NumPy Tutorial.

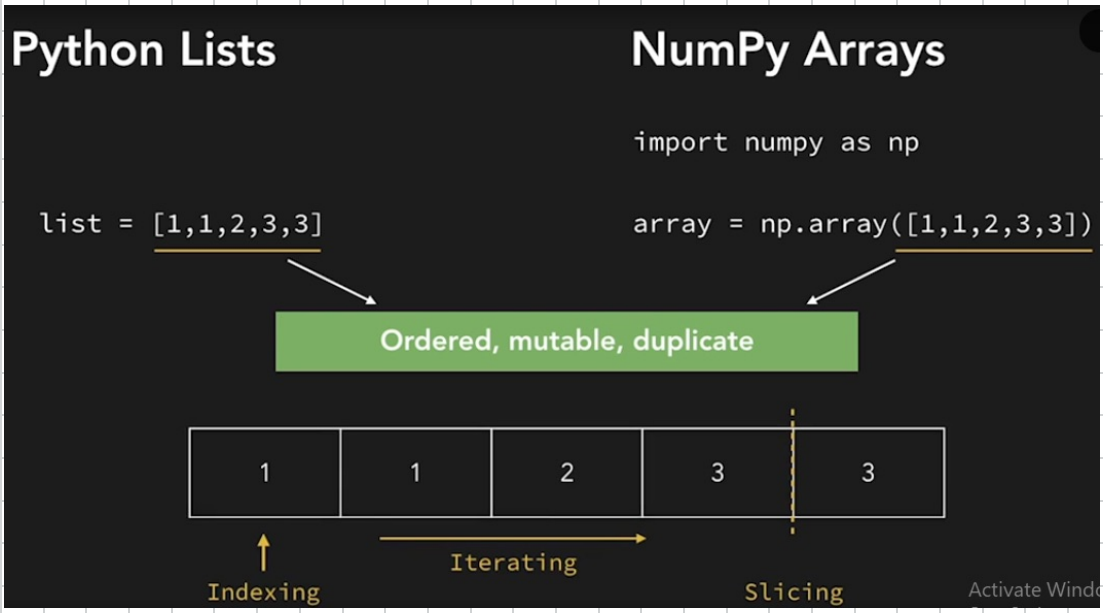
Notes made by Ayush Singh.

NumPy Course From LinkedIn.

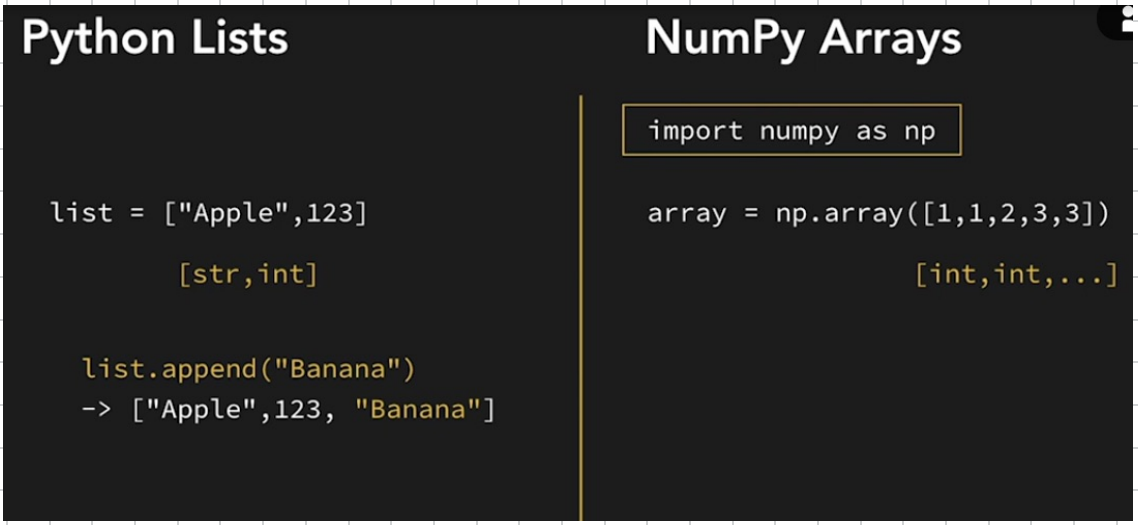
Instructor : Terezija Semenski

Python Lists	NumPy Arrays
• **No support** for vectorized operations	• **Supports** vectorized operations (addition, multiplications)
• **No fixed** type elements	• **Fixed** data type
• For loops **not efficient**	• More **efficient**

→ Similarities b/w Numpy & lists.

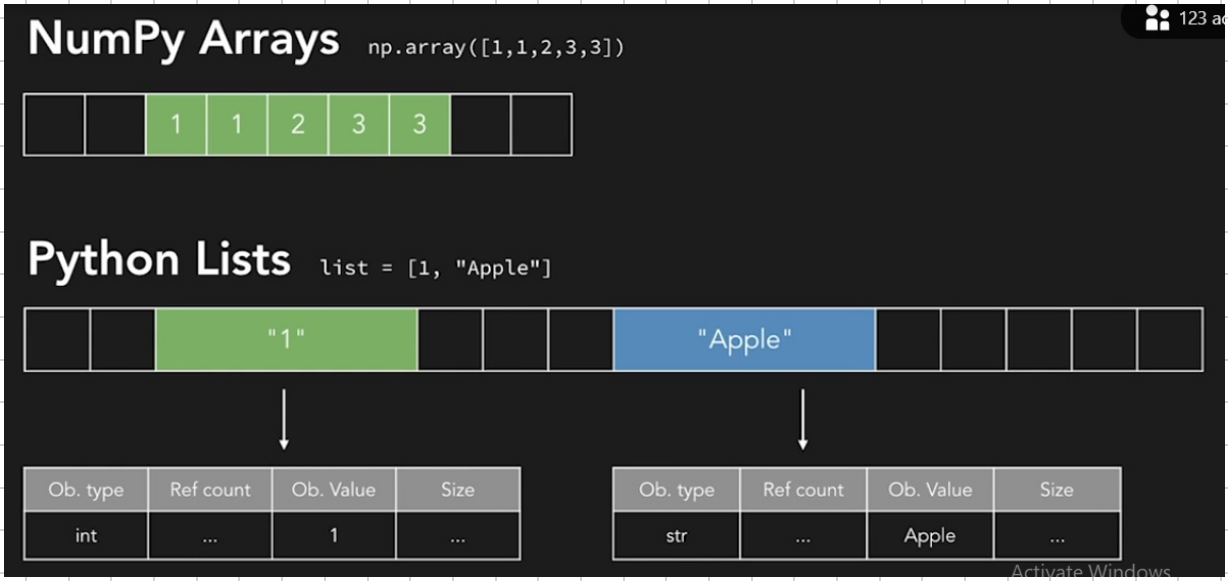


→ Differences b/w numpy array & lists.



Datatype of NumPy arrays are same.

Numpy array consumes less memory (almost 6 times less memory for storing 1000 elements). Numpy array only needs memory to store element value. While the lists have to store object type, reference count, object value & size.



Numpy arrays are faster than lists for loops. Also numpy arrays are more convenient to use especially for mathematical operations.

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

In [4]: integers=np.array([10,20,30,40,50])

In [6]: print(integers)
        [10 20 30 40 50]

In [7]: integers[0]
Out[7]: 10

In [9]: integers[0]=20
        integers
Out[9]: array([20, 20, 30, 40, 50])

In [11]: integers[0]=21.5
         integers
Out[11]: array([21, 20, 30, 40, 50])
```

← Here the values are chopped while assigning a floating

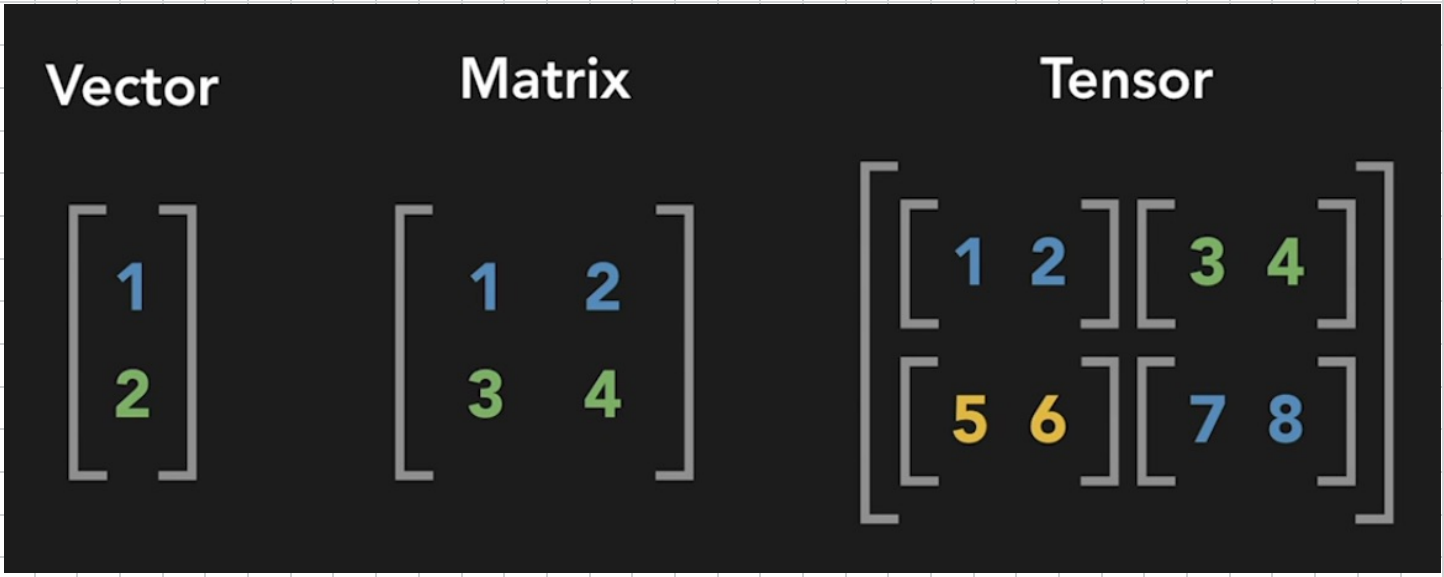
point integer as the numpy array's datatypes should be same.

```
In [12]: integers.dtype
Out[12]: dtype('int64')

In [13]: smallerIntegers=np.array(integers, dtype=np.int8)
         smallerIntegers
Out[13]: array([21, 20, 30, 40, 50], dtype=int8)

In [14]: integers.nbytes
Out[14]: 40

In [15]: smallerIntegers.nbytes
Out[15]: 5

In [16]: overflow = np.array([127,128,129], dtype = np.int8)
         overflow
Out[16]: array([ 127, -128, -127], dtype=int8)
```

```
In [17]: floats=np.array([1.2,2.3,3.4,5.1,8.3])
         print (floats)
         [1.2 2.3 3.4 5.1 8.3]

In [18]: floats.dtype
Out[18]: dtype('float64')
```

Numpy chooses a suitable datatype. We can also explicitly mention it based on the memory requirement & precision we want.

→ Multidimensional Numpy Arrays.




```
In [1]: import numpy as np

In [2]: nums=np.array([[1,2,3,4,5],[6,7,8,9,10]])
        nums

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

In [3]: nums[0,0]

Out[3]: 1

In [4]: nums[1,4]

Out[4]: 10

In [5]: nums.ndim

Out[5]: 2
```

Array of two, 2 dimen-sional array is a 3D array.

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

In [7]: multi_arr

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

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

In [8]: multi_arr[1,0,2]

Out[8]: 9
```

```
[5] test_array = np.array([[[1 , 2, 4],[5, 6,7], [2,3,4]]])

[6] test_array

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

[7] test_array.ndim

    3

[8] test_array = np.array([[[1 , 2],[5, 6], [3,4]]])
    test_array.ndim

    3
```

So the numbers of lists used to make the numpy array decides the dimension of the numpy array.

→ Creating arrays from lists & other Python structures.

```
In [1]: import numpy as np

In [2]: first_list=[1,2,3,4,5,6,7,8,9,10]

In [3]: first_list
Out[3]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [4]: first_array=np.array(first_list)

In [5]: first_array
Out[5]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

In [6]: second_list=[1,2,3,-1.23,50,128000.56,4.56]

In [7]: second_array=np.array(second_list)

In [8]: second_array
Out[8]: array([ 1.0000000e+00,  2.0000000e+00,  3.0000000e+00, -1.2300000e+00,
                5.0000000e+01,  1.2800056e+05,  4.5600000e+00])
```

When the lists have different data-types, the numpy array converts all the elements into common datatype.

```
third_list=['Ann',111111,'Peter',111112,'Susan',111113,'John',111114]

third_array=np.array(third_list)

third_array

array(['Ann', '111111', 'Peter', '111112', 'Susan', '111113', 'John',
       '111114'], dtype='<U21')
```

← all the elements are converted into common dtype which is string.

```
first_tuple=(5, 10, 15, 20, 25, 30)

array_from_tuple=np.array(first_tuple)

array_from_tuple

array([ 5, 10, 15, 20, 25, 30])

array_from_tuple.dtype

dtype('int64')

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

arr_from_multi_dim_list=np.array(multi_dim_list)

arr_from_multi_dim_list

array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
```

Creating numpy arrays with tuples & multidimensional arrays.

→ Intrinsic Numpy array creation.

Arange is used to generate evenly spaced integer number.

numpy.arange

`numpy.arange([start, ]stop, [step, ]dtype=None, *, like=None)`

Return evenly spaced values within a given interval.

Values are generated within the half-open interval `[start, stop)` (in other words, the interval including *start* but excluding *stop*). For integer arguments the function is equivalent to the Python built-in *range* function, but returns an ndarray rather than a list.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use `numpy.linspace` for these cases.

Parameters:

start : *integer or real, optional*
Start of interval. The interval includes this value. The default start value is 0.

stop : *integer or real*
End of interval. The interval does not include this value, except in some cases where *step* is not an integer and floating point round-off affects the length of *out*.

step : *integer or real, optional*
Spacing between values. For any output *out*, this is the distance between two adjacent values, `out[i+1] - out[i]`. The default step size is 1. If *step* is specified as a position argument, *start* must also be given.

```
import numpy as np
```

```
integers_array=np.arange(10)
integers_array
```

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

```
integers_second_array=np.arange(100,130)
integers_second_array
```

```
array([100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112,
       113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125,
       126, 127, 128, 129])
```

```
integers_third_array=np.arange(100,151,2)
integers_third_array
```

```
array([100, 102, 104, 106, 108, 110, 112, 114, 116, 118, 120, 122, 124,
       126, 128, 130, 132, 134, 136, 138, 140, 142, 144, 146, 148, 150])
```

Linspace can be used to generate evenly spaced floating point number.

numpy.linspace

`numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)` [source]

Return evenly spaced numbers over a specified interval.

Returns *num* evenly spaced samples, calculated over the interval [*start*, *stop*].

The endpoint of the interval can optionally be excluded.

Changed in version 1.16.0: Non-scalar *start* and *stop* are now supported.

Changed in version 1.20.0: Values are rounded towards `-inf` instead of `0` when an integer `dtype` is specified. The old behavior can still be obtained with `np.linspace(start, stop, num).astype(int)`

Parameters:

start : *array_like*
The starting value of the sequence.

stop : *array_like*
The end value of the sequence, unless *endpoint* is set to False. In that case, the sequence consists of all but the last of `num + 1` evenly spaced samples, so that *stop* is excluded. Note that the step size changes when *endpoint* is False.

```
first_floats_arr=np.linspace(10,20)
first_floats_arr
```

```
array([10.        , 10.20408163, 10.40816327, 10.6122449 , 10.81632653,
       11.02040816, 11.2244898 , 11.42857143, 11.63265306, 11.83673469,
       12.04081633, 12.24489796, 12.44897959, 12.65306122, 12.85714286,
       13.06122449, 13.26530612, 13.46938776, 13.67346939, 13.87755102,
       14.08163265, 14.28571429, 14.48979592, 14.69387755, 14.89795918,
       15.10204082, 15.30612245, 15.51020408, 15.71428571, 15.91836735,
       16.12244898, 16.32653061, 16.53061224, 16.73469388, 16.93877551,
       17.14285714, 17.34693878, 17.55102041, 17.75510204, 17.95918367,
       18.16326531, 18.36734694, 18.57142857, 18.7755102 , 18.97959184,
       19.18367347, 19.3877551 , 19.59183673, 19.79591837, 20.        ])
```

```
second_floats_arr=np.linspace(10,20,5)
second_floats_arr
```

```
array([10. , 12.5, 15. , 17.5, 20. ])
```

Linspace third argument is the number of elements we want in an array, while the third argument in arange denotes the step size.

numpy.random.rand

`random.rand(d0, d1, ..., dn)`

Random values in a given shape.

> **Note**
> This is a convenience function for users porting code from Matlab, and wraps `random_sample`. That function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like `numpy.zeros` and `numpy.ones`.

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)`.

Parameters:

d0, d1, ..., dn : *int, optional*
The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

Returns:

out : *ndarray, shape (d0, d1, ..., dn)*
Random values.

```
first_rand_arr=np.random.rand(10)
first_rand_arr
```

```
array([0.71260704, 0.39256516, 0.46726988, 0.58597865, 0.76926131,
       0.47810476, 0.83048169, 0.95593212, 0.41926085, 0.85799609])
```

```
second_rand_arr=np.random.rand(4,4)
second_rand_arr
```

```
array([[0.46697392, 0.71251143, 0.64890587, 0.43792779],
       [0.79516025, 0.1888439 , 0.04296639, 0.210475  ],
       [0.27597024, 0.96000445, 0.32202358, 0.18595349],
       [0.33870221, 0.89603373, 0.07431204, 0.79535699]])
```

numpy.random.randint

`random.randint(low, high=None, size=None, dtype=int)`

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution of the specified dtype in the "half-open" interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

> **Note**
>
> New code should use the `integers` method of a `default_rng()` instance instead; please see the Quick Start.

Parameters:

low : *int or array-like of ints*
: Lowest (signed) integers to be drawn from the distribution (unless `high=None`, in which case this parameter is one above the *highest* such integer).

high : *int or array-like of ints, optional*
: If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`). If array-like, must contain integer values

```
third_rand_arr=np.random.randint(0,100,20)
third_rand_arr
```

```
array([93, 97, 30,  5, 49, 44, 83, 14, 61, 54, 68, 96, 60, 26, 63, 44, 18,
       61, 93,  2])
```

→ Creating arrays filled with constant values.

numpy.zeros

`numpy.zeros(shape, dtype=float, order='C', *, like=None)`

Return a new array of given shape and type, filled with zeros.

Parameters:

shape : *int or tuple of ints*
: Shape of the new array, e.g., `(2, 3)` or `2`.

dtype : *data-type, optional*
: The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

order : *{'C', 'F'}, optional, default: 'C'*
: Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

like : *array_like*
: Reference object to allow the creation of arrays which are not NumPy arrays. If an array-like passed in as `like` supports the `__array_function__` protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.
New in version 1.20.0.

```
import numpy as np
```

```
first_z_array=np.zeros(5)
first_z_array
```

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

```
second_z_array=np.zeros((4,5))
second_z_array
```

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

numpy.ones

`numpy.ones(shape, dtype=None, order='C', *, like=None)` [source]

Return a new array of given shape and type, filled with ones.

Parameters:

shape : *int or sequence of ints*
: Shape of the new array, e.g., `(2, 3)` or `2`.

dtype : *data-type, optional*
: The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

order : *{'C', 'F'}, optional, default: C*
: Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

like : *array_like*
: Reference object to allow the creation of arrays which are not NumPy arrays. If an array-like passed in as `like` supports the `__array_function__` protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.
New in version 1.20.0.

```
first_ones_array=np.ones(6)
first_ones_array
```

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

```
second_ones_array=np.ones((7,8))
second_ones_array
```

```
array([[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.]])
```

```
third_ones_array=np.ones((4,5),dtype=int)
third_ones_array
```

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

Fill function is used to fill an already existing 1D array with similar value.

Full function is used to create both 1D & 2D array. Since fill function is used to fill an already existing array so we need to take the help of empty function to create an array then fill with specified value.

Empty function creates an array with unspecified datatypes.

numpy.ndarray.fill

method

`ndarray.fill(value)`

Fill the array with a scalar value.

Parameters: value : *scalar*

All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1.,  1.])
```

numpy.full

`numpy.full(shape, fill_value, dtype=None, order='C', *, like=None)` [source]

Return a new array of given shape and type, filled with *fill_value*.

Parameters:

shape : *int or sequence of ints*
Shape of the new array, e.g., `(2, 3)` or `2`.

fill_value : *scalar or array_like*
Fill value.

dtype : *data-type, optional*
The desired data-type for the array The default, None, means `np.array(fill_value).dtype`.

order : *{'C', 'F'}, optional*
Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

like : *array_like*
Reference object to allow the creation of arrays which are not NumPy arrays. If an array-like passed in as `like` supports the `__array_function__` protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.

numpy.empty

`numpy.empty(shape, dtype=float, order='C', *, like=None)`

Return a new array of given shape and type, without initializing entries.

Parameters:

shape : *int or tuple of int*
Shape of the empty array, e.g., `(2, 3)` or `2`.

dtype : *data-type, optional*
Desired output data-type for the array, e.g, **`numpy.int8`**. Default is **`numpy.float64`**.

order : *{'C', 'F'}, optional, default: 'C'*
Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

like : *array_like*
Reference object to allow the creation of arrays which are not NumPy arrays. If an array-like passed in as `like` supports the `__array_function__` protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.

```
first_fill_array=np.empty(10,dtype=int)
first_fill_array.fill(12)
first_fill_array
```

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

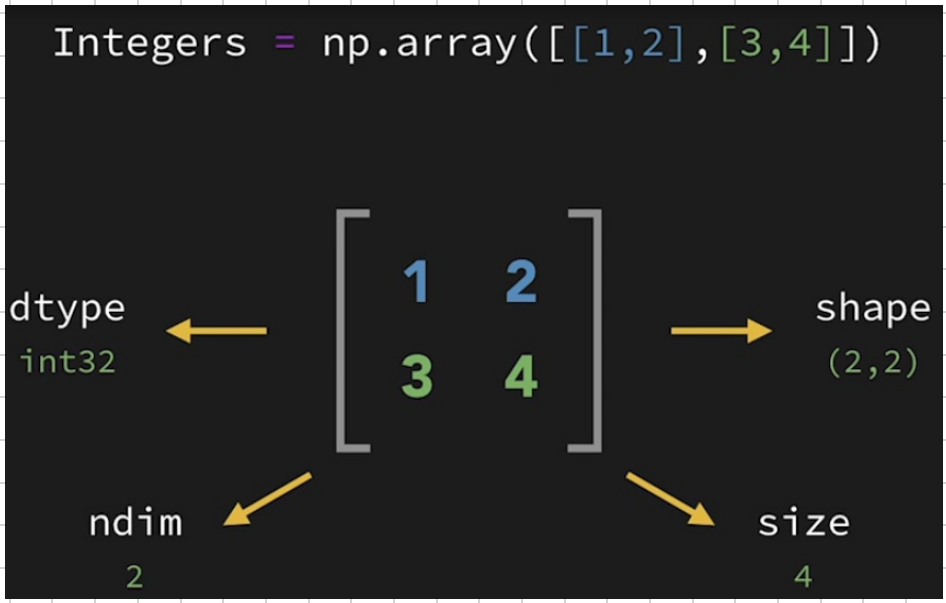
```
first_full_array=np.full(5,10)
first_full_array
```

```
array([10, 10, 10, 10, 10])
```

```
second_full_array=np.full((4,5),8)
second_full_array
```

```
array([[8, 8, 8, 8, 8],
       [8, 8, 8, 8, 8],
       [8, 8, 8, 8, 8],
       [8, 8, 8, 8, 8]])
```

→ Finding the shape and size of an array.



number of elements in each dimension

Total no. of elements

```
import numpy as np
```

```
first_arr=np.arange(20)
first_arr
```

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

```
second_arr=np.linspace((1,2),(10,20),10)
second_arr
```

```
array([[ 1.,  2.],
       [ 2.,  4.],
       [ 3.,  6.],
       [ 4.,  8.],
       [ 5., 10.],
       [ 6., 12.],
       [ 7., 14.],
       [ 8., 16.],
       [ 9., 18.],
       [10., 20.]])
```

```
third_arr=np.full((2,2,2),10)
third_arr
```

```
array([[[10, 10],
        [10, 10]],

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

```
np.shape(first_arr)
```

```
(20,)
```

```
np.shape(second_arr)
```

```
(10, 2)
```

```
np.shape(third_arr)
```

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

```
np.size(first_arr)
```

```
20
```

```
np.size(second_arr)
```

```
20
```

```
np.size(third_arr)
```

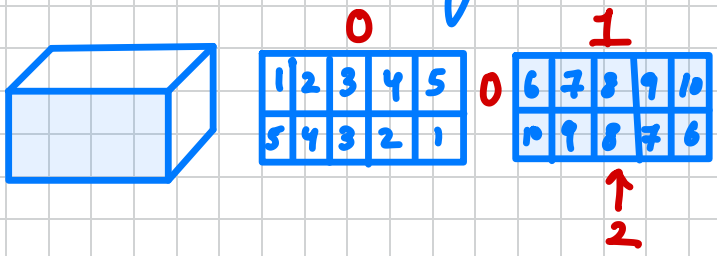
```
8
```

Which number from the multidimensional array will output?

```
import numpy as np
multi_DArray = np.array( [ [ [1,2,3,4,5],
                             [5,4,3,2,1] ],
                           [ [6,7,8,9,10],
                             [10,9,8,7,6] ] ] )
multi_DArray[1, 0, 2]
```

- 7
- 5
- 9
- 8 Correct

Its shape is kind of cuboid.



→ Adding, removing and sorting elements

```
import numpy as np
```

```
first_arr=np.array([1, 2, 3, 5])
first_arr
```

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

```
new_first_arr=np.insert(first_arr,3,4)
new_first_arr
```

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

```
second_arr=np.array([1,2,3,4])
second_arr
```

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

```
new_second_arr=np.append(second_arr,5)
new_second_arr
```

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

← adding element 3 at index 4.

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

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

```
del_arr=np.delete(third_arr,4)
del_arr
```

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

```
integers_arr=np.random.randint(0,20,20)
integers_arr
```

```
array([14, 18,  1,  6, 17,  9,  3,  9,  0, 14, 10, 15, 13, 11, 10, 13, 11,
       12, 18, 15])
```

```
print(np.sort(integers_arr))
```

```
[ 0  1  3  6  9  9 10 10 11 11 12 13 13 14 14 15 15 17 18 18]
```

```
integers_2dim_arr=np.array([[3, 2, 5,7, 4], [5, 0, 8,3, 1]])
integers_2dim_arr
```

```
array([[3, 2, 5, 7, 4],
       [5, 0, 8, 3, 1]])
```

```
print(np.sort(integers_2dim_arr))
```

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

```
colors=np.array(['orange','green','yellow','white','black','pink','blue','red'])
colors
```

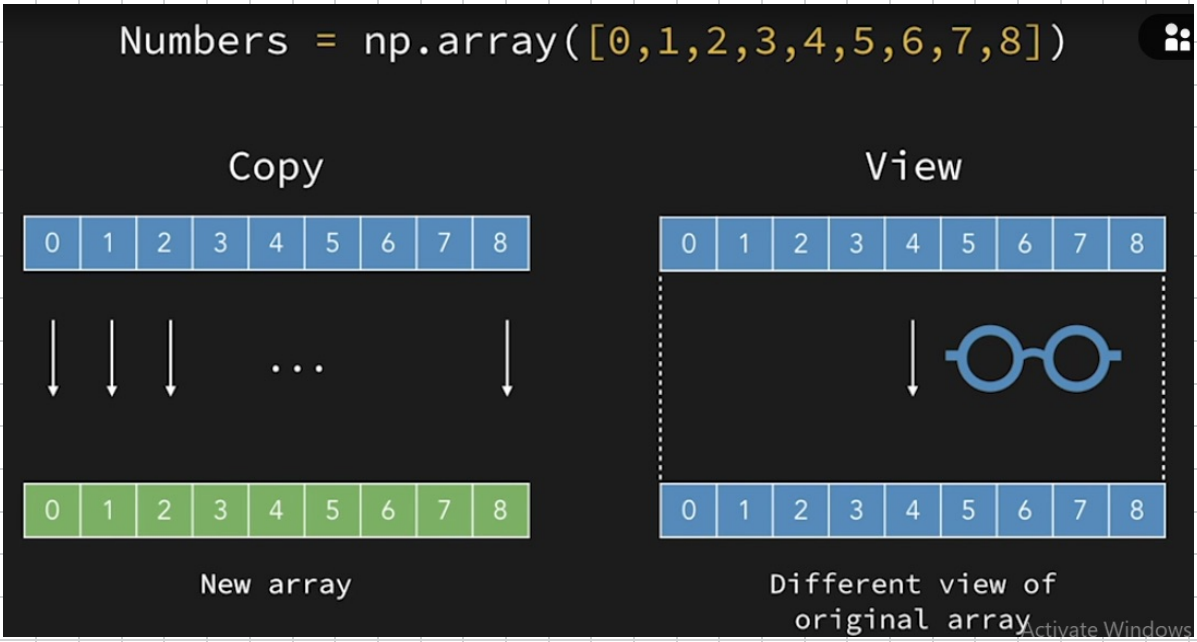
```
array(['orange', 'green', 'yellow', 'white', 'black', 'pink', 'blue',
       'red'], dtype='<U6')
```

```
print(np.sort(colors))
```

```
['black' 'blue' 'green' 'orange' 'pink' 'red' 'white' 'yellow']
```

Sort function sorts the copied numpy array. It can even sort 2d array or array of strings.

→ Copies & views.



Copy is physically stored at different location, while view gives differently named reference to the same location in the memory.

```
import numpy as np
```

```
students_ids_number=np.array([1111,1212,1313,1414,1515,1616,1717,1818])
students_ids_number
```

```
array([1111, 1212, 1313, 1414, 1515, 1616, 1717, 1818])
```

```
students_ids_number_reg=students_ids_number
print("id of students_ids_number",id(students_ids_number))
print("id of students_ids_number_reg",id(students_ids_number_reg))
```

```
id of students_ids_number 140313891242320
id of students_ids_number_reg 140313891242320
```

```
students_ids_number_reg[1]=2222
print(students_ids_number)
print(students_ids_number_reg)
```

```
[1111 2222 1313 1414 1515 1616 1717 1818]
[1111 2222 1313 1414 1515 1616 1717 1818]
```

Assignment don't make copy of the original array. Any changes we make is reflected in the original array

```
students_ids_number_cp=students_ids_number.copy()
```

```
print(students_ids_number_cp)
```

```
[1111 2222 1313 1414 1515 1616 1717 1818]
```

```
print(students_ids_number_cp==students_ids_number)
```

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

```
print ("id of students_ids_number",id(students_ids_number))
print("id of students_ids_number_cp",id(students_ids_number_cp))
```

```
id of students_ids_number 140313891242320
id of students_ids_number_cp 140313896668112
```

```
students_ids_number[0]=1000
print ("original: ", students_ids_number)
print("copy: ",students_ids_number_cp)
```

```
original:  [1000 2222 1313 1414 1515 1616 1717 1818]
copy:  [1111 2222 1313 1414 1515 1616 1717 1818]
```

← Example of making Deep copy. (copy owns the data) Changes in one is not reflected in another.

```
students_ids_number_v=students_ids_number.view()
```

```
students_ids_number_v[0]=2000
print("original: ", students_ids_number)
print("view:",students_ids_number_v)
```

```
original:  [2000 2222 1313 1414 1515 1616 1717 1818]
view: [2000 2222 1313 1414 1515 1616 1717 1818]
```

```
print(students_ids_number_cp.base)
print(students_ids_number_v.base)
```

```
None
[2000 2222 1313 1414 1515 1616 1717 1818]
```

← Base is one way to check whether the array owns the data or not.

It returns None if the array owns the data. & else refers to the original object if it doesnot own it.

→ Reshaping Arrays.

numpy.reshape¶

`numpy.reshape(a, newshape, order='C')` [source]

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

Parameters: a : *array_like*

Array to be reshaped.

newshape : *int or tuple of ints*

The new shape should be compatible with the original shape. If an integer, then the result will be a 1-D array of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.

order : *{'C', 'F', 'A'}, optional*

Read the elements of *a* using this index order, and place the elements into the reshaped array using this index order. 'C' means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of indexing. 'A' means to read / write the elements in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise.

```
first_arr=np.arange(1,13)
first_arr
```

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

```
second_arr=np.reshape(first_arr,(3,4))
second_arr
```

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

```
third_arr=np.reshape(first_arr,(6,2))
third_arr
```

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

We can only reshape when the number of elements in current array is same as number of elements after reshaping, else we will get valueError. (Ex: we cannot reshape array of size 12 into shape (4, 4))

```
fifth_arr=np.reshape(first_arr,(3,2,2))
print(fifth_arr)
print("Dimensions of fifth_arr is ",fifth_arr.ndim)
```

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

 [[ 5  6]
  [ 7  8]]

 [[ 9 10]
  [11 12]]]
Dimensions of fifth_arr is  3
```

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

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

```
seventh_arr_flat=np.reshape(sixth_arr,-1)
seventh_arr_flat
```

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

Reshaping high dimensional array into 1 dimensional array is called flattening.

numpy.ndarray.flatten

method

`ndarray.flatten(order='C')`

Return a copy of the array collapsed into one dimension.

Parameters: order : *{'C', 'F', 'A', 'K'}, optional*

'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

Returns: y : *ndarray*

A copy of the input array, flattened to one dimension.

Creates a copy, so changes in one will not be reflected in another.

numpy.ravel

`numpy.ravel(a, order='C')` [source]

Return a contiguous flattened array.

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

As of NumPy 1.10, the returned array will have the same type as the input array. (for example, a masked array will be returned for a masked array input)

Parameters: a : *array_like*

Input array. The elements in *a* are read in the order specified by *order*, and packed as a 1-D array.

order : *{'C','F', 'A', 'K'}, optional*

The elements of *a* are read using this index order. 'C' means to index the elements in row-major, C-style order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in column-major, Fortran-style order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. 'A' means to read the elements in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise. 'K' means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By

```
eighth_arr_flat=sixth_arr.flatten()
print("eighth_arr_flat:",eighth_arr_flat)
ninth_arr_rav=sixth_arr.ravel()
print("ninth_arr_rav:",ninth_arr_rav)
```

```
eighth_arr_flat: [1 2 3 4 5 6]
ninth_arr_rav: [1 2 3 4 5 6]
```

```
eighth_arr_flat[0]=100
```

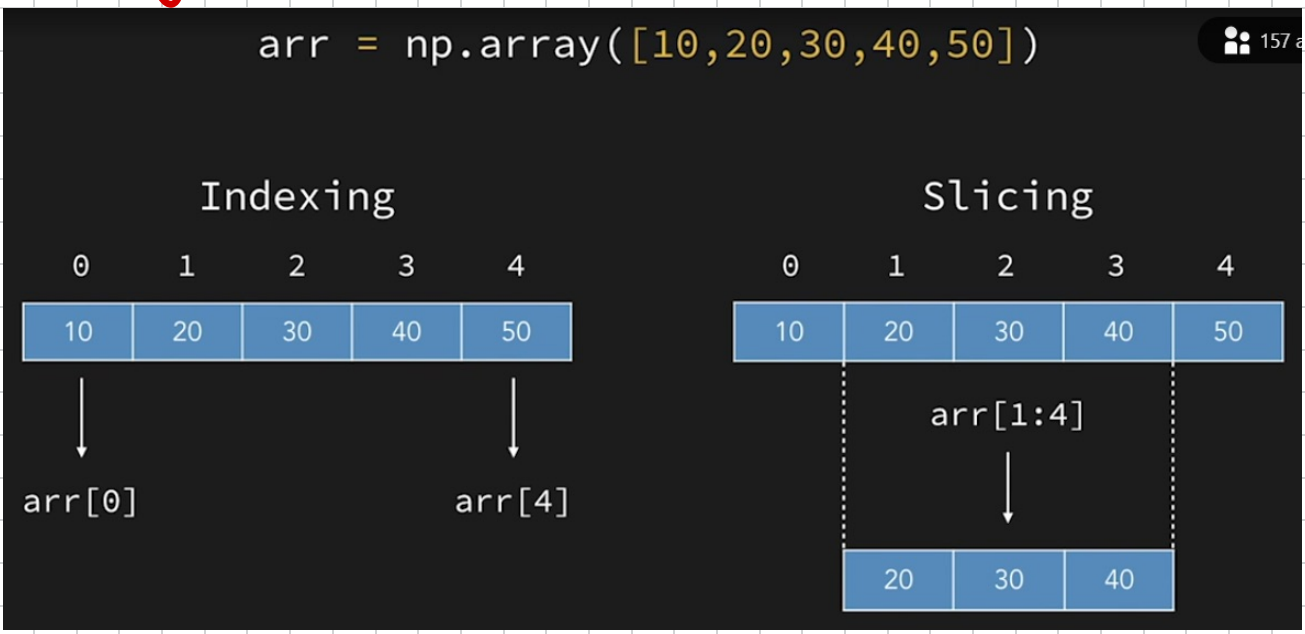
```
ninth_arr_rav[0]=200
```

```
print("eighth_arr_flat:",eighth_arr_flat)
print("ninth_arr_rav:",ninth_arr_rav)
print("sixth_arr:",sixth_arr)
```

```
eighth_arr_flat: [100   2   3   4   5   6]
ninth_arr_rav: [200   2   3   4   5   6]
sixth_arr: [[200   2]
 [  3   4]
 [  5   6]]
```

Creates a view, so changes will be reflected but it will require less memory storage

→ Indexing & slicing.




```
import numpy as np
```

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

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

```
twodim_arr[1,1]
```

```
5
```

```
twodim_arr[1]
```

```
array([4, 5, 6, 7])
```

```
threedim_arr=np.reshape(np.arange(3*4*5),(3,4,5))
threedim_arr
```

```
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, 32, 33, 34],
        [35, 36, 37, 38, 39]],

       [[40, 41, 42, 43, 44],
        [45, 46, 47, 48, 49],
        [50, 51, 52, 53, 54],
        [55, 56, 57, 58, 59]]])
```

```
threedim_arr[0,2,3]
```

```
13
```

```
threedim_arr[2,-1,-1]
```

```
59
```

```
onedim_arr=np.arange(10)
onedim_arr
```

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

```
onedim_arr[2:6]
```

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

```
onedim_arr[:5]
```

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

```
onedim_arr[-3:]
```

```
array([7, 8, 9])
```

```
onedim_arr[::2]
```

```
array([0, 2, 4, 6, 8])
```

```
twodim_arr
```

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

```
twodim_arr[1:,1:]
```

```
array([[ 5,  6,  7],
       [ 9, 10, 11]])
```

```
twodim_arr[1,:]
```

```
array([4, 5, 6, 7])
```

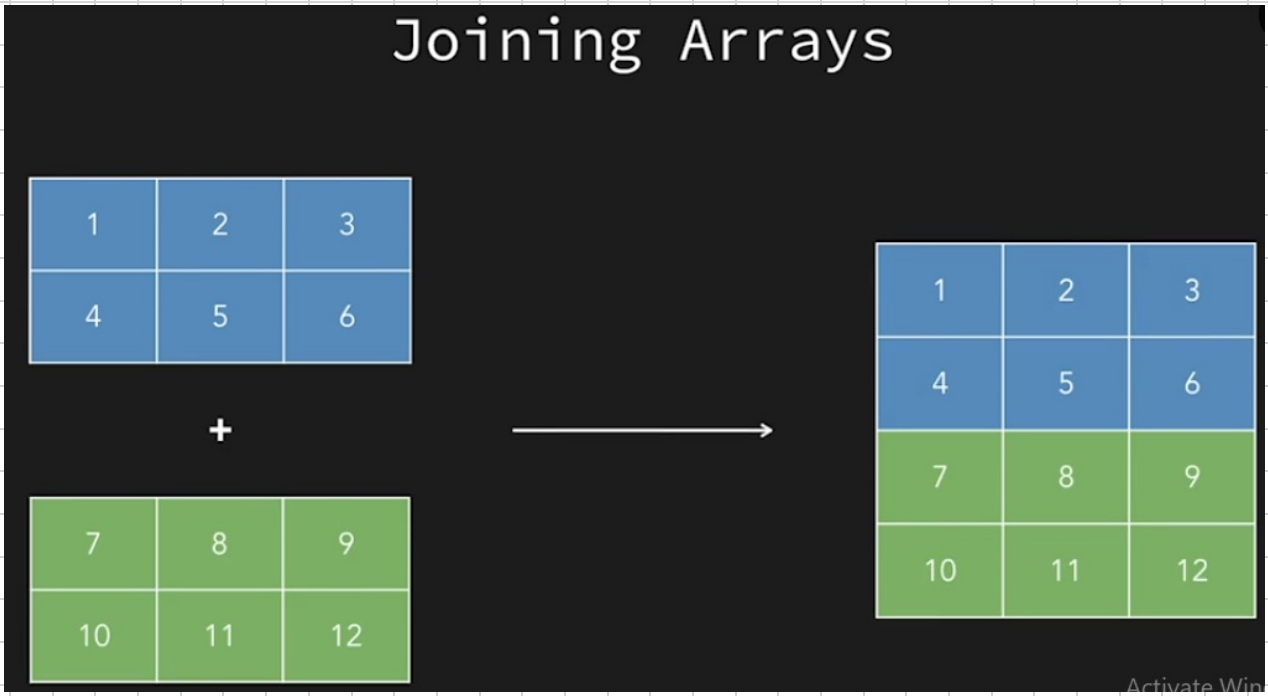
```
twodim_arr[:,2]
```

```
array([ 2,  6, 10])
```

→ Joining & splitting arrays.

Functions for Joining arrays

1) concatenate 2) stack 3) hstack 4) vstack.




```
first_arr=np.arange(1,11)
second_arr=np.arange(11,21)
print("first_arr",first_arr)
print("second_arr",second_arr)
```

```
first_arr [ 1  2  3  4  5  6  7  8  9 10]
second_arr [11 12 13 14 15 16 17 18 19 20]
```

```
con_arr=np.concatenate((first_arr,second_arr))
con_arr
```

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

```
third_2darr=np.array([[1,2,3,4,5], [6,7,8,9,10]])
third_2darr
```

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

```
fourth_2darr=np.array([[11,12,13,14,15], [16,17,18,19,20]])
fourth_2darr
```

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

```
con2d_arr = np.concatenate((third_2darr,fourth_2darr),axis=1)
con2d_arr
```

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

By default axis is 0, which means along the row, else we can specify axis explicitly

Difference b/w stack & concatenate is that, stack is done along new axis.

```
st_arr = np.stack((first_arr,second_arr))
st_arr
```

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

```
hst_arr=np.hstack((first_arr,second_arr))
hst_arr
```

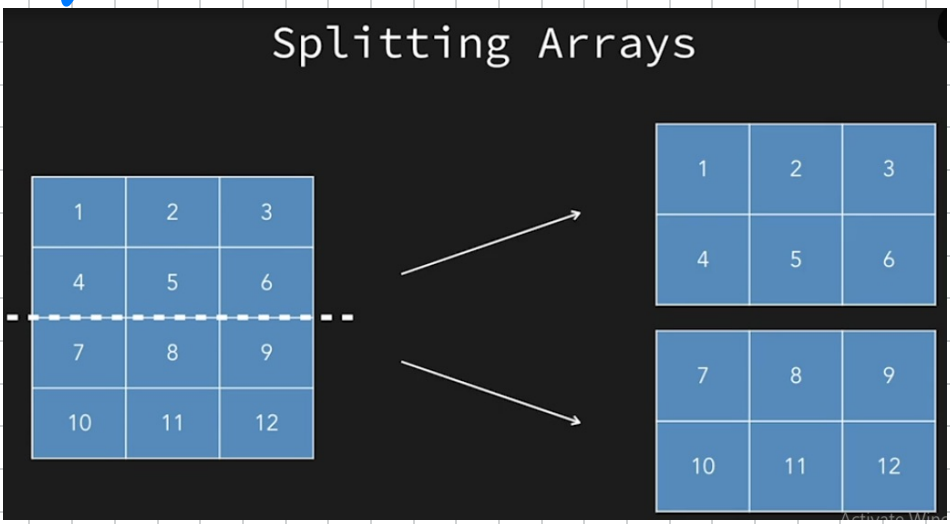
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20])
```

```
vst_arr=np.vstack((first_arr,second_arr))
vst_arr
```

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

Functions for splitting Arrays.

1) split 2) array_split 3) hsplit 4) vsplit



Splitting of arrays need to be done carefully, keeping the number of elements in mind.

```
fifth_arr=np.arange(1,13)
fifth_arr
```

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

```
sp_arr=np.array_split(fifth_arr,4)
sp_arr
```

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

```
print(sp_arr[1])
```

```
[4 5 6]
```

```
sp_arr=np.array_split(fifth_arr,8)
sp_arr
```

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

```
hs_arr=np.hsplit(third_2darr,5)
hs_arr
```

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

```
vs_arr=np.vsplit(third_2darr,2)
vs_arr
```

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

→ Arithmetic operations and functions

Vectorization helps operation can be executed in parallel on multiple elements of the array.

Vectorization is better than using 'for loops for lists!

It results in higher performing code, less verbose code & better maintainability.

```
import numpy as np
```

```
a=np.arange(1,11)
b=np.arange(21,31)
print("a",a)
print("b",b)
```

```
a [ 1  2  3  4  5  6  7  8  9 10]
b [21 22 23 24 25 26 27 28 29 30]
```

```
a+b
```

```
array([22, 24, 26, 28, 30, 32, 34, 36, 38, 40])
```

```
b-a
```

```
array([20, 20, 20, 20, 20, 20, 20, 20, 20, 20])
```

```
a*b
```

```
array([ 21,  44,  69,  96, 125, 156, 189, 224, 261, 300])
```

```
b/a
```

```
array([21.        , 11.        ,  7.66666667,  6.        ,  5.        ,
        4.33333333,  3.85714286,  3.5       ,  3.22222222,  3.        ])
```

```
c=np.arange(2,12)
print("c",c)
```

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

```
a**c
```

```
array([          1,           8,          81,        1024,
            15625,      279936,     5764801,   134217728,
       3486784401, 100000000000])
```

```
a*2
```

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

```
np.add(a,b)
```

```
array([22, 24, 26, 28, 30, 32, 34, 36, 38, 40])
```

```
np.subtract(b,a)
```

```
array([20, 20, 20, 20, 20, 20, 20, 20, 20, 20])
```

```
np.multiply(a,b)
```

```
array([ 21,  44,  69,  96, 125, 156, 189, 224, 261, 300])
```

```
np.divide(b,a)
```

```
array([21.        , 11.        ,  7.66666667,  6.        ,  5.        ,
        4.33333333,  3.85714286,  3.5       ,  3.22222222,  3.        ])
```

```
np.mod(b,a)
```

```
array([0, 0, 2, 0, 0, 2, 6, 4, 2, 0])
```

```
np.power(a,c)
```

```
array([          1,           8,          81,        1024,
            15625,      279936,     5764801,   134217728,
       3486784401, 100000000000])
```

```
np.sqrt(a)
```

```
array([1.        , 1.41421356, 1.73205081, 2.        , 2.23606798,
       2.44948974, 2.64575131, 2.82842712, 3.        , 3.16227766])
```

→ Broadcasting.

In order to apply arithmetic operations on arrays with different shapes.

Dimensions are compatible if their axes on a one-by-one basis, they have either the same length or a length of one.

Broadcasting

The term **broadcasting** describes how NumPy treats arrays with different shapes during arithmetic operations.

Subject to certain constraints, the smaller array is **broadcast** across the larger array so that they have compatible shapes.

```
d=np.arange(24).reshape(2,3,4)
d
```

```
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]]])
```

```
e=np.arange(4)
e
```

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

```
d-e
```

```
array([[[ 0,  0,  0,  0],
        [ 4,  4,  4,  4],
        [ 8,  8,  8,  8]],

       [[12, 12, 12, 12],
        [16, 16, 16, 16],
        [20, 20, 20, 20]]])
```

```
import numpy as np
```

```
a=np.arange(1,10).reshape(3,3)
a
```

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

```
b=np.arange(1,4)
b
```

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

```
a+b
```

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

← Broadcast & add the row to each element.

```
c=np.arange(1,3)
c
```

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

```
a+c
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-e200917bc55f> in <module>
----> 1 a+c

ValueError: operands could not be broadcast together with shapes (3,3) (2,)
```

← dimensions of c are not compatible with a, hence we got value error while broadcasting

→ Aggregate Functions.

```
import numpy as np
```

```
first_arr=np.arange(10,110,10)
first_arr
```

```
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
```

```
second_arr=np.arange(10,100,10).reshape(3,3)
second_arr
```

```
array([[10, 20, 30],
       [40, 50, 60],
       [70, 80, 90]])
```

```
third_arr=np.arange(10,110,10).reshape(2,5)
third_arr
```

```
array([[ 10,  20,  30,  40,  50],
       [ 60,  70,  80,  90, 100]])
```

```
first_arr.sum()
```

```
550
```

```
second_arr.sum()
```

```
450
```

```
third_arr.sum()
```

```
550
```

```
second_arr.sum(axis=0)
```

```
array([120, 150, 180])
```

```
second_arr.sum(axis=1)
```

```
array([ 60, 150, 240])
```

```
first_arr.prod()
```

```
36288000000000000
```

```
second_arr.prod()
```

```
362880000000000
```

```
third_arr.prod()
```

```
36288000000000000
```

```
third_arr.prod(axis=0)
```

```
array([ 600, 1400, 2400, 3600, 5000])
```

```
np.average(first_arr)
```

```
55.0
```

```
np.average(second_arr)
```

```
50.0
```

```
np.average(third_arr)
```

```
55.0
```

```
np.min(first_arr)
```

```
10
```

```
np.max(first_arr)
```

```
100
```

```
np.mean(first_arr)
```

```
55.0
```

```
np.std(first_arr)
```

```
28.722813232690143
```

→ How to get unique items and counts.

```
import numpy as np
```

```
first_arr=np.array([1,2,3,4,5,6,1,2,7,2,1,10,7,8])
```

```
np.unique(first_arr)
```

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

```
second_arr=np.array([[1, 1, 2,1] ,[ 3, 1, 2,1] , [1, 1, 2, 1], [ 7, 1, 1, 1]])
second_arr
```

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

```
np.unique(second_arr)
```

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

```
second_arr=np.array([[1, 1, 2,1] ,[ 3, 1, 2,1] , [1, 1, 2, 1], [ 7, 1, 1, 1]])
second_arr
```

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

```
np.unique(second_arr)
```

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

```
np.unique(second_arr,axis=0)
```

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

```
np.unique(second_arr,axis=1)
```

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

```
np.unique(first_arr,return_index= True)
```

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

```
np.unique(second_arr,return_counts=True)
```

```
(array([1, 2, 3, 7]), array([11,  3,  1,  1]))
```

→ Transpose like operations.

numpy.transpose

numpy.transpose(*a*, *axes=None*) [source]

Reverse or permute the axes of an array; returns the modified array.

For an array a with two axes, transpose(a) gives the matrix transpose.

Refer to **numpy.ndarray.transpose** for full documentation.

Parameters: a : *array_like*
Input array.

axes : *tuple or list of ints, optional*
If specified, it must be a tuple or list which contains a permutation of [0,1,..,N-1] where N is the number of axes of a. The i'th axis of the returned array will correspond to the axis numbered `axes[i]` of the input. If not specified, defaults to `range(a.ndim)[::-1]`, which reverses the order of the axes.

Returns: p : *ndarray*
a with its axes permuted. A view is returned whenever possible.

```
first_2dimarr=np.arange(12).reshape((3,4))
first_2dimarr
```

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

```
np.transpose(first_2dimarr)
```

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

```
second_2dimarr=np.arange(6).reshape(3,2)
second_2dimarr
```

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

```
np.transpose(second_2dimarr,(1,0))
```

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

Reshape makes the array by the inputs, while transpose swaps rows and columns of an array.

numpy.moveaxis

numpy.moveaxis(*a, source, destination*) [source]

Move axes of an array to new positions.

Other axes remain in their original order.

New in version 1.11.0.

Parameters: **a** : *np.ndarray*
The array whose axes should be reordered.

source : *int or sequence of int*
Original positions of the axes to move. These must be unique.

destination : *int or sequence of int*
Destination positions for each of the original axes. These must also be unique.

Returns: **result** : *np.ndarray*
Array with moved axes. This array is a view of the input array.

```
first_3dimarr=np.arange(24).reshape(2,3,4)
first_3dimarr
```

```
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]]])
```

numpy.swapaxes

numpy.swapaxes(*a, axis1, axis2*) [source]

Interchange two axes of an array.

Parameters: **a** : *array_like*
Input array.

axis1 : *int*
First axis.

axis2 : *int*
Second axis.

Returns: **a_swapped** : *ndarray*
For NumPy >= 1.10.0, if *a* is an ndarray, then a view of *a* is returned; otherwise a new array is created. For earlier NumPy versions a view of *a* is returned only if the order of the axes is changed, otherwise the input array is returned.

```
np.moveaxis(first_3dimarr,0,-1)
```

```
array([[[ 0, 12],
        [ 1, 13],
        [ 2, 14],
        [ 3, 15]],

       [[ 4, 16],
        [ 5, 17],
        [ 6, 18],
        [ 7, 19]],

       [[ 8, 20],
        [ 9, 21],
        [10, 22],
        [11, 23]]])
```

```
np.swapaxes(first_3dimarr,0,2)
```

```
array([[[ 0, 12],
        [ 4, 16],
        [ 8, 20]],

       [[ 1, 13],
        [ 5, 17],
        [ 9, 21]],

       [[ 2, 14],
        [ 6, 18],
        [10, 22]],

       [[ 3, 15],
        [ 7, 19],
        [11, 23]]])
```

→ Reversing an Array.

Flip function reverses an array along a specified axis preserving the shape of the array. It takes 2 parameters, input array & the axis along which the array is reversed.

```
import numpy as np
```

```
arr_1dim=[10,1,9,2,8,3,7,4,6,5]
arr_1dim
```

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

```
arr_1dim[::-1]
```

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

```
np.flip(arr_1dim)
```

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

```
arr_2dim=np.arange(9).reshape(3,3)
arr_2dim
```

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

```
np.flip(arr_2dim)
```

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

```
np.flip(arr_2dim,1)
```

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

```
arr_3dim=np.arange(24).reshape(2,3,4)
arr_3dim
```

```
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]]])
```

```
np.flip(arr_3dim,1)
```

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

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

```
np.flip(arr_3dim,2)
```

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

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