NumPy (4.1)

MONA ADLAKHA Aryabhatta College, University of Delhi

References

1. Refer guidelines: Ch4: 4.1-4.2, Usage of rand(), randn() and randint() functions of NumPy
2. https://numpy.org/doc/stable/user/quickstart.html (https://numpy.org/doc/stable/user/quickstart.html)
3. W3Schools tutorial: https://www.w3schools.com/python/numpy/numpy_intro.asp (https://www.w3schools.com/python/numpy/numpy_intro.asp)

```
- homogeneous multidimensional array
- It is a table of elements (usually numbers), all of the same type,
indexed by a tuple of non-negative integers.
- In NumPy dimensions are called axes.

For example:
- the array for the coordinates of a point in 3D space, [1, 2, 1], has
one axis.
- this axis has 3 elements in it, so we say it has a length of 3.
```

In [2]:
```
import numpy as np
```

```
numpy.array function
- Can create an array from a regular Python list or tuple using the
array function.
- The type of the resulting array is deduced from the type of the
elements in the sequences.
```

In [3]:
```
# 1-D array
arr1=np.array([1,2,3,4,5]) #create using a list
print(arr1)
```

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

```
# 1-D array
arr1=np.array((1,2,3,4,5)) #create using a tuple
print(arr1)
```

In [4]:
```
# ERROR: calling array with multiple arguments, rather than providing a sin
arr1=np.array(1,2,3,4,5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
C:\Users\MONAAD~1\AppData\Local\Temp/ipykernel_33432/2955937289.py in <module>
      1 # ERROR: calling array with multiple arguments, rather than providing a single sequence as an argument
----> 2 arr1=np.array(1,2,3,4,5)

TypeError: array() takes from 1 to 2 positional arguments but 5 were given
```

In [5]:
```
# 0-D array
arr0=np.array(56)
print(arr0)
arr0.dtype
```

```
56
```

Out[5]:
```
dtype('int32')
```

The array below has 2 axes
The first axis has a length of 2, the second axis has a length of 5.

In [6]:
```
# 2-D array
# transforms sequences of sequences into two-dimensional arrays,
arr2=np.array([[1,2,3,4,5],[6,7,8,9,10]])
print(arr2)
```

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

In [7]:
```
# 3-D array
arr3=np.array([[[1,2,3,4,5],[6,7,8,9,10],[11,12,13,15,16]]])
print(arr3)
print(arr3.shape)
```

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

Data types in Numpy

[https://www.w3schools.com/python/numpy/numpy_data_types.asp (https://www.w3schools.com/python/numpy/numpy_data_types.asp)](https://www.w3schools.com/python/numpy/numpy_data_types.asp)

```
Below is a list of all data types in NumPy and the characters used to
represent them.

i - integer
b - boolean
u - unsigned integer
f - float
c - complex float
m - timedelta
M - datetime
O - object
S - string
U - unicode string
V - fixed chunk of memory for other type ( void )
```

ndarray.ndim

```
ndarray.ndim:
- the number of axes (dimensions) of the array
```

In [90]:
```
arr2.ndim
```

Out[90]:
```
2
```

ndarray.size

```
ndarray.size:
- the total number of elements of the array.
- This is equal to the product of the elements of shape.
```

In [8]:
```
print(arr2)
print(arr2.size)
```

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

ndarray.dtype

```
ndarray.dtype:
- an object describing the type of the elements in the array.
- One can create or specify dtype’s using standard Python types.
- Additionally NumPy provides types of its own
- e.g. numpy.int32, numpy.int16, and numpy.float64, etc.
```

In [92]:
```
arr2.dtype
```

Out[92]:
```
dtype('int32')
```

In [12]:
```
#The type of the array can also be explicitly specified at creation time:
c = np.array([[1, 2], [3, 4]], dtype=complex)
print(c)
print(c.dtype)
```

```
[[1.+0.j 2.+0.j]
 [3.+0.j 4.+0.j]]
complex128
```

```
ValueError: In Python ValueError is raised when the type of passed
argument to a function is unexpected/incorrect.
```

In [94]:
```
arrE = np.array(['a', '2', '3'], dtype='i')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call las
t)
C:\Users\MONAAD~1\AppData\Local\Temp/ipykernel_34036/242163293.py in <mod
ule>
----> 1 arrE = np.array(['a', '2', '3'], dtype='i')

ValueError: invalid literal for int() with base 10: 'a'
```

ndarray.shape

```
ndarray.shape
- This is a tuple of integers indicating the size of the array in each
dimension.
- For a matrix with n rows and m columns, shape will be (n,m).
- The length of the shape tuple is therefore the number of axes, ndim.
```

In [13]:
```
print(arr2)
arr2.shape
```

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

Out[13]:
```
(2, 5)
```

In [16]:
```
print(arr0)
print("dimension =", arr0.ndim) # number of axes
print("shape =", arr0.shape) #tuple of integers indicating the size of the
print("size =", arr0.size) # total number of elements
print("type =", arr0.dtype)
type(arr0)
```

```
56
dimension = 0
shape = ()
size = 1
type = int32
```

Out[16]:
```
numpy.ndarray
```

In [97]:
```
print(arr2)
print("dimension =", arr2.ndim) # number of axes
print("shape =", arr2.shape) #tuple of integers indicating the size of the
print("size =", arr2.size) # total number of elements
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
dimension = 2
shape = (2, 5)
size = 10
```

numpy.arange() function

```
numpy.arange() function
- used to generate an array with evenly spaced values within a specified
interval
- function returns a one-dimensional array of type numpy.ndarray.

SYNTAX: numpy.arange([start, ]stop, [step, ]dtype=None)
```

In [98]:
```
my_arr = np.arange(10) # returns an ndarray
print(type(my_arr))
print(my_arr)
print(my_arr.dtype)
```

```
<class 'numpy.ndarray'>
[0 1 2 3 4 5 6 7 8 9]
int32
```

In [99]:
```
a = np.arange(10.0)
print(a)
print(a.dtype)
```

```
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
float64
```

In [100]:
```
np.arange(5,9)
```
Out[100]:
```
array([5, 6, 7, 8])
```

In [17]:
```
print(np.arange(5,9))
```
```
[5 6 7 8]
```

In [ ]:

In [101]:
```
my_list = list(range(10))
print(my_arr)
```
```
[0 1 2 3 4 5 6 7 8 9]
```

In [102]:
```
my_list[6]
```
Out[102]:
```
6
```

numpy.random.randn()

```
If positive int_like arguments are provided, randn generates an array of
shape (d0, d1, ..., dn), filled with random floats sampled from a
univariate “normal” (Gaussian) distribution of mean 0 and variance 1. A
single float randomly sampled from the distribution is returned if no
argument is provided.
```

In [103]:
```
data1 = np.random.randn() # no argument is passed
data1
```
Out[103]:
```
-2.3331363347462255
```

In [104]:
```
data1 = np.random.randn(4)
print(data1)
print("dimension =", data1.ndim)
print("shape =", data1.shape)
print("size =", data1.size)
```
```
[-0.60375642 -1.04366194  1.36811314  1.5368956 ]
dimension = 1
shape = (4,)
size = 4
```

In [105]:
```
data1 = np.random.randn(4,)
data2 = np.random.randn(4,)
```

In [106]:
```
data1
```

Out[106]:
```
array([-2.0208327 , -0.93259063,  0.929466  ,  0.77663329])
```

In [107]:
```
print(data1)
```

```
[-2.0208327  -0.93259063  0.929466    0.77663329]
```

In [108]:
```
data2
```

Out[108]:
```
array([ 0.39618175,  1.19276609, -1.05046029,  0.58576348])
```

In [109]:
```
data2D = np.random.randn(4,2)
print(data2D)
print("dimension =", data2D.ndim)
print("shape =", data2D.shape)
print("size =", data2D.size)
```

```
[[ 1.21823406  1.74074758]
 [-1.11434085 -0.39568197]
 [ 0.02095032  1.15207729]
 [ 0.60003254 -0.17979428]]
dimension = 2
shape = (4, 2)
size = 8
```

In [110]:
```
data3D = np.random.randn(2,3,4)
print(data3D)
print("dimension =", data3D.ndim)
print("shape =", data3D.shape)
print("size =", data3D.size)
```

```
[[[ 0.37512274  0.89138773 -1.17377731  0.86390354]
  [-0.36918698  0.05444447 -0.18040142  0.65471241]
  [ 3.21813828 -0.25451469  1.09110046 -0.3128593 ]]

 [[ 1.18999283  0.41274646  0.13751601 -1.41654525]
  [-1.23578612  0.4592293  -0.62813189  0.07130364]
  [-0.28327309  1.46181478  0.85717345  0.84205929]]]
dimension = 3
shape = (2, 3, 4)
size = 24
```

In [ ]:
```

```

In [111]:
```
3 + data1 *10
```

Out[111]:
```
array([-17.20832704,  -6.32590627,  12.29466001,  10.76633287])
```

In [112]:
```
data1
```

Out[112]:
```
array([-2.0208327 , -0.93259063,  0.929466  ,  0.77663329])
```

In [113]:
```
data1 + data2
```

Out[113]:
```
array([-1.62465096,  0.26017546, -0.12099429,  1.36239677])
```

In [114]:
```
data3 = np.random.randn(4,3)
```

In [115]:
```
data3.shape
```

Out[115]:
```
(4, 3)
```

In [116]:
```
data1.shape
```

Out[116]:
```
(4,)
```

In [117]:
```
data1.dtype
```

Out[117]:
```
dtype('float64')
```

In [19]:
```
lst1 = [6,5,0.3,-1]
arr1 = np.array(lst1)
print(type(arr1), arr1.dtype)
```

```
<class 'numpy.ndarray'> float64
```

In [20]:
```
print(arr1)
```

```
[ 6.   5.   0.3 -1. ]
```

In [120]:
```
lst2 = [6,5,3,-1]
arr2 = np.array(lst2)
print(type(arr2), arr2.dtype, arr2, sep="....")
```

```
<class 'numpy.ndarray'>....int32....[ 6  5  3 -1]
```

In [25]:
```
lst3 = [[6,5,3,-1],[2,3,4,8]] #2-d list
arr3 = np.array(lst3)
print(type(arr3), arr3.dtype,sep="....")
print(arr3)
```

```
<class 'numpy.ndarray'>....int32
[[ 6  5  3 -1]
 [ 2  3  4  8]]
```

In [26]:
```
arr3.ndim
```
Out[26]:
```
2
```

In [27]:
```
arr3.shape
```
Out[27]:
```
(2, 4)
```

In [28]:
```
tup1 = (16,15,13,-11)
arr4 = np.array(tup1)
print(type(arr4), arr4.dtype, arr4, sep="....")
```
```
<class 'numpy.ndarray'>....int32....[ 16  15  13 -11]
```

In [29]:
```
arr5 = np.array((16,15,13,-11))
arr5
```
Out[29]:
```
array([ 16,  15,  13, -11])
```

In [22]:
```
np.zeros(3) # creates an array of zeros
```
Out[22]:
```
array([0., 0., 0.])
```

In [23]:
```
np.zeros(3).dtype
```
Out[23]:
```
dtype('float64')
```

In [31]:
```
np.zeros((2,3)) # creates an array of zeros, tuple passed for the shape
```
Out[31]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

In [24]:
```
np.zeros((2,3,4))
```
Out[24]:
```
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]])
```

In [6]:
```
#creates an array without initializing its value to any particular value
np.empty((2,3))
```
Out[6]:
```
array([[-6.95222783e-310,  6.43418863e-235,  1.29217778e-311],
       [ 1.29217778e-311,  1.97626258e-323,  5.43472210e-323]])
```

Table 4-1. Array creation functions

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype; copies the input data by default
`asarray`	Convert input to ndarray, but do not copy if the input is already an ndarray
`arange`	Like the built-in `range` but returns an ndarray instead of a list
`ones`, `ones_like`	Produce an array of all 1s with the given shape and dtype; `ones_like` takes another array and produces a ones array of the same shape and dtype
`zeros`, `zeros_like`	Like `ones` and `ones_like` but producing arrays of 0s instead
`empty`, `empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`full`, `full_like`	Produce an array of the given shape and dtype with all values set to the indicated "fill value" `full_like` takes another array and produces a filled array of the same shape and dtype
`eye`, `identity`	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

Table 4-2. NumPy data types

Type	Type code	Description
`int8`, `uint8`	`i1`, `u1`	Signed and unsigned 8-bit (1 byte) integer types
`int16`, `uint16`	`i2`, `u2`	Signed and unsigned 16-bit integer types
`int32`, `uint32`	`i4`, `u4`	Signed and unsigned 32-bit integer types
`int64`, `uint64`	`i8`, `u8`	Signed and unsigned 64-bit integer types
`float16`	`f2`	Half-precision floating point
`float32`	`f4` or `f`	Standard single-precision floating point; compatible with C float
`float64`	`f8` or `d`	Standard double-precision floating point; compatible with C double and Python `float` object
`float128`	`f16` or `g`	Extended-precision floating point
`complex64`, `complex128`, `complex256`	`c8`, `c16`, `c32`	Complex numbers represented by two 32, 64, or 128 floats, respectively
`bool`	?	Boolean type storing `True` and `False` values
`object`	O	Python object type; a value can be any Python object
`string_`	S	Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use `'S10'`
`unicode_`	U	Fixed-length Unicode type (number of bytes platform specific); same specification semantics as `string_` (e.g., `'U10'`)

In [7]:
```
x=3
type(x)
```
Out[7]:
```
int
```

In [8]:
```
a1 = np.array([1,2,3])
a1.dtype
```
Out[8]:
```
dtype('int32')
```

In [9]:
```
a2 = np.array([1,2,3], dtype = np.float64)
a2.dtype
```

Out[9]:
```
dtype('float64')
```

In [10]:
```
a2
```

Out[10]:
```
array([1., 2., 3.])
```

In [12]:
```
farr1 = a1.astype(np.float64)
print(farr1)
print(a1.dtype, farr1.dtype)
```

```
[1. 2. 3.]
int32 float64
```

In [37]:
```
a1
```

Out[37]:
```
array([1, 2, 3])
```

In [38]:
```
farr1
```

Out[38]:
```
array([1., 2., 3.])
```

In [39]:
```
a2
```

Out[39]:
```
array([1., 2., 3.])
```

In [13]:
```
a3 = np.array([1.5, 2.6, 3.1])
print(a3, a3.dtype)
iarr1 = a3.astype(np.int32)
print(iarr1)
print(a3.dtype, iarr1.dtype)
```

```
[1.5 2.6 3.1] float64
[1 2 3]
float64 int32
```

In [16]:
```
a=1.6
b=int(a)
print(a,b)
```

```
1.6 1
```

astype()

Given array of strings representing numbers, can use astype to convert them to numeric form

In [27]:
```
numeric_strings = np.array(['1.25', '-9.6d', '42'], dtype=np.string_)
print(numeric_strings.dtype)
numeric_strings.astype(float)
```

```
|S5

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
C:\Users\MONAAD~1\AppData\Local\Temp/ipykernel_33432/3555613002.py in <module>
      1 numeric_strings = np.array(['1.25', '-9.6d', '42'], dtype=np.string_)
      2 print(numeric_strings.dtype)
----> 3 numeric_strings.astype(float)

ValueError: could not convert string to float: b'-9.6d'
```

In [29]:
```
numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
print(numeric_strings.dtype)
nArr=numeric_strings.astype(float) # same as float64
print(numeric_strings, numeric_strings.dtype)
print(nArr, nArr.dtype)
```

```
|S4
[b'1.25' b'-9.6' b'42'] |S4
[ 1.25 -9.6  42.  ] float64
```

Can use another array's dtype attribute to create an array of that data type

In [17]:
```
int_array = np.arange(10)
print(int_array, int_array.dtype)
calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
print(calibers, calibers.dtype)
a = int_array.astype(calibers.dtype)  #int_array.astype(float64)
a
```

```
[0 1 2 3 4 5 6 7 8 9] int32
[0.22  0.27  0.357 0.38  0.44  0.5  ] float64
```

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

In [32]:
```
int_array
```

Out[32]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Calling astype always creates a new array (copy of the data), even if the new dtype is the same as the old dtype

In [ ]:

Arithmetic operations with scalars propagate the scalar argument to each element in the array

In [37]:
```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
print(arr)
```

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

In [38]:
```
a2 = 1 / arr
```

In [39]:
```
print(arr)
print(a2)
```

```
[[1. 2. 3.]
 [4. 5. 6.]]
[[1.         0.5        0.33333333]
 [0.25       0.2        0.16666667]]
```

In [40]:
```
arr ** 0.5
```

Out[40]:
```
array([[1.        , 1.41421356, 1.73205081],
       [2.        , 2.23606798, 2.44948974]])
```

Comparisons between arrays of the same size yield boolean arrays

In [41]:
```
arr
```

Out[41]:
```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

In [42]:
```
arr2 = np.array([[0., 4., 1.], [7., 4., 12.]])
arr2
```

Out[42]:
```
array([[ 0.,  4.,  1.],
       [ 7.,  4., 12.]])
```

In [44]:
```
arr+arr2
```

Out[44]:
```
array([[ 1.,  6.,  4.],
       [11.,  9., 18.]])
```

In [43]:
```
arr2 >= arr
```

Out[43]:
```
array([[False,  True, False],
       [ True, False,  True]])
```

In [45]:
```
arr1
```

Out[45]:
```
array([ 6. ,  5. ,  0.3, -1. ])
```

In [ ]:
```
@@@@@@####BRODCASTING#####
```

```
Operations between differently sized arrays is called broadcasting
Discussed in more detail in Appendix A.
```

Basic Indexing and Slicing

In [50]:
```
arr = np.arange(11,20)
arr
```

Out[50]:
```
array([11, 12, 13, 14, 15, 16, 17, 18, 19])
```

In [51]:
```
arr[5] #indexing
```

Out[51]:
```
16
```

In [52]:
```
arr[5:8] #slicing
```

Out[52]:
```
array([16, 17, 18])
```

If you assign a scalar value to a slice, the value is propagated (or broadcasted henceforth) to the entire selection

In [18]:
```
#assign a scalar value to a slice -
arr = np.arange(11,20)
print(arr)
print(arr[5:8])
arr[5:8] = 888
print(arr)
```

```
[11 12 13 14 15 16 17 18 19]
[16 17 18]
[ 11  12  13  14  15 888 888 888  19]
```

Numpy arrays diiffent from Python’s built-in lists:

- array slices are views on the original array=> This means that the data is not copied, and any modifications to the view will be reflected in the source array

In [19]:
```
#1. create a slice of arr
arr = np.arange(11,20)
arr_slice = arr[5:8]
arr_slice
```

Out[19]:
```
array([16, 17, 18])
```

In [20]:
```
# 2a. change values in arr_slice, the mutations are reflected in the origin
print(arr)
arr_slice[1] = 777
print(arr)
```

```
[11 12 13 14 15 16 17 18 19]
[ 11  12  13  14  15  16 777  18  19]
```

In [21]:
```
# 2b. "bare" slice [:] will assign to all values in an array
print(arr)
print(arr_slice)
arr_slice[:] = 64
print(arr)
```

```
[ 11  12  13  14  15  16 777  18  19]
[ 16 777  18]
[11 12 13 14 15 64 64 64 19]
```

copy()

If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array

In [22]:
```
arr = np.arange(10)
arr_slice = arr[5:8].copy()
print(arr_slice)
print(arr)
```

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

In [23]:
```
arr_slice[1] = 909
print(arr_slice)
print(arr)
```

```
[  5 909   7]
[0 1 2 3 4 5 6 7 8 9]
```

Higher dimentional arrays

```
Two-dimensional array
```

```
- the elements at each index are no longer scalars but rather one-
dimensional arrays
```

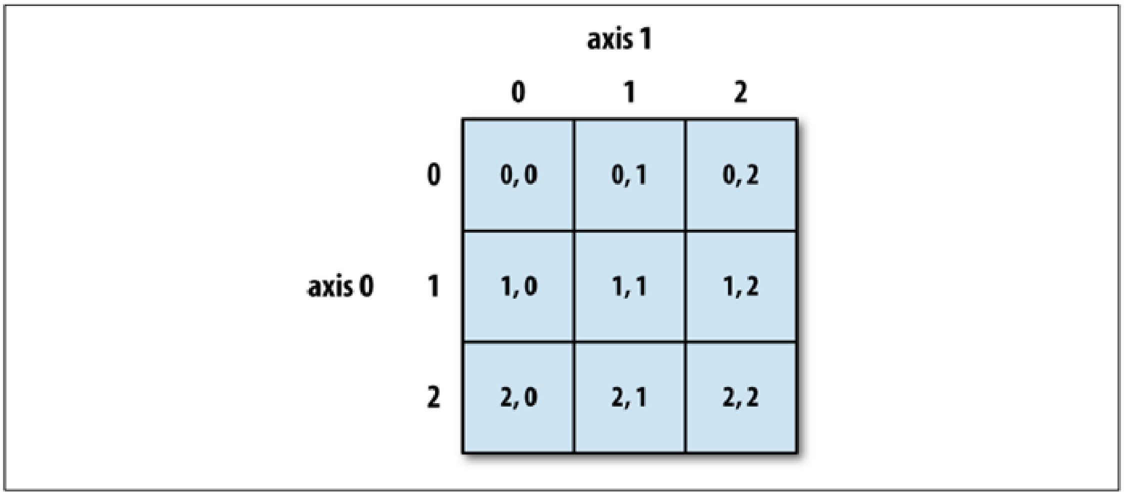

Figure 4-1. Indexing elements in a NumPy array

In [24]:
```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d)
arr2d[2]
```

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

Out[24]:
```
array([7, 8, 9])
```

In [123]:
```
# accessing individual elements
```

In [124]:
```
arr2d[0][2] #individual elements accessed recursively
```

Out[124]:
```
3
```

In [125]:
```
arr2d[0, 2] # pass a comma-separated list of indices to select individual e
```

Out[125]:
```
3
```

In [126]:
```
arr2d[0]
```

Out[126]:
```
array([1, 2, 3])
```

In [127]:
```
arr2[0]
```

Out[127]:
```
array([0., 4., 1.])
```

In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions.

In [3]:
```
# the 2 × 2 × 3 array, arr3d, arr3d[0] is a 2 × 3 array
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr3d.ndim)
print(arr3d.shape)
arr3d
```

```
3
(2, 2, 3)
```

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

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

In [4]:
```
arr3d[0][0][0] #arr3d[0,0,0]
```

Out[4]:
```
1
```

In [5]:
```
arr3d[1][1] #arr3d[1,1]
```

Out[5]:
```
array([10, 11, 12])
```

In [7]:
```
arr3d[0] # 2d array returned
```

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

In [8]:
```
print(arr3d)
old_values = arr3d[0].copy()
arr3d[0] = 42
arr3d
```

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

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

Out[8]:
```
array([[[42, 42, 42],
        [42, 42, 42]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

In [9]:
```
old_values
```

Out[9]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [10]:
```
arr3d[0] = old_values
arr3d
```

Out[10]:
```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

In [11]:
```
#arr3d[1, 0] gives all of the values whose indices start with (1, 0), form
arr3d[1, 0]
```

Out[11]:
```
array([7, 8, 9])
```

In [12]:
```
# Alternatively, indexed in two steps
# Step 1
x = arr3d[1]
print(x)
# Step 2
x[0]
```

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

Out[12]:
```
array([7, 8, 9])
```

Indexing with slices

In [70]:
```
arr = np.arange(10)
arr
```

Out[70]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [71]:
```
# ndarrays can be sliced
arr[5:8]
```

Out[71]:
```
array([5, 6, 7])
```

In [14]:
```
# 2-D Array
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2d
```

Out[14]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [15]:
```
arr2d[:2] # selects the first two rows of arr2d
```

Out[15]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

pass multiple slices

- obtain array views of the same number of dimensions.
- By mixing integer indexes and slices, get lower dimensional slices

In [22]:
```
arr2d
```

Out[22]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [23]:
```
arr2d[:2, 1:]
```

Out[23]:
```
array([[2, 3],
       [5, 6]])
```

In [24]:
```
arr2d[:1, 2:] # result is a 2d array
```

Out[24]:
```
array([[3]])
```

In [25]:
```
arr2d[:1, 2:][0]# result is a 1d array
```

Out[25]:
```
array([3])
```

In [26]:
```
arr2d[:1, 2:][0][0]# result is a 0d array
```

Out[26]:
```
3
```

In [35]:
```
a = arr2d[:1, 2:]
print(a,a.ndim)
print(a[0],a[0].ndim)
print(a[0][0],a[0][0].ndim)
```

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

In [36]:
```
arr2d
```

Out[36]:
```
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

In [37]:
```
#select the second row but only the first two columns
arr2d[1, :2]
```

Out[37]:
```
array([4, 0])
```

In [38]:
```
#select the third column but only the first two rows   *****
arr2d[:2, 2]
```

Out[38]:
```
array([0, 0])
```

In [ ]:

In [39]:
```
#colon by itself means to take the entire axis   *****
arr2d[:, :1]
```

Out[39]:
```
array([[1],
       [4],
       [7]])
```

In [40]:
```
arr2d[:2, 1:]
```

Out[40]:
```
array([[0, 0],
       [0, 0]])
```

Assigning to a slice expression assigns to the whole selection:

In [41]:
```
arr2d[:2, 1:] = 0
```

In [42]:
```
arr2d
```

Out[42]:
```
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

In [ ]:

Boolean Indexing

```
This is a type of advanced indexing which is used when the resultant
object is meant to be the result of Boolean operations, such as
comparison operators.
```

Example 1

In [43]:
```
# Example 1: items greater than 5
x = np.array([[ 0,  1,  2],[ 3,  4,  5],[ 6,  7,  8],[ 9, 10, 11]])
x
```

Out[43]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [46]:
```
a = x>5  # a is a boolean array
print(a)
# This boolean array is passed when indexing the array x
x[a]
```

```
[[False False False]
 [False False False]
 [ True  True  True]
 [ True  True  True]]
```

Out[46]:
```
array([ 6,  7,  8,  9, 10, 11])
```

In [48]:
```
# Alternatively, in single step
x[x > 5]
```

Out[48]:
```
array([ 6,  7,  8,  9, 10, 11])
```

Example 2

```
Suppose each name corresponds to a row in the data array and we wanted
to select all the rows with corresponding name 'Bob' (i.e. rows with
index 0 and 3)
```

```
Like arithmetic operations, comparisons (such as ==) with arrays are
also vectorized.
Thus, comparing names with the string 'Bob' yields a boolean array
```

In [51]:
```
# array of names with duplicates
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
names
```

Out[51]:
```
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

In [52]:
```
# generating an array using randn function generate an array of random norr
data = np.random.randn(7, 4)
data
```

Out[52]:
```
array([[ 0.73306676,  1.63158556, -0.43868445, -0.43539892],
       [ 0.35252711, -0.87424822,  0.07827878,  0.52866901],
       [-0.74617094,  0.14910413,  0.09339057, -1.2405628 ],
       [-0.4073256 ,  0.39018441, -1.04150493, -1.09103013],
       [-1.1602655 ,  0.0171261 , -1.30286833, -0.09872869],
       [ 0.42608207, -1.06788264, -0.16000847, -1.07645539],
       [-0.1165941 ,  0.01132781, -0.94498526, -0.25123313]])
```

In [55]:
```
names
```

Out[55]:
```
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

In [56]:
```
names == "Bob"  # True at index 0,3
```

Out[56]:
```
array([ True, False, False,  True, False, False, False])
```

In [61]:
```
#This boolean array can be passed when indexing the array
data[names == 'Bob'] # returns row 0 and 3 of data
```

Out[61]:
```
array([[ 0.73306676,  1.63158556, -0.43868445, -0.43539892],
       [-0.4073256 ,  0.39018441, -1.04150493, -1.09103013]])
```

```
- The boolean array must be of the same length as the array axis it’s
indexing

- Boolean selection will fail if the boolean array is not the correct
length, it is recommend care when using this feature.
```

In [62]:
```
data[names == 'Bob', 2:]
```

Out[62]:
```
array([[-0.43868445, -0.43539892],
       [-1.04150493, -1.09103013]])
```

In [63]:
```
data[names == 'Bob', 3]
```

Out[63]:
```
array([-0.43539892, -1.09103013])
```

Example 2b

In [ ]:
```
# select data for all others, except Bob
```

In [149]:
```
names != 'Bob'
```

Out[149]:
```
array([False,  True,  True, False,  True,  True,  True])
```

In [152]:
```
data[names != 'Bob']  # using != operator
```

Out[152]:
```
array([[ 0.04148935,  2.65756412,  1.9097909 , -0.66803221],
       [-0.10932864,  0.45910202, -0.52826133,  0.13182746],
       [ 0.59031437, -0.18722703, -0.21928411, -0.92838616],
       [-0.00928052, -0.70175191,  1.28834738,  0.98356384],
       [-0.15833311,  0.68302444, -2.49969158, -2.40368797]])
```

In [153]:
```
data[~(names == 'Bob')] # using ~ operator
```

Out[153]:
```
array([[ 0.04148935,  2.65756412,  1.9097909 , -0.66803221],
       [-0.10932864,  0.45910202, -0.52826133,  0.13182746],
       [ 0.59031437, -0.18722703, -0.21928411, -0.92838616],
       [-0.00928052, -0.70175191,  1.28834738,  0.98356384],
       [-0.15833311,  0.68302444, -2.49969158, -2.40368797]])
```

In [69]:
```
x = np.array([[2,4],[5,1]])
w1=np.eye(2)
print(w1)
w=np.eye(2)*x
print(w)
```

```
[[1. 0.]
 [0. 1.]]
[[2. 0.]
 [0. 1.]]
```

In [70]:
```
z=np.ones_like(x)
print(z)
```

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

In [72]:
```
a = np.ones((3,2))
print(a)
```

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

In [75]:
```
a1 = np.ones(9)
print(a1)
```

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

In [ ]:
```
aaa
```

The ~ operator can be useful when you want to invert a general condition

In [154]:
```
cond = names == 'Bob'
data[~cond]
```

Out[154]:
```
array([[ 0.04148935,  2.65756412,  1.9097909 , -0.66803221],
       [-0.10932864,  0.45910202, -0.52826133,  0.13182746],
       [ 0.59031437, -0.18722703, -0.21928411, -0.92838616],
       [-0.00928052, -0.70175191,  1.28834738,  0.98356384],
       [-0.15833311,  0.68302444, -2.49969158, -2.40368797]])
```

- The Python keywords and and or do not work with boolean arrays.
Use & (and) and | (or) instead

- Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic operators like & (and) and | (or)

In [14]:
```
names
```

Out[14]:
```
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

In [155]:
```
mask = (names == 'Bob') | (names == 'Will')
mask
```

Out[155]:
```
array([ True, False,  True,  True,  True, False, False])
```

In [156]:
```
data[mask]
```

Out[156]:
```
array([[ 0.27415301,  1.01906259,  1.47708284, -0.47090873],
       [-0.10932864,  0.45910202, -0.52826133,  0.13182746],
       [ 0.04107582,  1.38307928,  0.59278498,  1.49310956],
       [ 0.59031437, -0.18722703, -0.21928411, -0.92838616]])
```

Selecting data from an array by boolean indexing always creates a copy of the data, even if the returned array is unchanged (as in all above examples).

In [15]:
```
data = np.random.randn(7, 4)
data
```

Out[15]:
```
array([[ 1.9841926 ,  0.02490136, -1.05969915,  0.20543926],
       [ 0.76459831,  1.13837446, -0.02112344,  0.75179457],
       [ 2.55381686,  1.46109436,  1.55227011,  0.09925178],
       [ 0.11516685, -0.364902  , -1.0657805 , -0.87011183],
       [-0.19506677,  0.0107136 , -0.34065633, -0.74752875],
       [-0.16355998,  0.87912877,  0.21827526, -0.15869218],
       [ 2.08183192,  0.76867047, -0.25807129, -1.39484498]])
```

In [16]:
```
# Setting values with boolean arrays

data[data < 0] = 0
data
```

Out[16]:
```
array([[1.9841926 , 0.02490136, 0.        , 0.20543926],
       [0.76459831, 1.13837446, 0.        , 0.75179457],
       [2.55381686, 1.46109436, 1.55227011, 0.09925178],
       [0.11516685, 0.        , 0.        , 0.        ],
       [0.        , 0.0107136 , 0.        , 0.        ],
       [0.        , 0.87912877, 0.21827526, 0.        ],
       [2.08183192, 0.76867047, 0.        , 0.        ]])
```

In [17]:
```
names
```

Out[17]:
```
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

In [162]:
```
# Setting whole rows or columns using a one-dimensional boolean array
data[names != 'Joe'] = 7
data
```

Out[162]:
```
array([[7.        , 7.        , 7.        , 7.        ],
       [1.64672438, 0.87290525, 0.51320328, 0.        ],
       [7.        , 7.        , 7.        , 7.        ],
       [7.        , 7.        , 7.        , 7.        ],
       [7.        , 7.        , 7.        , 7.        ],
       [0.        , 0.92021892, 0.55098641, 0.        ],
       [0.        , 0.04456924, 0.8647162 , 0.        ]])
```

Fancy Indexing

Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays

In [19]:
```
arr = np.empty((8, 4))
```

In [20]:
```
for i in range(8):
    arr[i] = i
```

In [28]:
```
arr
```

Out[28]:
```
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],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

To select out a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order

In [30]:
```
arr[2]
```

Out[30]:
```
array([ 8,  9, 10, 11])
```

In [26]:
```
arr[4, 3, 0, 6]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
C:\Users\MONAAD~1\AppData\Local\Temp/ipykernel_2536/2175898046.py in <module>
----> 1 arr[4, 3, 0, 6]

IndexError: too many indices for array: array is 2-dimensional, but 4 were indexed
```

In [168]:
```
arr[[4, 3, 0, 6]]
```

Out[168]:
```
array([[4., 4., 4., 4.],
       [3., 3., 3., 3.],
       [0., 0., 0., 0.],
       [6., 6., 6., 6.]])
```

In [22]:
```
arr
```

Out[22]:
```
array([[0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.],
       [4., 4., 4., 4.],
       [5., 5., 5., 5.],
       [6., 6., 6., 6.],
       [7., 7., 7., 7.]])
```

In [170]:
```
arr[[-3, -5, -7]] #negative indices selects rows from the end
```

Out[170]:
```
array([[5., 5., 5., 5.],
       [3., 3., 3., 3.],
       [1., 1., 1., 1.]])
```

In [23]:
```
arr = np.arange(32)
arr
```

Out[23]:
```
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, 24, 25, 26, 27, 28, 29, 30, 31])
```

In [25]:
```
arr = np.arange(32).reshape((8, 4))
```

In [172]:
```
arr
```

Out[172]:
```
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],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

Passing multiple index arrays - selects a one-dimensional array of elements corresponding to each tuple of indices

In [175]:
```
arr[[1, 5, 7, 2]]
```

Out[175]:
```
array([[ 4,  5,  6,  7],
       [20, 21, 22, 23],
       [28, 29, 30, 31],
       [ 8,  9, 10, 11]])
```

In [174]:
```
arr[[1, 5, 7, 2], [0, 3, 1, 2]] # elements (1, 0), (5, 3), (7, 1), and (2,
```

Out[174]:
```
array([ 4, 23, 29, 10])
```

In [ ]:
```
################
```

In [31]:
```
arr
```

Out[31]:
```
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],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

In [32]:
```
arr[[1, 5, 7, 2]]
```

Out[32]:
```
array([[ 4,  5,  6,  7],
       [20, 21, 22, 23],
       [28, 29, 30, 31],
       [ 8,  9, 10, 11]])
```

In [178]:
```
arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]    #*****
```

Out[178]:
```
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

In [ ]:

Transposing Arrays and Swapping Axes

```
Transposing is a special form of reshaping that similarly returns a view
on the underlying data without copying  anything.
Arrays have the transpose method and also the special T attribute
```

In [2]:
```
arr = np.arange(15).reshape((3, 5))
arr
```

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

In [3]:
```
arr.shape
```

Out[3]:
```
(3, 5)
```

In [ ]:

In [4]:
```
arr.T # view is returned
```

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

In [5]:
```
arr.shape  # original array, no change
```

Out[5]:
```
(3, 5)
```

In [6]:
```
arr
```

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

```
inner matrix product using np.dot
```

In [7]:
```
arr = np.random.randn(6, 3)
arr
```

Out[7]:
```
array([[ 1.33947136,  0.12996198, -0.45386506],
       [ 0.28790223,  1.1020587 , -0.5691221 ],
       [-1.28681354,  1.28491604,  1.08099913],
       [-0.65904142, -0.99329287,  1.98409826],
       [ 0.38152898,  0.88826111, -1.10257725],
       [ 0.64084846,  0.21869393, -0.82499963]])
```

In [8]:
```
np.dot(arr.T, arr) #   3x5    5x3  #@@@@@@@@#
```

Out[8]:
```
array([[ 4.52354699, -0.02841369, -4.41980293],
       [-0.02841369,  4.70589829, -2.42778759],
       [-4.41980293, -2.42778759,  7.53139946]])
```

transpose()

```
https://www.geeksforgeeks.org/python-numpy-numpy-transpose/

numpy.transpose()
- It can transpose the 2-D numpy arrays
- it has no effect on 1-D arrays
Parameter: only pass (0, 1) or (1, 0)
Eg.array of shape (2, 3) to change it (3, 2) you should pass (1, 0)
where 1 as 3 and 0 as 2.
Returns: ndarray
```

In [77]:
```
arr = np.arange(15).reshape((3, 5))
arr
```

Out[77]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

In [78]:
```
arr.shape
```

Out[78]:
```
(3, 5)
```

In [79]:
```
arr.transpose(1,0) # returns transposed array
```

Out[79]:
```
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

In [80]:
```
arr.transpose(0,1) # returns original array
```

Out[80]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

In [81]:
```python
# 3-D array
arr3D = np.arange(16).reshape((2, 2, 4))
arr3D
```

Out[81]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

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

In [82]:
```python
# the axes have been reordered with the second axis first, the first axis s
# and the last axis unchanged
arr3D.transpose((1, 0, 2))
```

Out[82]:
```
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],

       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

swapaxes()

```
https://www.geeksforgeeks.org/numpy-swapaxes-function-python/

numpy.swapaxes() function interchange two axes of an array.

arr : [array_like] input array.
axis1 : [int] First axis.
axis2 : [int] Second axis.
Return : [ndarray] In earlier NumPy versions, a view of arr is returned
only if the order of the axes is changed, otherwise the input array is
returned. For NumPy >= 1.10.0,
if arr is an ndarray, then a view of arr is returned; otherwise a new
array is created
```

In [83]:
```python
arr3D
```

Out[83]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

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

In [84]:
```python
arr3D.swapaxes(1,2)
```

Out[84]:
```
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],

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

In [85]:
```python
arr3D.swapaxes(1,0)
```

Out[85]:
```
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],

       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

In [86]:
```python
arr3D # original array unchanged
```

Out[86]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

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