17.1464282 , 33.62869145])
```

[93]:
```
np.std(a1,axis=1)
```

[93]:
```
array([19.8158186 , 36.05859429, 25.7207223 ])
```

[94]:
```
#var
np.var(a1)
```

[94]:
```
865.3333333333334
```

[95]:
```
np.var(a1,axis=0)
```

[95]:
```
array([ 954.88888889,  294.        , 1130.88888889])
```

[96]:
```
np.var(a1,axis=1)
```

[96]:
```
array([ 392.66666667, 1300.22222222,  661.55555556])
```

[36]:
```
# mean squared error

actual = np.random.randint(1,50,25)
predicted = np.random.randint(1,50,25)
```

```
def mse(actual,predicted):
  return np.mean((actual - predicted)**2)

mse(actual,predicted)
```

[36]:
```
406.0
```

[38]:
```
actual
```

[38]:
```
array([10, 31, 16, 17,  6, 37, 10,  2, 32,  8, 22, 12, 44, 17, 17,  2, 17,
       15, 42, 42, 17, 32, 39, 18, 41])
```

[39]:
```
predicted
```

[39]:
```
array([20, 23, 23, 14, 24, 49, 32, 41, 49, 45, 33,  3, 10,  8,  8, 15, 27,
       49, 33,  3, 24, 42, 10, 32, 49])
```

[37]:
```
# binary cross entropy
np.mean((actual - predicted)**2)
```

[3]:
```
406.0
```

[97]:
```
#trigonomoetric functions
np.sin(a1)
```

[9]:
```
array([[ 0.83665564, -0.8462204 ,  0.92002604],
       [ 0.0353983 , -0.96139749,  0.6569866 ],
       [-0.90557836, -0.521551  , -0.82181784]])
```

[98]:
```
np.sinh(a1)
```

[9]:
```
array([[6.59407867e+08, 4.87240172e+09, 3.11757454e+27],
       [8.25818127e+37, 1.20774764e+07, 5.48316123e+02],
       [1.32445611e+10, 1.04582975e+24, 3.03801511e+37]])
```

[33]:
```
a = np.arange(10)
print(np.sin(a))
```

```
[ 0.          0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427
 -0.2794155   0.6569866   0.98935825  0.41211849]
```

[34]:
```
# sigmoid
def sigmoid(array):
  return 1/(1 + np.exp(-(array)))


a = np.arange(100)

sigmoid(a)
```

[34]:
```
array([0.5       , 0.73105858, 0.88079708, 0.95257413, 0.98201379,
       0.99330715, 0.99752738, 0.99908895, 0.99966465, 0.99987661,
       0.9999546 , 0.9999833 , 0.99999386, 0.99999774, 0.99999917,
       0.99999969, 0.99999989, 0.99999996, 0.99999998, 0.99999999,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ])
```

[99]:
```
# dot product
a2 = np.arange(12).reshape(3,4)
print(a2)
print()
a3 = np.arange(12,24).reshape(4,3)
print(a3)
print()

print(np.dot(a2,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]]

[[114 120 126]
 [378 400 422]
 [642 680 718]]
```

[102]:
```
c1=np.array([[1,2,3],[4,5,6],[7,8,9]])
c2=np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])

print(np.dot(c1,c2))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-102-7b87adb09e1a> in <module>
      2 c2=np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
      3
----> 4 print(np.dot(c1,c2))

<__array_function__ internals> in dot(*args, **kwargs)

ValueError: shapes (3,3) and (4,3) not aligned: 3 (dim 1) != 4 (dim 0)
```

[103]:
```
c1=np.array([[1,2,3],[4,5,6],[7,8,9]])
c2=np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])

print(np.dot(c2,c1))
```

```
[[ 30  36  42]
 [ 66  81  96]
 [102 126 150]
 [138 171 204]]
```

[104]:
```
# log and exponents
print(np.exp(a1))
```

```
[[1.31881573e+09 9.74480345e+09 6.23514908e+27]
 [1.65163625e+38 2.41549528e+07 1.09663316e+03]
 [2.64891221e+10 2.09165950e+24 6.07603023e+37]]
```

[105]:
```
print(np.log(a1))
```

```
[[3.04452244 3.13549422 4.15888308]
 [4.47733681 2.83321334 1.94591015]
 [3.17805383 4.02535169 4.46590812]]
```

[108]:
```
#round
v1=np.random.random((2,3))*100
v1
```

[108]:
```
array([[89.06161604, 34.23808495, 84.98938361],
       [83.28381439, 27.84207591, 43.24638347]])
```

[109]:
```
np.round(v1)
```

[109]:
```
array([[89., 34., 85.],
       [83., 28., 43.]])
```

[110]:
```
#floor
np.floor(v1)
```

[110]:
```
array([[89., 34., 84.],
       [83., 27., 43.]])
```

[111]:
```
#ceil
np.ceil(v1)
```

[111]:
```
array([[90., 35., 85.],
       [84., 28., 44.]])
```

30 Indexing and Slicing

[113]:
```
a1 = np.arange(10)
a2 = np.arange(12).reshape(3,4)
a3 = np.arange(8).reshape(2,2,2)
print(a1)
print()
print(a2)
print()
print(a3)
```

```
[0 1 2 3 4 5 6 7 8 9]

[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

[114]:
```
print(a1)
print(a1[-1])
```

[114]:
```
9
```

[116]:
```
print(a1)
print(a1[0])
```

```
[0 1 2 3 4 5 6 7 8 9]
0
```

[117]:
```
print(a1)
print(a1[-3])
```

```
[0 1 2 3 4 5 6 7 8 9]
7
```

[128]:
```
print(a2)
print(a2[1])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[4 5 6 7]
```

[118]:
```
print(a2)
print(a2[1][2])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
6
```

[119]:
```
print(a2)
print(a2[-2][-2])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
6
```

[136]:
```
print(a3)
print(",,,,,,,,")
print(a3[1])
```

```
[[[0 1]
  [2 3]]

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

[121]:
```
print(a3)
print(a3[1][0][1])
```

```
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
5
```

[122]:
```
print(a3)
print(a3[-1][-2][-1])
```

```
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
5
```

[123]:
```
print(a1)
print(a1[2:5:1])
```

```
[0 1 2 3 4 5 6 7 8 9]
[2 3 4]
```

[124]:
```
print(a1)
print(a1[::-1])
```

```
[0 1 2 3 4 5 6 7 8 9]
[9 8 7 6 5 4 3 2 1 0]
```

[125]:
```
print(a1)
print(a1[-1:0:-2])
```

```
[0 1 2 3 4 5 6 7 8 9]
[9 7 5 3 1]
```

[127]:
```
print(a2)
print(".............")
print(a2[::-1])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
…
[[ 8  9 10 11]
 [ 4  5  6  7]
 [ 0  1  2  3]]
```

[131]:
```
print(a2)
print(".............")
print(a2[0,:])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
…
[0 1 2 3]
```

[132]:
```
print(a2)
print(".............")
print(a2[:,2])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
…
[ 2  6 10]
```

[133]:
```
print(a2)
print(".............")
print(a2[1:,1:3])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
…
[[ 5  6]
 [ 9 10]]
```

[134]:
```
print(a2)
print(".............")
print(a2[::2,::3])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
…
[[ 0  3]
 [ 8 11]]
```

[136]:
```
print(a2)
print(".............")
print(a2[::2,1::2])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
…
[[ 1  3]
 [ 9 11]]
```

[138]:
```
a3 = np.arange(1,28).reshape(3,3,3)
print(a3)
```

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

 [[10 11 12]
  [13 14 15]
  [16 17 18]]
```

```
 [[19 20 21]
  [22 23 24]
  [25 26 27]]]
```

[140]:
```
print(a3)
print("*"*45)
print(a3[0,1,:])
```

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

 [[10 11 12]
  [13 14 15]
  [16 17 18]]

 [[19 20 21]
  [22 23 24]
  [25 26 27]]]
*********************************************
[4 5 6]
```

[143]:
```
#print(a3)
print("*"*45)
print(a3[1])
```

```
*********************************************
[[10 11 12]
 [13 14 15]
 [16 17 18]]
```

[144]:
```
#print(a3)
print("*"*45)
print(a3[::2])
```

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

 [[19 20 21]
  [22 23 24]
  [25 26 27]]]
```

[142]:
```
#print(a3)
print("*"*45)
print(a3[:,::2,::2])
```

```
*********************************************
[[[ 1  3]
```

```
 [ 7  9]]

[[10 12]
 [16 18]]

[[19 21]
 [25 27]]]
```

[145]:
```
#print(a3)
print("*"*45)
print(a3[1,:,1])
```

```
*********************************************
[11 14 17]
```

[147]:
```
#print(a3)
print("*"*45)
print(a3[2,1:,1:])
```

```
*********************************************
[[23 24]
 [26 27]]
```

[148]:
```
#print(a3)
print("*"*45)
print(a3[::2,0,::2])
```

```
*********************************************
[[ 1  3]
 [19 21]]
```

[184]:
```
# Fancy Indexing
a = np.arange(24).reshape(6,4)
print(a)
a[:,[0,2,3]]
```

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

[184]:
```
array([[ 0,  2,  3],
       [ 4,  6,  7],
       [ 8, 10, 11],
       [12, 14, 15],
       [16, 18, 19],
       [20, 22, 23]])
```

[2]:
```
# Fancy Indexing
import numpy as np
a = np.arange(24).reshape(6,4)
print(a)
a[[0,2,3],:]
```

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

[2]:
```
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

[183]:
```
a = np.arange(24).reshape(6,4)
print(a)
a[1:5,[0,2,3]]
```

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

[186]:
```
array([[ 4,  6,  7],
       [ 8, 10, 11],
       [12, 14, 15],
       [16, 18, 19]])
```

[187]:
```
a = np.arange(24).reshape(6,4)
print(a)
a[[1,2,3],[0,2,3]]
```

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

[187]:
```
array([ 4, 10, 15])
```

[188]:
```
a = np.random.randint(1,100,24).reshape(6,4)
print(a)
```

```
[[60 88 27 43]
 [91 40 73 79]
 [78 81 54 35]
 [57 67 29 33]
 [24 48 73 21]
 [ 2 28 87 49]]
```

[16]:
```
 #Boolean Indexing
a = np.random.randint(1,100,24).reshape(6,4)
print(a)
```

```
[[43 52 42 38]
 [68 76 37 28]
 [99 79 64 96]
 [17 91 75 33]
 [34 27  9 44]
 [98 42 60 24]]
```

[17]:
```
# find all numbers greater than 50
print([a>50])
print(a[a > 50])
```

```
[array([[False,  True, False, False],
       [ True,  True, False, False],
       [ True,  True,  True,  True],
       [False,  True,  True, False],
       [False, False, False, False],
       [ True, False,  True, False]])]
[52 68 76 99 79 64 96 91 75 98 60]
```

[18]:
```
# find out even numbers
print([a % 2 == 0])
a[a % 2 == 0]
```

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

[18]:
```
array([52, 42, 38, 68, 76, 28, 64, 96, 34, 44, 98, 42, 60, 24])
```

[6]:
```
# find all numbers greater than 50 and are even

a[(a > 50) & (a % 2 == 0)]
```

[6]:
```
array([68, 90, 78, 56, 78, 94])
```

[7]:
```
# find all numbers not divisible by 7
a[~(a % 7 == 0)]
```

[7]:
```
array([ 8, 11, 69, 68,  3, 10, 50, 24, 59, 61,  6,  8, 90, 25, 78, 53, 78,
       71, 59, 94, 40, 16, 34])
```

31 Iterating

[149]:
```
print(a1)

for i in a1:
  print(i)
```

```
[0 1 2 3 4 5 6 7 8 9]
0
1
2
3
4
5
6
7
8
9
```

[150]:
```
print(a2)
print("*"*50)
for i in a2:
  print(i)
  for j in i:
    print(j)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
**************************************************
[0 1 2 3]
0
1
2
3
[4 5 6 7]
4
5
6
7
[ 8  9 10 11]
8
```

```
9
10
11
```

[157]:
```
print(a3)
print("*"*100)
for i in a3:
    print(i)
    print("*"*50)
    for j in i:
        print(j)
        for k in j:
            print(k)
```

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

 [[10 11 12]
  [13 14 15]
  [16 17 18]]

 [[19 20 21]
  [22 23 24]
  [25 26 27]]]
****************************************************************************************************
**********
[[1 2 3]
 [4 5 6]
 [7 8 9]]
**************************************************
[1 2 3]
1
2
3
[4 5 6]
4
5
6
[7 8 9]
7
8
9
[[10 11 12]
 [13 14 15]
 [16 17 18]]
**************************************************
[10 11 12]
```

```
10
11
12
[13 14 15]
13
14
15
[16 17 18]
16
17
18
[[19 20 21]
 [22 23 24]
 [25 26 27]]
**************************************************
[19 20 21]
19
20
21
[22 23 24]
22
23
24
[25 26 27]
25
26
27
```

[159]:
```
print(a3)
print(np.nditer(a3))
```

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

 [[10 11 12]
  [13 14 15]
  [16 17 18]]

 [[19 20 21]
  [22 23 24]
  [25 26 27]]]
<numpy.nditer object at 0x000001F9C5A31E90>
```

[158]:
```
for i in np.nditer(a3):
  print(i)
```

```
1
2
```

- Reshaping
- Transpose
- Ravel

[162]:
```
print(a2)
print("*"*50)
print(a2.reshape(2,6))
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
**************************************************
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
```

[163]:
```
print(a2)
print("*"*50)
print(np.transpose(a2))
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
**************************************************
```

```
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

[165]:
```
print(a2.T)
```

```
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

[166]:
```
print(a2)
print("*"*50)
print(a2.ravel())
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
**************************************************
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

[167]:
```
print(a3)
print("*"*50)
print(a3.ravel())
```

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

 [[10 11 12]
  [13 14 15]
  [16 17 18]]

 [[19 20 21]
  [22 23 24]
  [25 26 27]]]
**************************************************
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 25 26 27]
```

32 Stacking

[169]:
```
a4 = np.arange(12).reshape(3,4)
a5 = np.arange(12,24).reshape(3,4)
print(a4)
print()
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]]
```

[170]:
```
np.hstack((a4,a5))
```

[170]:
```
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]])
```

[171]:
```
# Vertical stacking
np.vstack((a4,a5))
```

[171]:
```
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]])
```

33 Splitting

[172]:
```
# horizontal splitting
a4
```

[172]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

[176]:
```
print(np.hsplit(a4,4))
```

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

[175]:
```
print(np.hsplit(a4,2))
```

[175]:
```
[array([[0, 1],
        [4, 5],
        [8, 9]]),
 array([[ 2,  3],
        [ 6,  7],
        [10, 11]])]
```

[177]:
```
# vertical splitting
print(np.vsplit(a4,3))
```

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

[178]:
```
print(np.vsplit(a4,1))
```

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

34 The array_split() function takes the following parameter values.

- array: This is the input array to be split. It is a required parameter.
- indices_or_sections: This is an integer representation of the number of the section of the array to be split. An error is raised if the number of splits specified is not possible. It is a required parameter.
- axis: This is the axis along which the split is done. It is an optional parameter.

Return value

- The array_split() function returns a list of sub-arrays of the input array.

[15]:
```
from numpy import array, array_split
# creating the input arrray
first_array = array([1, 2, 3, 4, 5, 6, 7, 8, 9])

# splitting the input array into 3 sub-arrays
my_array = array_split(first_array, 3)

# printing the split array
print(my_array)
```

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

[16]:
```
from numpy import array, array_split
# creating the input arrray
first_array = array([1, 2, 3, 4, 5, 6, 7, 8, 9])

# splitting the input array into 3 sub-arrays
my_array = array_split(first_array, 5)
```

```
# printing the split array
print(my_array)
```

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

[17]:
```
a=np.arange(12).reshape(3,4)
a
```

[17]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

[18]:
```
np.array_split(a,3,axis=1)
```

[18]:
```
[array([[0, 1],
        [4, 5],
        [8, 9]]),
 array([[ 2],
        [ 6],
        [10]]),
 array([[ 3],
        [ 7],
        [11]])]
```

[19]:
```
np.array_split(a,3,axis=0)
```

[19]:
```
[array([[0, 1, 2, 3]]), array([[4, 5, 6, 7]]), array([[ 8,  9, 10, 11]])]
```

[20]:
```
b=np.arange(12).reshape(3,2,2)
b
```

[20]:
```
array([[[ 0,  1],
        [ 2,  3]],

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

       [[ 8,  9],
        [10, 11]]])
```

[21]:
```
np.array_split(b,3,axis=0)
```

[21]:
```
[array([[[0, 1],
         [2, 3]]]),
 array([[[4, 5],
         [6, 7]]]),
 array([[[ 8,  9],
```

```
           [10, 11]]])]
```

[22]:
```
np.array_split(b,3,axis=1)
```

[22]:
```
[array([[[0, 1]],

        [[4, 5]],

        [[8, 9]]]),
 array([[[ 2,  3]],

        [[ 6,  7]],

        [[10, 11]]]),
 array([], shape=(3, 0, 2), dtype=int32)]
```

[23]:
```
np.array_split(b,3,axis=2)
```

[23]:
```
[array([[[ 0],
         [ 2]],

        [[ 4],
         [ 6]],

        [[ 8],
         [10]]]),
 array([[[ 1],
         [ 3]],

        [[ 5],
         [ 7]],

        [[ 9],
         [11]]]),
 array([], shape=(3, 2, 0), dtype=int32)]
```


35 Broadcasting

- The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations.
- The smaller array is "broadcast" across the larger array so that they have compatible shapes.

[20]:
```
# same shape
a = np.arange(6).reshape(2,3)
b = np.arange(6,12).reshape(2,3)

print(a)
```

```
print()
print(b)
print()
print(a+b)
```

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

[[ 6  7  8]
 [ 9 10 11]]

[[ 6  8 10]
 [12 14 16]]
```

[10]:
```
print(a-b)
```

```
[[-6 -6 -6]
 [-6 -6 -6]]
```

[11]:
```
print(a*b)
```

```
[[ 0  7 16]
 [27 40 55]]
```

[21]:
```
print(a*b)
```

```
[[0.         0.14285714 0.25      ]
 [0.33333333 0.4        0.45454545]]
```

[22]:
```
print(a**b)
```

```
[[       0        1      256]
 [   19683  1048576 48828125]]
```

[23]:
```
print(a%b)
```

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

[24]:
```
print(a//b)
```

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

[25]:
```
# diff shape
a = np.arange(6).reshape(2,3)
b = np.arange(3).reshape(1,3)

print(a)
print()
print(b)
```

```
print()
print(a+b)
```

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

[[0 1 2]]

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


36 Broadcasting Rules

- 1. Make the two arrays have the same number of dimensions.

If the numbers of dimensions of the two arrays are different, add new dimensions with size 1 to the head of the array with the smaller dimension.

- 2. Make each dimension of the two arrays the same size.

If the sizes of each dimension of the two arrays do not match, dimensions with size 1 are stretched to the size of the other array. If there is a dimension whose size is not 1 in either of the two arrays, it cannot be broadcasted, and an error is raised.

[26]:
```
a = np.arange(12).reshape(4,3)
b = np.arange(3)

print(a)
print()
print(b)
print()
print(a+b)
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]

[0 1 2]

[[ 0  2  4]
 [ 3  5  7]
 [ 6  8 10]
 [ 9 11 13]]
```

[27]:
```
print(a*b)
```

```
[[ 0  1  4]
 [ 0  4 10]
```

```
 [ 0  7 16]
 [ 0 10 22]]
```

[28]:
```
a = np.arange(12).reshape(3,4)
b = np.arange(3)

print(a)
print()
print(b)
print()
print(a+b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

[0 1 2]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_14796\2714742711.py in <module>
      6 print(b)
      7 print()
----> 8 print(a+b)

ValueError: operands could not be broadcast together with shapes (3,4) (3,)
```

[29]:
```
a = np.arange(3).reshape(1,3)
b = np.arange(3).reshape(3,1)

print(a)
print()
print(b)
print()
print(a+b)
```

```
[[0 1 2]]

[[0]
 [1]
 [2]]

[[0 1 2]
 [1 2 3]
 [2 3 4]]
```

[30]:
```
a = np.arange(3).reshape(1,3)
b = np.arange(4).reshape(4,1)

print(a)
print()
print(b)
print()
print(a + b)
```

```
[[0 1 2]]

[[0]
 [1]
 [2]
 [3]]

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

[31]:
```
a = np.array([1])
# shape -> (1,1)
b = np.arange(4).reshape(2,2)
# shape -> (2,2)

print(a)
print()
print(b)
print()
print(a+b)
```

```
[1]

[[0 1]
 [2 3]]

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

[32]:
```
a = np.arange(16).reshape(4,4)
b = np.arange(4).reshape(2,2)

print(a)
print(b)

print(a+b)
```

```
[[ 0  1  2  3]
```

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_14796\1506314148.py in <module>
      5 print(b)
      6
----> 7 print(a+b)

ValueError: operands could not be broadcast together with shapes (4,4) (2,2)
```

37 Working with missing values

[40]:
```
a = np.array([1,2,3,4,np.nan,6])
a
```

[40]:
```
array([ 1.,  2.,  3.,  4., nan,  6.])
```

[41]:
```
np.isnan(a)
```

[41]:
```
array([False, False, False, False,  True, False])
```

[42]:
```
a[~np.isnan(a)]
```

[42]:
```
array([1., 2., 3., 4., 6.])
```

[46]:
```
a.fill(1)
```

[47]:
```
print(a)
```

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

[48]:
```
a = np.array([1,2,3,4,np.nan,6])
a
```

[48]:
```
array([ 1.,  2.,  3.,  4., nan,  6.])
```

[56]:
```
(np.mean(np.array([1,2,3,4,np.nan,6])))
```

[56]:
```
nan
```

[54]:
```
a = np.array([1,2,3,4,np.nan,6])
a
```

```
[nan nan nan nan nan nan]
```

[59]:
```
print(a)
```

```
[nan nan nan nan nan nan]
```