

NUMPY CRASH COURSE

Installation of NumPy

```
In [12]: pip install numpy
```

```
Requirement already satisfied: numpy in c:\users\bhadr\anaconda3\lib\site-packages (2.1.3)
```

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

1. Importing Data

Description:-

NumPy can directly read and write plain text or CSV-formatted data files. This is extremely useful when you want to load numeric datasets quickly without using Pandas.

```
In [3]: import numpy as np
```

LOADTXT

Reads simple numeric text files where all rows have equal columns.

Parameters:-

filename → Path to the text file.

delimiter → Character separating values (e.g., ',' or '\t').

Optional arguments: skiprows, usecols, dtype.

Use-Case: Import clean, numeric-only datasets.

```
In [24]: import numpy as np  
data = np.loadtxt('book.csv', delimiter=',', encoding='utf-8-sig')  
data
```

```
Out[24]: array([[ 1.,  10.,  50.],
   [ 2.,  20.,  40.],
   [ 3.,  30.,  70.],
   [ 4.,  40.,  90.],
   [ 5.,  50.,  65.],
   [ 6.,  60.,  84.],
   [ 7.,  70.,  50.],
   [ 8.,  80., 520.],
   [ 9.,  90.,  52.],
   [10., 100.,  78.]])
```

genfromtxt

Like loadtxt() but more flexible — handles missing values, headers, and mixed data types.

Use-Case: Real-world CSVs that may contain empty cells or headers.

```
In [32]: import numpy as np

data = np.genfromtxt(
    'book.csv',                      # your CSV file
    delimiter=',',                   # comma-separated
    skip_header=0,                  # skip the column names (if present)
    filling_values=0,               # replace blanks with 0
    encoding='utf-8-sig'            # handles the BOM (ï»¿)
)

print(data[:5]) ## its showing top 5 rows
print(data.shape) ## its describe the total rows and columns
```



```
[[ 1. 10. 50.]
 [ 2. 20. 40.]
 [ 3. 30. 70.]
 [ 4. 40. 90.]
 [ 5. 50. 65.]]
(10, 3)
```

savetxt:- np.savetxt(filename, array, delimiter=',')

Writes a NumPy array back to a text or CSV file.

Use-Case: Export processed or cleaned data.**

```
In [31]: a=np.savetxt("book1",data,delimiter=',')
print("file saved sucessfully")
```

```
file saved sucessfully
```

2 Creating Arrays

Description

NumPy arrays (ndarray) are the foundation of scientific computing. They're homogeneous (all elements have the same type) and stored efficiently in memory.

Types of arrays

One-Dimensional (1-D) Arrays Description: The most common and fundamental type, these are simple arrays (like a mathematical vector). They have a single row of data.

Example: [1, 2, 3, 4,]

Shape: (N,) (e.g., (4,))

ndim (Number of Dimensions): 1

Two-Dimensional (2-D) Arrays Description: Arrays that have rows and columns, like a mathematical matrix or a spreadsheet.

Example:

[[1, 2, 3],

[4, 5, 6]]

Shape: (R, C) (R = rows, C = columns; e.g., (2, 3))

ndim (Number of Dimensions): 2

Three-Dimensional (3-D) Arrays

Description: Arrays that contain 2-D arrays (matrices) as their elements. These are often used to represent concepts like a cube or a collection of matrices (like color images, where the three dimensions might represent height, width, and color channels).

Example:

[[[1, 2], [3, 4]],

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

Shape: (D, R, C) (D = depth/layers, R = rows, C = columns; e.g., (2, 2, 2))

ndim (Number of Dimensions): 3

N-Dimensional (N-D) Arrays

Description: The general term for arrays with any number of dimensions greater than 3. While 0-D, 1-D, 2-D, and 3-D are specific cases, NumPy's core power is its ability to handle arrays with N dimensions, hence the name ndarray (N-dimensional array).

Example: A 4-D array could represent a time-series of color images (Time, Height, Width, Channels).

Shape: (D_1, D_2, \dots, D_N) ndim (Number of Dimensions): N

creat in array in dimension in simple remind how many brackets you can generate in array creation the number of square brackets is equal to dimensions of array

```
In [44]: ## creating 1d array
arr1d = np.array([1, 2, 3, 4])
print("array dimension:", arr1d.ndim)
print("1-d array", arr1d)

## creating 2d array
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print("array dimension:", "arr2d.ndim")
print("2-d array", arr2d)

## creating 3d array
arr3d=np.array([[[1,2,3,4],[5,6,7,8],[10,11,12,13],[14,15,16,17]]])
print("array dimension:", arr3d.ndim)
print("3-d array", arr3d)
```

```
array dimension 1
1-d array [1 2 3 4]
array dimension: arr2d.ndim
2-d array [[1 2 3]
 [4 5 6]]
array dimension: 3
3-d array [[[ 1  2  3  4]
 [ 5  6  7  8]
 [10 11 12 13]
 [14 15 16 17]]]
```

np.zeros(shape)

Creates an array filled entirely with zeros.

Use-Case: Useful for initialization or placeholders.

```
In [64]: zeros = np.zeros((2, 4))
print("zeros in 2dd array")
print(zeros)
```

```
zero=np.zeros(((2,3,4)))
print("zeros in 3d array")
print(zero)
```

```
zeros in 2dd array
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
zeros in 3d array
[[[0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]]
```

np.ones(shape)

Creates an array where every element is one.

```
In [57]: ones = np.ones((2, 4))
print("ones in 2dd array")
print(ones)

one=np.ones(((2,3,4)))
print("ones in 3d array")
print(one)
```

```
ones in 2dd array
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]]
ones in 3d array
[[[1. 1. 1. 1.]
   [1. 1. 1. 1.]
   [1. 1. 1. 1.]]
```

np.eye(n)

Generates an identity matrix (diagonal of 1s, rest 0s).

Common in linear algebra and matrix transformations.

```
In [63]: arr=np.eye(5,5)
arr
```

```
Out[63]: array([[1., 0., 0., 0., 0.],
   [0., 1., 0., 0., 0.],
   [0., 0., 1., 0., 0.],
   [0., 0., 0., 1., 0.],
   [0., 0., 0., 0., 1.]])
```

np.arange(start, stop, step)

Returns evenly spaced values within a range — like Python's range() but as an array.

```
In [69]: arr1 = np.arange(0, 100, 2) ## arrange numbers in 0 to 100 in range of 2.
arr1
```

```
Out[69]: 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])
```

np.linspace(start, stop, num):

Creates a sequence of num evenly spaced points between two limits (inclusive).

Use-Case: Generating data for plots or numerical simulations.

```
In [72]: line = np.linspace(0, 1, 5) ## 0 to 1 is the interval 5 parts equally distri
print(line)

[0.  0.25 0.5  0.75 1. ]
```

np.full(shape, value)

Creates an array filled with a constant value.

```
In [74]: filled = np.full((5, 5), 9)
print(filled)

[[9 9 9 9 9]
 [9 9 9 9 9]
 [9 9 9 9 9]
 [9 9 9 9 9]
 [9 9 9 9 9]]
```

3.NumPy Random Number Functions – rand(), randint(), randn()

1. numpy.random.rand()

****Description:**** Generates random floating-point numbers between 0 and 1 from a uniform distribution.

Syntax:

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

Parameters:

d0, d1, ..., dn: Dimensions of the output array.

Use Case:

Used in simulations, normalization, or initializing random weights in ML models.

```
In [80]: import numpy as np

# Single random number
print("Single random number:", np.random.rand())

# 1D array of 5 random numbers
print("1D array of 5 random numbe:", np.random.rand(5))

# 2D array (3x2)
print('2D array (3x2):')
print(np.random.rand(3, 2))
```

Single random number: 0.6070003939456445
1D array of 5 random numbe: [0.83969391 0.96053244 0.6464185 0.27927977 0.7797173]
2D array (3x2):
[[0.9554153 0.64026255]
[0.23262798 0.56987242]
[0.64775771 0.29204844]]

2. numpy.random.randn()

Description:

Generates random floating-point numbers following a standard normal distribution.

Mean = 0, Standard Deviation = 1.

Values can be negative or positive.

□ **Syntax:** np.random.randn(d0, d1, ..., dn)

```
In [94]: import numpy as np

# Single random number (mean 0, std 1)
print("Single random number (mean 0, std 1) :- ")
print(np.random.randn())
```

```
# 1D array of 5 normally distributed numbers
print("1D array of 5 normally distributed numbers :- ")
print(np.random.randn(5))

# 2D array (2x3)
print("2D array (2x3):- ")
print(np.random.randn(2, 3))
```

```
Single random number (mean 0, std 1) :-
1.3923999438401062
1D array of 5 normally distributed numbers :-
[ 0.48750447 -0.26319803 -0.04454444 -1.30185539 -0.18995846]
2D array (2x3):-
[[ 1.24205904 -0.01575455  1.96869388]
 [-0.19886616 -0.48428737  0.47224801]]
```

3. numpy.random.randint()

Description:

Generates random integer numbers within a specified range.

Follows a discrete uniform distribution.

Can generate either a single number or an array of integers.

Syntax: np.random.randint(low, high=None, size=None, dtype=int)

low → lower bound (inclusive)

high → upper bound (exclusive)

size → shape of output array

dtype → type of integers (default = int)

Use Case:

Used for sampling integer values — e.g., dice rolls, random IDs, categorical sampling.

```
In [90]: import numpy as np

# Random integer between 0 and 10
print('Random integer between 0 and 10 :',np.random.randint(10))

# Random integer between 100 and 1000 to taken by 10 random numbers
print("Random integer between 100 and 1000 to taken by 10 random numbers:")
print(np.random.randint(100,1000,10))

# Random integer between 5 and 15
print('Random integer between 5 and 15 :',np.random.randint(5, 15))
```

```
# 2x3 matrix of random integers between 1 and 100
print("2x3 matrix of random integers between 1 and 100:")
print(np.random.randint(1, 100, (2, 3)))
```

```
Random integer between 0 and 10 : 9
Random integer between 100 and 1000 to taken by 10 random numbers:
[933 524 495 547 863 460 351 901 546 891]
Random integer between 5 and 15 : 9
2x3 matrix of random integers between 1 and 100:
[[47 94 21]
 [39  1 12]]
```

4. Inspecting Array Properties

Description

Inspecting helps you understand an array's structure, which is crucial before manipulation or model training.

I. arr.shape

Returns tuple → (rows, columns) for 2D arrays.

```
In [96]: arr = np.array([[1,2,3],[4,5,6]])
print(arr.shape)
```

(2, 3)

II. arr.size

Number of total elements.

```
In [97]: print(arr.size)
```

6

III. arr.dtype

Shows element type (e.g., int32, float64).

```
In [98]: print(arr.dtype)
```

int64

IV. arr.astype(dtype)

Converts elements to another data type.

```
In [102... arr_f = arr.astype(float)
```

```
print(arr_f)
```

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

V. arr.tolist()

Converts NumPy array back to a Python list.

```
In [105... lst = arr.tolist()  
print(lst)
```

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

VI. np.info(object)

Displays function or object documentation directly inside Jupyter.

```
In [110... np.info(np.zeros) ## its give the full information in the objects.
```

```
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, optional
    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.

.. versionadded:: 1.20.0

Returns
-----
out : ndarray
    Array of zeros with the given shape, dtype, and order.

See Also
-----
zeros_like : Return an array of zeros with shape and type of input.
empty : Return a new uninitialized array.
ones : Return a new array setting values to one.
full : Return a new array of given shape filled with value.

Examples
-----
>>> import numpy as np
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])

>>> np.zeros((5,), dtype=int)
array([0, 0, 0, 0, 0])

>>> np.zeros((2, 1))
array([[ 0.],
       [ 0.]))

>>> s = (2,2)
>>> np.zeros(s)
array([[ 0.,  0.],
       [ 0.,  0.]))

>>> np.zeros((2,), dtype=[('x', 'i4'), ('y', 'i4')]) # custom dtype
```

```
array([(0, 0), (0, 0)],  
      dtype=[('x', '<i4'), ('y', '<i4')])
```