



Introduction to NumPy

NumPy is short for "Numerical Python" and is a powerful library that provides a broad array of functionality for performing fast numerical and mathematical operations in Python.

Key Features of NumPy:

- **NumPy's** core data structure is the **ndArray** (or n-dimensional array).
- Much of the functionality of **NumPy** is written in the **C** programming language behind the scenes to optimize performance.
- **NumPy** is used in other popular Python packages like **Pandas**, **Matplotlib**, and **scikit-learn**.

A First Look at NumPy Arrays:

At first glance, a **NumPy array** might look similar to a **List** (or in the case of multi-dimensional arrays, a List of Lists...). However, there are significant differences that make **NumPy arrays** faster and more efficient for numerical computations.

Lists vs NumPy Arrays:

- **Lists** can hold different data types (integers, strings, etc.), which can limit performance when performing mathematical operations.
- **NumPy arrays** are homogeneous, meaning all values in the array are of the same type. This uniformity allows **NumPy** to perform calculations much faster and more efficiently.

NumPy serves as the foundation for many advanced computational tasks in **Python**, including data analysis, machine learning, and artificial intelligence.

▼ Installing NumPy

```
In [14]: 1 pip install numpy
```

Requirement already satisfied: numpy in g:\anaconda\lib\site-packages (1.26.3)

Note: you may need to restart the kernel to use updated packages.

WARNING: Ignoring invalid distribution ~orch (g:\Anaconda\Lib\site-packages)
DEPRECATION: Loading egg at g:\anaconda\lib\site-packages\huggingface_hub-0.24.6-py3.8.egg is deprecated. pip 24.3 will enforce this behaviour change. A possible replacement is to use pip for package installation. Discussion can be found at <https://github.com/pypa/pip/issues/12330> (<https://github.com/pypa/pip/issues/12330>)

WARNING: Ignoring invalid distribution ~orch (g:\Anaconda\Lib\site-packages)

WARNING: Ignoring invalid distribution ~orch (g:\Anaconda\Lib\site-packages)

```
In [9]: 1 # import library
        2 import numpy as np
```

```
In [10]: 1 from numpy import *
```

```
In [11]: 1 # Creating a 1-D list
        2 list1 = [12, 3, 45, 4, 6]
```

```
In [12]: 1 list1
```

```
Out[12]: [12, 3, 45, 4, 6]
```

```
In [13]: 1 # Creating a 1-D array
        2 a = np.array(list1)
        3 a
```

```
Out[13]: array([12,  3, 45,  4,  6])
```

```
In [14]: 1 # Creating a 2-D list
        2 list2d = [[1, 2, 3], [4, 5, 6]]
        3 list2d
```

```
Out[14]: [[1, 2, 3], [4, 5, 6]]
```

```
In [15]: 1 # Creating a 2-D array
        2 b = np.array(list2d)
        3 b
```

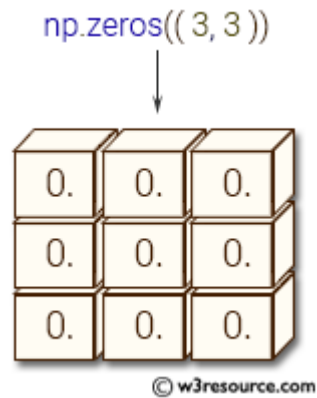
```
Out[15]: array([[1, 2, 3],
                [4, 5, 6]])
```

NumPy provides us the ability to do so much more than we could do with Lists, and at a much, much faster speed!

```
In [16]: 1 # Creating an array filled with zeros
        2 np.zeros(5)
```

```
Out[16]: array([0., 0., 0., 0., 0.])
```

- np.zeros



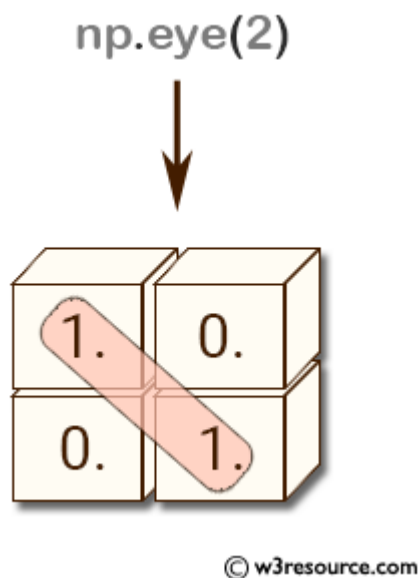
```
In [17]: 1 # Creating an array filled with ones
        2 np.ones(5)
```

```
Out[17]: array([1., 1., 1., 1., 1.])
```

```
In [18]: 1 np.eye(4)
```

```
Out[18]: array([[1., 0., 0., 0.],
                [0., 1., 0., 0.],
                [0., 0., 1., 0.],
                [0., 0., 0., 1.]])
```

- np.eye



```
In [19]: 1 np.eye(4, k=1)
```

```
Out[19]: array([[0., 1., 0., 0.],
               [0., 0., 1., 0.],
               [0., 0., 0., 1.],
               [0., 0., 0., 0.]])
```

```
In [20]: 1 np.eye(4, k=1, dtype=int)
```

```
Out[20]: array([[0, 1, 0, 0],
               [0, 0, 1, 0],
               [0, 0, 0, 1],
               [0, 0, 0, 0]])
```

```
In [21]: 1 e = np.full(4, 10)
         2 e
```

```
Out[21]: array([10, 10, 10, 10])
```

```
In [22]: 1 e = np.full((2, 2), 7)
         2 e
```

```
Out[22]: array([[7, 7],
               [7, 7]])
```

- **np.empty** : Creates a new array without initializing its entries. The values in the array are whatever happens to already exist at that memory location.

```
In [23]: 1 np.empty(4)
```

```
Out[23]: array([1., 1., 1., 1.])
```

- **The np.diag function** returns the diagonal elements of the array, or it can create a diagonal matrix from a 1D array.
- **The np.count_nonzero function** returns the number of non-zero elements in the array.

```
In [24]: 1 arr = np.array([2, 8, 2, 4])
         2 arr
```

```
Out[24]: array([2, 8, 2, 4])
```

```
In [25]: 1 diag_arr = diag(arr)
         2 diag_arr
```

```
Out[25]: array([[2, 0, 0, 0],
               [0, 8, 0, 0],
               [0, 0, 2, 0],
               [0, 0, 0, 4]])
```

```
In [26]: 1 count_nonzero(diag_arr)
```

```
Out[26]: 4
```

```
In [27]: 1 np.count_nonzero(diag_arr == 0)
```

```
Out[27]: 12
```

- **The type function** returns the type of the array.

```
In [28]: 1 b
```

```
Out[28]: array([[1, 2, 3],  
               [4, 5, 6]])
```

```
In [29]: 1 type(b)
```

```
Out[29]: numpy.ndarray
```

- **The shape function** returns the shape of the array.

```
In [30]: 1 b.shape
```

```
Out[30]: (2, 3)
```

- **The len function** returns the size of the array.

```
In [31]: 1 len(b)
```

```
Out[31]: 2
```

- **The ndim function** returns the number of dimensions of the array.

```
In [32]: 1 ndim(b)
```

```
Out[32]: 2
```

The difference between using len() and ndim in NumPy

```
In [33]: 1 arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
        2 print(len(arr_2d))  
        3  
        4 arr_1d = np.array([1, 2, 3, 4, 5])  
        5 print(len(arr_1d))  
        6
```

```
3
```

```
5
```

```
In [34]: 1 arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
2 print(arr_2d.ndim)
3
4 arr_1d = np.array([1, 2, 3, 4, 5])
5 print(arr_1d.ndim)
6
7 arr_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
8 print(arr_3d.ndim)
9
```

```
2
1
3
```

- **The size function** returns the size of the array.

```
In [35]: 1 b.size
```

```
Out[35]: 6
```

- **np.unique(k)** returns the unique elements in the array.
- **np.unique(k, return_index=True)** returns both the unique elements and their indices in the original array.

```
In [36]: 1 k = np.array([1, 2, 3, 1, 4, 2])
2 x = np.unique(k)
3 print(x)
```

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

```
In [37]: 1 import numpy as np
2
3 k = np.array([1, 2, 3, 1, 4, 2])
4 unique_values, indices = np.unique(k, return_index=True)
5 print("Unique values:", unique_values)
6 print("Indices of unique values:", indices)
7
```

```
Unique values: [1 2 3 4]
```

```
Indices of unique values: [0 1 2 4]
```

np.delete() function is used to remove elements from an array. You can specify the index (or indices) of the elements you want to delete.

```
In [38]: 1 arr = np.array([10, 20, 30, 40])
2
3 # Delete element 20 (at index 1)
4 d = np.delete(arr, 1)
5 print(d)
6
```

```
[10 30 40]
```

- **np.linspace** : Returns an array of evenly spaced values over a specified range, where you can define the number of points.

```
In [39]: 1 arr = np.linspace(0, 1, 5)
          2 arr
```

```
Out[39]: array([0. , 0.25, 0.5 , 0.75, 1.  ])
```

```
In [40]: 1 arr = np.linspace(0, 20, 5)
          2 arr
```

```
Out[40]: array([ 0.,  5., 10., 15., 20.])
```

```
In [111]: 1 np.linspace(1, 30, num=5, retstep=True)
```

```
Out[111]: (array([ 1. ,  8.25, 15.5 , 22.75, 30.  ]), 7.25)
```

- **np.arange** : Returns an array with evenly spaced values within a specified range.

```
In [41]: 1 arr = np.arange(0, 10)
          2 arr
          3
```

```
Out[41]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [42]: 1 arr = np.arange(0, 100, 2)
          2 arr
```

```
Out[42]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
                34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66,
                68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98])
```

```
In [43]: 1 arr[0:7]
```

```
Out[43]: array([ 0,  2,  4,  6,  8, 10, 12])
```

- **np.random.random** : Returns an array of random float numbers between 0 and 1 with a specified shape.

```
In [44]: 1 arr = np.random.random(2)
          2 arr
```

```
Out[44]: array([0.83132899, 0.62244802])
```

```
In [45]: 1 np.random.random((3, 4))
```

```
Out[45]: array([[0.5149957 , 0.89072198, 0.63850039, 0.39302396],
                [0.38450142, 0.17935874, 0.17906248, 0.76745543],
                [0.14021109, 0.99098683, 0.43920458, 0.05044245]])
```

```
In [46]: 1 np.random.random(1, 50, 6)
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[46], line 1  
----> 1 np.random.random(1, 50, 6)  
  
File numpy\random\mttrand.pyx:439, in numpy.random.mtrand.RandomState.random()  
  
TypeError: random() takes at most 1 positional argument (3 given)
```

```
In [ ]: 1
```

- **np.random.randint** : Generates an array of random integers within a specified range, with a defined shape.

```
In [47]: 1 arr = np.random.randint(0, 10)  
2 arr
```

```
Out[47]: 7
```

```
In [48]: 1 arr = np.random.randint(0, 10, 6)  
2 arr
```

```
Out[48]: array([4, 8, 8, 1, 3, 4])
```

```
In [49]: 1 np.random.randint(1, 100, size=(3, 6))  
2 # 3 is num of row  
3 # 6 num of col
```

```
Out[49]: array([[47, 16, 60, 36, 77, 96],  
                [20, 65, 34, 2, 65, 34],  
                [60, 81, 24, 32, 45, 42]])
```



```
In [50]: 1 np.random.randint(1, 100, size=(3, 6, 4))
        2 # 3 is num matreix
        3 # 6 is n rows and 4 is n cols
```

```
Out[50]: array([[50, 47, 71, 18],
               [16, 55, 7, 19],
               [47, 13, 71, 94],
               [1, 28, 72, 77],
               [77, 71, 12, 38],
               [34, 26, 86, 5]],

              [[22, 20, 86, 46],
               [58, 91, 89, 41],
               [13, 61, 6, 53],
               [20, 73, 23, 64],
               [48, 41, 97, 25],
               [94, 48, 86, 98]],

              [[63, 20, 93, 30],
               [40, 42, 41, 86],
               [13, 80, 95, 74],
               [78, 56, 46, 85],
               [77, 17, 16, 58],
               [60, 29, 56, 85]]])
```

- **np.random.rand()** : Returns a random float number between 0 and 1.

The `rand()` function generates random numbers from a uniform distribution over the interval [0, 1). You can specify the shape of the output array.

```
In [51]: 1 np.random.rand()
```

```
Out[51]: 0.7407834850919429
```

```
In [52]: 1 np.random.rand(15)
```

```
Out[52]: array([0.95697019, 0.67825021, 0.64047113, 0.01688786, 0.57134385,
                0.57272712, 0.0767637 , 0.13810313, 0.80279171, 0.30048646,
                0.76192702, 0.45702715, 0.69292646, 0.33026299, 0.25773429])
```

```
In [53]: 1 np.random.rand(6, 2)
```

```
Out[53]: array([[0.30561439, 0.2970974 ],
                [0.43777276, 0.66151346],
                [0.92556491, 0.807772 ],
                [0.56131113, 0.32536229],
                [0.00385748, 0.06905026],
                [0.80386826, 0.31321035]])
```

```
In [54]: 1 np.random.rand(4, 6, 2)
```

```
Out[54]: array([[0.52100952, 0.08234169],
                [0.96127324, 0.51175863],
                [0.62014095, 0.71109284],
                [0.37563643, 0.7466189 ],
                [0.64912717, 0.66498203],
                [0.92087481, 0.19520187]],

               [[0.00645547, 0.94479499],
                [0.25004584, 0.51708205],
                [0.0715918 , 0.50638825],
                [0.08494928, 0.8363789 ],
                [0.67744791, 0.43680427],
                [0.28845098, 0.50335823]],

               [[0.94243431, 0.01012649],
                [0.69730536, 0.08729183],
                [0.25825253, 0.57532119],
                [0.55700506, 0.13917815],
                [0.14232493, 0.1262439 ],
                [0.32428943, 0.08465649]],

               [[0.15927355, 0.24332263],
                [0.25126754, 0.0811254 ],
                [0.32427468, 0.33347819],
                [0.32639264, 0.42894158],
                [0.72406334, 0.4802626 ],
                [0.15647443, 0.41432929]]])
```

- **np.random.uniform** : Generates random numbers using a uniform distribution between the low and high values, and always returns floating-point numbers (float).

```
In [55]: 1 np, random.uniform()
```

```
Out[55]: (<module 'numpy' from 'g:\\Anaconda\\Lib\\site-packages\\numpy\\__init__.py'>,
          0.2561627576655623)
```

```
In [56]: 1 np, random.uniform(1, 100)
```

```
Out[56]: (<module 'numpy' from 'g:\\Anaconda\\Lib\\site-packages\\numpy\\__init__.py'>,
          65.6816054165069)
```

```
In [57]: 1 np, random.uniform(1, 100, 3)
```

```
Out[57]: (<module 'numpy' from 'g:\\Anaconda\\Lib\\site-packages\\numpy\\__init__.py'>,
          array([24.61422686, 26.29868795, 60.50486221]))
```

```
In [58]: 1 np, random.uniform((3, 10))
```

```
Out[58]: (<module 'numpy' from 'g:\\Anaconda\\Lib\\site-packages\\numpy\\__init__.py'>,
          array([1.67526209, 4.37226836]))
```

```
In [59]: 1 np.random.uniform(1, 100, (3, 3))
```

```
Out[59]: (<module 'numpy' from 'g:\Anaconda\Lib\site-packages\numpy\__init__.py'>,
array([[84.50695237, 38.6550524 , 46.10856998],
       [71.54563951, 77.20615456, 13.59929281],
       [39.59539905, 97.10476079, 99.99028198]]))
```

- **np.max** : Returns the maximum value in an array.
- **np.min** : Returns the minimum value in an array.
- **np.mean** : Calculates the average of the values in the array.
- **np.sum** : Returns the sum of all elements in the array.
- **np.std** : Calculates the standard deviation of the values in the array.



data

1	2
3	4
5	6

.max() = 6

data

1	2
3	4
5	6

.min() = 1

data

1	2
3	4
5	6

.sum() = 21

Max, Min, Mean, Sum, Standard Deviation of a 1D array

```
In [60]: 1 array_1d = np.random.randint(0, 10, 7)
2 array_1d
```

```
Out[60]: array([8, 3, 2, 7, 4, 5, 7])
```

```
In [61]: 1 array_1d.max()
```

```
Out[61]: 8
```

```
In [62]: 1 array_1d.min()
```

```
Out[62]: 2
```

```
In [63]: 1 array_1d.mean()
```

```
Out[63]: 5.142857142857143
```

```
In [64]: 1 array_1d.sum()
```

```
Out[64]: 36
```

```
In [65]: 1 array_1d.std()
```

```
Out[65]: 2.0995626366712954
```

Max, Min, Mean, Sum, Standard Deviation of a 2D array

```
In [66]: 1 array_2d = np.random.randint(0, 10, (2, 3))
        2 array_2d
```

```
Out[66]: array([[6, 4, 3],
               [5, 8, 3]])
```

```
In [67]: 1 array_2d.max()
        2 # Output: 9
        3
        4 array_2d.max(axis=0)
        5 # axis=0 means column wise
        6 # Output: array([9, 8, 7])
        7
        8 array_2d.max(axis=1)
        9 # axis=1 means row wise
       10 # Output: array([7, 9, 8])
       11
       12 array_2d.min(axis=0)
       13 # Output: array([1, 2, 3])
       14
       15 array_2d.min(axis=1)
       16 # Output: array([1, 3, 2])
       17
       18 array_2d.mean(axis=0)
       19 # Output: array([5., 5., 5.])
       20
       21 array_2d.mean(axis=1)
       22 # Output: array([4., 6., 5.])
       23
       24 array_2d.sum(axis=0)
       25 # Output: array([15, 15, 15])
       26
       27 array_2d.sum(axis=1)
       28 # Output: array([10, 18, 15])
```

```
Out[67]: array([13, 16])
```

- **argmax()** : Returns the index of the maximum value in the array .
- **argmin()** : Returns the index of the minimum value in the array .

```
In [68]: 1 b
```

```
Out[68]: array([[1, 2, 3],
               [4, 5, 6]])
```

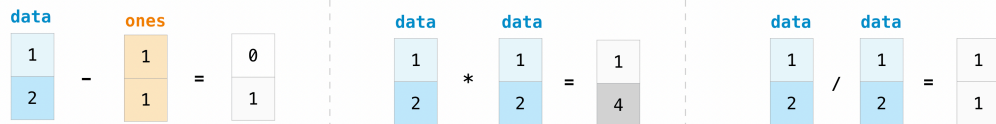
```
In [69]: 1 b.argmax()
```

```
Out[69]: 5
```

```
In [70]: 1 b.argmin()
```

```
Out[70]: 0
```

- **Arithmetic operations on a 1D array**



```
In [71]: 1 a = np.array([1, 2, 3, 4, 5])
2
3 a + 10
4 # Output: array([11, 12, 13, 14, 15])
5 a - 10
6 # Output: array([-9, -8, -7, -6, -5])
7 a * 10
8 # Output: array([10, 20, 30, 40, 50])
9 a / 10
10 # Output: array([0.1, 0.2, 0.3, 0.4, 0.5])
11 a**2
12 # Output: array([ 1,  4,  9, 16, 25])
13 a % 2
14 # Output: array([1, 0, 1, 0, 1])
15 a // 2
16 # Output: array([0, 1, 1, 2, 2])
```

Out[71]: array([0, 1, 1, 2, 2], dtype=int32)

- **Mathematical operations on a 1D array**

```
In [72]: 1 a = np.array([-2, -1, 0, 1, 2])
2
3 np.square(a)
4 # Output: array([4, 1, 0, 1, 4])
5 np.sqrt(a)
6 # Output: array([nan, nan, 0. , 1. , 1.41421356])
7 np.sin(a)
8 # Output: array([-0.9092, -0.8414, 0. , 0.8414, 0.9092])
9 np.cos(a)
10 # Output: array([-0.4161, 0.5403, 1. , 0.5403, -0.4161])
11 np.tan(a)
12 # Output: array([ 2.1850, -1.5574, 0. , 1.5574, -2.1850])
13 np.sign(a)
14 # Output: array([-1, -1, 0, 1, 1])
```

C:\Users\AL-MOSTAFA\AppData\Local\Temp\ipykernel_14772\1094679551.py:5: RuntimeWarning: invalid value encountered in sqrt
np.sqrt(a)

Out[72]: array([-1, -1, 0, 1, 1])

When you use the `sin()`..etc function in NumPy (or any similar function in Python), the angle is expected to be in radians, not degrees. If you provide an angle in degrees, you need to convert it to radians before applying the trigonometric function.

```
In [73]: 1 a = sin(30)
         2 a
```

Out[73]: -0.9880316240928618

```
In [74]: 1 # Convert the angle from degrees to radians
         2 angle_radians = np.radians(30)
         3
         4 # Calculate the sine of the angle in degrees
         5 a = np.sin(angle_radians)
         6 print(a)
         7
```

0.49999999999999994

```
In [75]: 1 # Convert 30 to radians
         2 a = np.sin(30 * np.pi / 180)
         3 print(a)
         4
```

0.49999999999999994

```
In [76]: 1 a = sin(deg2rad(45))
         2 a.round(3)
```

Out[76]: 0.707

- **Arithmetic on multiple arrays & dot product**

```
In [77]: 1 a = np.array([1, 2, 3])
         2 b = np.array([4, 5, 6])
         3
         4 np.add(a, b)
         5 # Output: array([5, 7, 9])
         6 np.subtract(a, b)
         7 # Output: array([-3, -3, -3])
         8 np.multiply(a, b)
         9 # Output: array([ 4, 10, 18])
        10 np.divide(a, b)
        11 # Output: array([0.25, 0.4 , 0.5 ])
        12 np.dot(a, b)
        13 # Output: 32
        14
```

Out[77]: 32

- **Array comparison**

```
In [78]: 1 a = np.array([1, 2, 3])
        2 b = np.array([4, 2, 6])
        3
        4 a == b
        5 # Output: array([False,  True, False])
        6
```

Out[78]: array([False, True, False])

```
In [79]: 1 np.array_equal(a, b)
        2 # Output: False
```

Out[79]: False

- **Accessing elements in a 1D array**

```
In [80]: 1 a = np.array([1, 2, 3, 4, 5])
        2
        3 a[0]
        4
```

Out[80]: 1

```
In [81]: 1 a[1:4]
```

Out[81]: array([2, 3, 4])

```
In [82]: 1 print(a[-1]) # last element
        2 a[-2] # second last element
```

5

Out[82]: 4

- **Accessing elements in a 2D array**

```
In [83]: 1 a = np.array([[1, 2, 3], [4, 5, 6]])
        2
        3 a[0]
```

Out[83]: array([1, 2, 3])

```
In [84]: 1 a[0][1]
```

Out[84]: 2

```
In [85]: 1 a[1][2]
```

Out[85]: 6

▼ 1D Array Slicing

In NumPy, **slicing** allows you to extract specific parts of an array without having to manually loop through the elements. The slicing operation uses the syntax:

```
array[start:stop:step]
```

Where:

- **start** is the index to start slicing (inclusive).
- **stop** is the index to end slicing (exclusive).
- **step** is the interval between each element to be selected.

```
In [86]: 1 arr = np.random.randint(1,10,10)
         2 arr
```

```
Out[86]: array([5, 6, 2, 1, 1, 4, 1, 5, 8, 4])
```

```
In [87]: 1 # Slice from index 2 to 7 (exclusive)
         2 sliced_arr = arr[2:7]
         3
         4 print(sliced_arr)
```

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

```
In [88]: 1 # Slice from index 2 to 7 (exclusive), step 2
         2
         3 sliced_arr = arr[2:7:2]
         4
         5 print(sliced_arr)
```

```
[2 1 1]
```

```
In [89]: 1 arr[:] # Selects all elements
```

```
Out[89]: array([5, 6, 2, 1, 1, 4, 1, 5, 8, 4])
```

▼ 2D Array Slicing

For a **2D array**, you can slice rows and columns separately. The syntax becomes:

```
array[start_row:stop_row:step_row, start_col:stop_col:step_col]
```

- **start_row:stop_row:step_row** : Slices the rows.
- **start_col:stop_col:step_col** : Slices the columns.

```
In [90]: 1 arr = np.random.randint(1,25,(4,4))
         2 arr
```

```
Out[90]: array([[15,  9, 11,  5],
                [ 6,  1,  7, 14],
                [12,  1,  9, 15],
                [15,  8,  3, 23]])
```



```
In [91]: 1 sliced_arr_2d = arr[0:3, 1:3]
        2 print(sliced_arr_2d)
```

```
[[ 9 11]
 [ 1  7]
 [ 1  9]]
```

```
In [92]: 1 sliced_arr_2d = arr[:, :] # Selects all elements.
        2 print(sliced_arr_2d)
```

```
[[15  9 11  5]
 [ 6  1  7 14]
 [12  1  9 15]
 [15  8  3 23]]
```

```
In [93]: 1 arr[3:] # Starts at index 3 and goes to the end.
```

```
Out[93]: array([[15,  8,  3, 23]])
```

```
In [94]: 1 arr[:3] # Selects elements from the beginning up to index 3 (exclusive).
```

```
Out[94]: array([[15,  9, 11,  5],
                [ 6,  1,  7, 14],
                [12,  1,  9, 15]])
```

```
In [95]: 1 arr[::-1] # Reverses the array.
```

```
Out[95]: array([[15,  8,  3, 23],
                [12,  1,  9, 15],
                [ 6,  1,  7, 14],
                [15,  9, 11,  5]])
```

reverses the elements of the array, starting from the last element and stepping backwards to the first element.

In [96]:

```
1 import numpy as np
2
3 arr = np.array([[0, 1, 2],
4                 [3, 4, 5],
5                 [6, 7, 8]])
6
7 # Reversing rows (vertical reversal)
8 reversed_rows = arr[::-1, :]
9 print("Reversed rows:")
10 print(reversed_rows)
11
12 # Reversing columns (horizontal reversal)
13 reversed_columns = arr[:, ::-1]
14 print("\nReversed columns:")
15 print(reversed_columns)
16
17 # Reversing both rows and columns
18 reversed_both = arr[::-1, ::-1]
19 print("\nReversed both rows and columns:")
20 print(reversed_both)
21
```

Reversed rows:

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

Reversed columns:

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

Reversed both rows and columns:

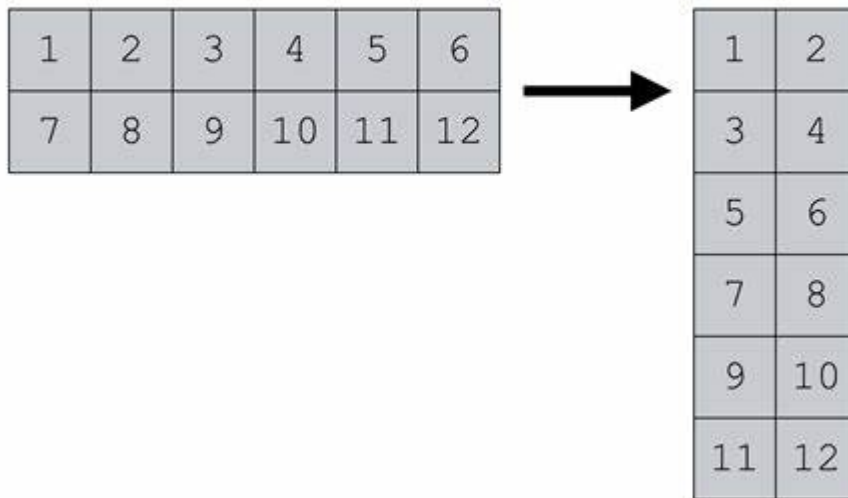
```
[[8 7 6]
 [5 4 3]
 [2 1 0]]
```

- **The reshape function** returns a new array with the specified shape, without changing the original data.



`reshape()` takes a NumPy array of one shape ...

And reconfigures it into an array with a new shape



```
In [97]: 1 b = [1, 5, 9, 7, 6, 3, 4, 7, 5, 1, 5, 12]
         2 b
```

```
Out[97]: [1, 5, 9, 7, 6, 3, 4, 7, 5, 1, 5, 12]
```

```
In [98]: 1 len(b)
```

```
Out[98]: 12
```

```
In [99]: 1 s = np.reshape(b, (1, 12))
         2 s
```

```
Out[99]: array([[ 1,  5,  9,  7,  6,  3,  4,  7,  5,  1,  5, 12]])
```

```
In [100]: 1 c = np.array(b)
          2 c
```

```
Out[100]: array([ 1,  5,  9,  7,  6,  3,  4,  7,  5,  1,  5, 12])
```

```
In [101]: 1 len(c)
```

```
Out[101]: 12
```

```
In [102]: 1 c.reshape(1, 12)
```

```
Out[102]: array([[ 1,  5,  9,  7,  6,  3,  4,  7,  5,  1,  5, 12]])
```

Row × Column = Size: The reshaped array `s` has 1 row and 12 columns, which results in a size of 12 elements ($1 * 12 = 12$).

```
In [103]: 1 c.reshape(2, 6)
```

```
Out[103]: array([[ 1,  5,  9,  7,  6,  3],
                 [ 4,  7,  5,  1,  5, 12]])
```

```
In [104]: 1 c.reshape(3, 4)
```

```
Out[104]: array([[ 1,  5,  9,  7],
                 [ 6,  3,  4,  7],
                 [ 5,  1,  5, 12]])
```

```
In [105]: 1 c.reshape(4, 3)
```

```
Out[105]: array([[ 1,  5,  9],
                 [ 7,  6,  3],
                 [ 4,  7,  5],
                 [ 1,  5, 12]])
```

```
In [106]: 1 c.reshape(6, 2)
```

```
Out[106]: array([[ 1,  5],
                 [ 9,  7],
                 [ 6,  3],
                 [ 4,  7],
                 [ 5,  1],
                 [ 5, 12]])
```

```
In [107]: 1 c.reshape(2,3,2 )
```

```
Out[107]: array([[[ 1,  5],
                 [ 9,  7],
                 [ 6,  3]],

                 [[ 4,  7],
                 [ 5,  1],
                 [ 5, 12]]])
```

Row × Column × Depth = Size

```
In [108]: 1 c = c.reshape(4,3,1 )
          2 c
```

```
Out[108]: array([[[ 1],
                 [ 5],
                 [ 9]],

                 [[ 7],
                 [ 6],
                 [ 3]],

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

                 [[ 1],
                 [ 5],
                 [12]])])
```

- The **flatten** function returns a 1D array containing all the elements of the original array.

```
In [109]: 1 c.flatten()
```

```
Out[109]: array([ 1,  5,  9,  7,  6,  3,  4,  7,  5,  1,  5, 12])
```

- The **ravel** function returns a 1D array containing all the elements of the original array, but it provides a **view** of the original array (instead of a copy), meaning that changes made to the raveled array will affect the original array.

```
In [110]: 1 np.ravel(c)
```

```
Out[110]: array([ 1,  5,  9,  7,  6,  3,  4,  7,  5,  1,  5, 12])
```

Feature	flatten()	ravel()
Returns	A copy of the original array	A reference/view of the original array
Effect of Modification	No effect on the original array	Modifies the original array
Memory Usage	Occupies additional memory (slower)	No additional memory usage (faster)
Type	Method of ndarray	Function in NumPy library

So, if you need to modify the flattened array and reflect those changes in the original, **ravel()** is better. If you need a copy to keep the original array unaffected, use **flatten()**.

```
In [ ]: 1
```

- The **random.choice** function returns a random element from a given array or list. It can also randomly select multiple elements, with or without replacement.

```
In [402]: 1 b
```

```
Out[402]: array([ 5,  9,  5,  4,  7,  1,  7,  1, 12,  6,  5,  3])
```

```
In [403]: 1 random.choice(b)
```

```
Out[403]: 12
```

- The **random.shuffle** function randomly reorders the elements of a list in-place. It modifies the original list directly.
- **b.copy()** : When you need to preserve the original list `b` and avoid modifying it, you can create a copy using `.copy()`.

```
In [404]: 1 random.shuffle(b)
          2 b
```

```
Out[404]: array([ 5,  7,  7,  6, 12,  1,  1,  4,  5,  5,  9,  3])
```

```
In [410]: 1 z = b.copy()
          2
          3 random.shuffle(z)
          4
          5 print("Shuffled list:")
          6 print(z)
          7
          8 print("\nOriginal list:")
          9 print(b)
```

Shuffled list:
[3 4 1 5 6 9 5 7 7 5 1 12]

Original list:
[5 7 7 6 12 1 1 4 5 5 9 3]

```
In [412]: 1 c = np.arange(18)
          2 c
```

Out[412]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17])

```
In [413]: 1 c = c.reshape(3,6)
          2 c
```

Out[413]: array([[0, 1, 2, 3, 4, 5],
 [6, 7, 8, 9, 10, 11],
 [12, 13, 14, 15, 16, 17]])

```
In [414]: 1 c = np.arange(18).reshape(3,6)
          2 c
```

Out[414]: array([[0, 1, 2, 3, 4, 5],
 [6, 7, 8, 9, 10, 11],
 [12, 13, 14, 15, 16, 17]])

```
In [ ]: 1 arr = np.arange(100).reshape(10, 2, 5)
        2 print(arr)
```

```
[[[ 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]]

  [[60 61 62 63 64]
   [65 66 67 68 69]]

  [[70 71 72 73 74]
   [75 76 77 78 79]]

  [[80 81 82 83 84]
   [85 86 87 88 89]]

  [[90 91 92 93 94]
   [95 96 97 98 99]]]
```

Breakdown:

- 10 is the number of matrices (or depth).
- 2 is the number of rows in each matrix.
- 5 is the number of columns in each row.

```
In [418]: 1 print(arr.shape)
```

```
(10, 2, 5)
```

Stacking arrays in NumPy refers to the process of joining two or more arrays along a specific axis. There are several functions available in NumPy to stack arrays:

- **np.vstack()** : Stacks arrays vertically (along rows).

```
In [423]: 1 arr1 = np.array([1, 2, 3])
          2 arr2 = np.array([4, 5, 6])
          3
          4 result = np.vstack((arr1, arr2))
          5 print(result)
          6
```

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

- **np.hstack()** : Stacks arrays horizontally (along columns).

```
In [424]: 1 arr1 = np.array([1, 2, 3])
          2 arr2 = np.array([4, 5, 6])
          3
          4 result = np.hstack((arr1, arr2))
          5 print(result)
          6
```

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

- **np.dstack()** : Stacks arrays along the third axis (depth).

```
In [425]: 1 arr1 = np.array([[1, 2], [3, 4]])
          2 arr2 = np.array([[5, 6], [7, 8]])
          3
          4 result = np.dstack((arr1, arr2))
          5 print(result)
          6
```

```
[[[1 5]
   [2 6]]
```

```
 [[3 7]
   [4 8]]]
```

```
In [426]: 1 arr1 = np.array([1, 2, 3])
          2 arr2 = np.array([4, 5, 6])
          3
          4 result = np.concatenate((arr1, arr2), axis=0)
          5 print(result)
          6
```

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



```
In [429]: 1 arr1 = np.array([1, 2, 3])
          2 arr2 = np.array([4, 5, 6])
          3
          4 result = np.concatenate((arr1, arr2), axis=1)
          5 print(result)
          6
```

AxisError Traceback (most recent call last)

Cell In[429], line 4

```
      1 arr1 = np.array([1, 2, 3])
      2 arr2 = np.array([4, 5, 6])
----> 4 result = np.concatenate((arr1, arr2), axis=1)
      5 print(result)
```

AxisError: axis 1 is out of bounds for array of dimension 1

```
In [428]: 1 arr1 = np.array([[1, 2], [3, 4]])
          2 arr2 = np.array([[5, 6], [7, 8]])
          3
          4 result = np.concatenate((arr1, arr2), axis=1)
          5 print(result)
          6
```

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

▼ Explanation:

- **1D arrays:** Concatenating along **axis 0** (the only axis for 1D arrays) simply combines the arrays into a longer 1D array.
- **2D arrays:** Concatenating along **axis 1** adds the columns of `arr2` to the columns of `arr1`.

```
In [ ]:
```

- **reduce** : Performs a cumulative operation (e.g., sum or multiplication) on the array.

```
In [442]: 1 import numpy as np
          2 m = np.array([1, 2, 3, 4])
          3 result = np.add.reduce(m)
          4 print(result) # (1+2+3+4)
          5
```

10

```
In [443]: 1 result = np.multiply.reduce(m)
          2 print(result) # (1*2*3*4)
          3
```

24

- **trace** : Computes the trace of a matrix (sum of diagonal elements).

```
In [452]: 1 q=np.arange(1,17).reshape(4,4)
          2 q
```

```
Out[452]: array([[ 1,  2,  3,  4],
                  [ 5,  6,  7,  8],
                  [ 9, 10, 11, 12],
                  [13, 14, 15, 16]])
```

```
In [453]: 1 trace_value = np.trace(q)
          2 print(trace_value)
```

```
34
```

- **det** : Computes the determinant of a matrix.

```
In [454]: 1 det_value = np.linalg.det(q)
          2 print(det_value)
          3
```

```
0.0
```

- **split** : Splits an array into sub-arrays at specified indices.

```
In [440]: 1 a = np.array([1, 2, 3, 4, 5, 6])
          2 split_result = np.split(a, (0, 4))
          3 print(split_result)
```

```
[array([], dtype=int32), array([1, 2, 3, 4]), array([5, 6])]
```

```
In [441]: 1 len(split_result)
          2
```

```
Out[441]: 3
```

```
In [462]: 1 a1, a2, a3 = split(a, (0, 4))
          2 print(a1)
          3 print(a2)
          4 print(a3)
```

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

▼ Explanation:




- `np.split(a, (0, 4))` divides the array `a` into parts based on the indices you provided. The result will have 3 parts:
 - One part before index 0 (which is an empty array).
 - One part from index 0 to index 4.
 - One part from index 4 to the end.

In []:

1

▼ Contact Me

Feel free to reach out to me anytime!

-  **Whatsapp:** 01008141749
-  **LinkedIn:** <https://www.linkedin.com/in/ahmed-elsany-0a588a223/>
(<https://www.linkedin.com/in/ahmed-elsany-0a588a223/>)
-  **Facebook:** <https://www.facebook.com/profile.php?id=100009780577339>
(<https://www.facebook.com/profile.php?id=100009780577339>)

▼ Don't Stop , It's Just Beginning