🧮 NumPy Essentials for Data Science and AI

Prepared by: **Kudum Veerabhadraiah**

>

NumPy Complete Refere — Index

No.	Topic	Description
1	**Importing Data**	Load and save text or CSV data using `np.loadtxt()`, `np.genfromtxt()`, and `np.savetxt()`.
2	**Creating Arrays**	Create arrays using `np.array()`, `np.zeros()`, `np.ones()`, `np.arange()`, `np.linspace()`, `np.random.rand()`, etc.
3	**Inspecting Properties**	Understand array attributes: `shape`, `dtype`, `size`, and type conversion using `astype()` or `tolist()`.
4	**Copying, Sorting & Reshaping**	Perform `copy()`, `view()`, `reshape()`, `resize()`, and `flatten()` operations efficiently.
5	**Adding & Removing Elements**	Dynamically modify arrays with `np.append()`, `np.insert()`, and `np.delete()`.
6	**Combining & Splitting**	Merge and divide arrays using `np.concatenate()`, `np.split()`, and `np.hsplit()`.
7	**Indexing & Slicing**	Access elements and subarrays efficiently using slicing and conditional selection.
8	**Fancy Indexing**	Select array elements using integer arrays or boolean masks for advanced data extraction.
9	**Mathematical Operations**	Perform arithmetic operations and broadcasting directly on NumPy arrays.
10	**Vector Math Operations**	Apply element-wise math functions such as `np.add()`, `np.multiply()`, `np.sqrt()`, and rounding.
11	**Statistics**	Compute descriptive metrics: `np.mean()`, `np.sum()`, `np.var()`, `np.std()`, `np.corrcoef()`.

Installation of NumPy

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

1. Importing Data

Description

NumPy provides powerful I/O functions to **read and write data** efficiently from text and CSV files.
These are widely used for **loading datasets**, **handling missing values**, and **saving results** in data analysis workflows.

Common Functions

Function	Description	Example Usage
`np.loadtxt()`	Loads data from a text file (fast, simple)	`np.loadtxt('data.txt', delimiter=',')`
`np.genfromtxt()`	Loads data with **missing value handling**	`np.genfromtxt('data.csv', delimiter=',', filling_values=0)`
`np.savetxt()`	Saves an array to a text or CSV file	`np.savetxt('output.csv', arr, delimiter=',')`

In [82]:
```
import numpy as np
```

I. 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.]])
```

II. 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)
```

III. 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 provides multiple methods to **create arrays** — from Python lists, sequences, or random data.
These arrays form the foundation for all numerical and scientific computations.

Common Functions

Function	Description	Example Usage
`np.array()`	Creates an array from a Python list or tuple	`np.array([1, 2, 3])`
`np.zeros()`	Creates an array filled with zeros	`np.zeros((2, 3))`
`np.ones()`	Creates an array filled with ones	`np.ones((3, 2))`
`np.eye()`	Creates an identity matrix	`np.eye(3)`
`np.arange()`	Creates evenly spaced values (like Python's `range()`)	`np.arange(0, 10, 2)`
`np.linspace()`	Creates evenly spaced numbers over a specified interval	`np.linspace(0, 1, 5)`
`np.full()`	Creates an array filled with a specific constant value	`np.full((2, 2), 7)`
`np.random.rand()`	Creates an array with random floats (0 to 1)	`np.random.rand(2, 3)`
`np.random.randint()`	Creates an array with random integers in a range	`np.random.randint(1, 10, (2, 3))`

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

creatin array in dimension in simple remind how many brackets you can generate in array creation the number of squre 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.]]

 [[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.]]

 [[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 distributed
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]]
```

->.NumPy Random Number Functions — rand(), randint(), randn()<-

I. 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]]
```

II. 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]]
```

III. 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]:
```python
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]]
```

3. Inspecting Properties

Description

NumPy provides several attributes and functions to **inspect, analyze, and convert array properties**.
These are essential to understand the **structure, data type, and memory layout** of arrays.

Common Functions & Attributes

Function / Attribute	Description	Example Usage
`shape`	Returns dimensions of the array (rows, columns)	`a.shape`
`dtype`	Returns the data type of array elements	`a.dtype`
`size`	Returns the total number of elements	`a.size`
`astype()`	Converts elements to a specified type	`a.astype(float)`
`tolist()`	Converts NumPy array to a regular Python list	`a.tolist()`
`np.info()`	Displays detailed information about a NumPy function or array	`np.info(np.mean)`

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 [3]:
```
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')])
```

4. Copying, Sorting & Reshaping

Description

NumPy provides various tools to **copy**, **sort**, and **reshape** arrays for data manipulation and analysis.
These operations are essential for **data organization**, **dimensional transformations**, and **memory control**.

Common Functions

Function	Description	Example Usage
`copy()`	Creates a deep copy of an array (independent of original)	`b = a.copy()`
`view()`	Creates a shallow copy (shares data with original)	`b = a.view()`
`sort()`	Sorts array elements along a specified axis	`np.sort(a)`
`flatten()`	Converts multi-dimensional array into 1D	`a.flatten()`
`T`	Transposes array (rows ↔ columns)	`a.T`
`reshape()`	Changes the shape without changing data	`a.reshape(2, 3)`
`resize()`	Changes shape and size (modifies original array)	`a.resize(3, 2)`

I. np.copy(arr)

Creates a deep copy — new memory space.

In [5]:
```python
a = np.array([1,2,3])
b = np.copy(a)
b[0] = 99
print(a)
print(b) ### its affecting only dupicated data not effecyed to origiinal data
```

```
[1 2 3]
[99  2  3]
```

II. arr.view()

Creates a shallow copy (changes reflect on both).

In [8]:
```python
v = a.view()
v[1] = 10
```

```
print(v)
print(a)  # Affected [1 10 3] in both original and duplicate data.
```

```
[ 1 10  3]
[ 1 10  3]
```

III. arr.sort(axis=0 or 1)

Sorts array in-place along an axis.

In [17]:
```
m = np.array([[3,1,2],[9,7,8]])
m.sort(axis=1)
print(m)
```

```
[[1 2 3]
 [7 8 9]]
```

IV. arr.flatten()

Returns a 1D copy of any array (no dimension nesting).

In [19]:
```
flat = m.flatten()
print(flat)
```

```
[1 2 3 7 8 9]
```

V. arr.reshape(rows, cols)

Changes the dimension but keeps total element count constant.

In [20]:
```
reshaped = np.arange(12).reshape(3,4)
print(reshaped)
```

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

VI. arr.T

Returns transpose — flips rows and columns.

In [22]:
```
print(reshaped.T)
```

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

VII. arr.resize(new_shape)

Modifies the original array's shape in-place (fills with zeros if needed).

In [24]:
```
reshaped.resize((2,6))
print(reshaped)
```

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

5. Adding & Removing Elements

Description

NumPy provides functions to **dynamically modify arrays** by adding, inserting, or deleting elements.
These operations create **new arrays** since NumPy arrays have **fixed size** once created.

Common Functions

Function	Description	Example Usage
`np.append()`	Adds elements to the end of an array	`np.append(a, [7, 8])`
`np.insert()`	Inserts values at a specific position	`np.insert(a, 1, 99)`
`np.delete()`	Deletes elements by index	`np.delete(a, [0, 2])`

I. np.append(arr, values, axis=None)

Adds new values at the end (creates new array).

append() always flattens arrays unless axis is specified.

In [25]:
```
arr = np.array([1,2,3])
arr2 = np.append(arr, [4,5])
print(arr2)
```

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

II. np.insert(arr, index, values, axis=None)

Inserts elements before the specified index.

In [26]:
```
arr = np.array([10,20,30])
inserted = np.insert(arr, 1, 15)
print(inserted)
```

```
[10 15 20 30]
```

III. np.delete(arr, index, axis=None)

Deletes elements along a given axis.

In numpy.delete(), for a two-dimensional array:

If axis=0, you delete rows. The index specified in the obj parameter refers to which row(s) to remove.

If axis=1, you delete columns. The index specified in the obj parameter refers to which column(s) to remove.

In [32]:
```
arr2d = np.array([[1,2,3],[4,5,6]])
deleted = np.delete(arr2d, 1, axis=0)
delete=np.delete(arr2d,1,axis=1)
print("deleted in columns using axis 1:")
print(delete)
print("deleted in columns using axis 1:",deleted)
```

```
deleted in columns using axis 1:
[[1 3]
 [4 6]]
deleted in columns using axis 1: [[1 2 3]]
```

6. Combining & Splitting Arrays

Description

NumPy allows you to **combine multiple arrays** into one or **split a large array** into smaller subsets.
These operations are essential for **data preprocessing**, **batch processing**, or **merging datasets**.

Common Functions

Function	Description	Example Usage
`np.concatenate()`	Joins two or more arrays along an axis	`np.concatenate((a, b), axis=0)`
`np.vstack()`	Stacks arrays vertically (row-wise)	`np.vstack((a, b))`
`np.hstack()`	Stacks arrays horizontally (column-wise)	`np.hstack((a, b))`
`np.split()`	Splits array into multiple sub-arrays	`np.split(a, 2)`

Function	Description	Example Usage
`np.hsplit()`	Splits array horizontally (by columns)	`np.hsplit(a, 2)`
`np.vsplit()`	Splits array vertically (by rows)	`np.vsplit(a, 2)`

I. np.concatenate((arr1, arr2), axis=0 or 1)

Merges arrays along rows (axis=0) or columns (axis=1).

In [35]:
```
a = np.array([[1,2],[3,4]])
b = np.array([[5,6]])
combined = np.concatenate((a,b), axis=0)
print(combined)
```

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

II. np.split(arr, num_splits)

Splits an array into multiple subarrays of equal size.

In [39]:
```
data = np.arange(10)
parts = np.split(data, 5)
print(parts)
```

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

III. np.hsplit(arr, num)

Splits horizontally (by columns).

In [80]:
```
arr1=np.arange(16).reshape(2,8)
left,right=np.hsplit(arr1,2)
print(left)
print(right)
```

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

In [81]:
```
# Create two 2x2 arrays
a = np.array([[1, 2],
              [3, 4]])

b = np.array([[5, 6],
              [7, 8]])
```

```python
# 1 Combine arrays vertically (along rows)
vertical = np.concatenate((a, b), axis=0)
print("Vertical Stack:\n", vertical)

# 2 Combine arrays horizontally (along columns)
horizontal = np.concatenate((a, b), axis=1)
print("\nHorizontal Stack:\n", horizontal)

# 3 Split array into equal parts
split_arr = np.split(vertical, 2)
print("\nSplit Arrays:")
for part in split_arr:
    print(part)

# 4 Horizontal split
h_split = np.hsplit(horizontal, 2)
print("\nHorizontal Split:")
for part in h_split:
    print(part)
```

```
Vertical Stack:
 [[1 2]
 [3 4]
 [5 6]
 [7 8]]

Horizontal Stack:
 [[1 2 5 6]
 [3 4 7 8]]

Split Arrays:
[[1 2]
 [3 4]]
[[5 6]
 [7 8]]

Horizontal Split:
[[1 2]
 [3 4]]
[[5 6]
 [7 8]]
```

7. NummPy Indexing & Slicing

📘 Overview

Efficiently access, extract, and manipulate specific parts of arrays — critical for **data cleaning, feature engineering, and model preparation**.

🔷 Basic Indexing

Concept	Syntax	Description	Example	Output
Access 1D element	`'arr[i]'`	Selects i-th element	`'arr[2]'`	Single value
Access 2D element	`'arr[i, j]'`	Selects element at row *i*, column *j*	`'arr[1, 2]'`	Single value
Negative index	`'arr[-1]'`	Selects last element	`'arr[-1]'`	Last element

✅ **Use Case:** Accessing specific data points in a table-like dataset.

In [51]:
```
arr = np.array([
    [10, 20, 30],
    [40, 50, 60],
    [70, 80, 90]
])

print("Original Array:\n", arr)
print("Element at [0,1]:", arr[0, 1])
print("Element at [2,2]:", arr[2, 2])
print("Last element:", arr[-1, -1])
```

```
Original Array:
 [[10 20 30]
 [40 50 60]
 [70 80 90]]
Element at [0,1]: 20
Element at [2,2]: 90
Last element: 90
```

I. Basic Slicing (Rows, Columns, Subarrays)

Operation	Syntax	Meaning	Example Output
Row range	`'arr[0:2, :]'`	Rows 0–1, all columns	`[[10 20 30], [40 50 60]]`
Column range	`'arr[:, 1:3]'`	All rows, columns 1–2	`[[20 30], [50 60], [80 90]]`
Subarray	`'arr[0:2, 0:2]'`	Rows 0–1, columns 0–1	`[[10 20], [40 50]]`
Step slicing	`'arr[::2, ::2]'`	Every 2nd row and 2nd column	`[[10 30], [70 90]]`

✅ **Use Case:** Extract feature subsets, first N rows, or specific columns.

In [52]:
```
print("First 2 rows:\n", arr[0:2, :])
print("Last 2 columns:\n", arr[:, 1:3])
```

```python
print("Top-left 2x2 block:\n", arr[0:2, 0:2])
print("Every 2nd element:\n", arr[::2, ::2])
```

```
First 2 rows:
 [[10 20 30]
 [40 50 60]]
Last 2 columns:
 [[20 30]
 [50 60]
 [80 90]]
Top-left 2x2 block:
 [[10 20]
 [40 50]]
Every 2nd element:
 [[10 30]
 [70 90]]
```

II. Negative Indexing

Type	Syntax	Description	Example Output
Last row	`'arr[-1]'`	Selects last row	`[70 80 90]`
Last column	`'arr[:, -1]'`	Selects last column	`[30 60 90]`
Bottom-right block	`'arr[-2:, -2:]'`	Last 2 rows & columns	`[[50 60], [80 90]]`

✅ **Use Case:** Quick access to recent or trailing data (e.g., last week, last month).

In [61]:
```python
print("original array:",arr)
print("Last row:\n", arr[-1])
print("Last column:\n", arr[:, -1])
print("Bottom-right 2x2 block:\n", arr[-2:, -2:])
```

```
original array: [[10 20 30]
 [40 50 60]
 [70 80 90]]
Last row:
 [70 80 90]
Last column:
 [30 60 90]
Bottom-right 2x2 block:
 [[50 60]
 [80 90]]
```

8. Advanced Indexing (Fancy Indexing)

Use lists or arrays of indices to select arbitrary elements.

Concept	Syntax	Description	Example Output
Select specific rows	`'arr[[0, 2]]'`	Rows 0 and 2	`[[10 20 30], [70 80 90]]`

Concept	Syntax	Description	Example Output
Select specific columns	`'arr[:, [1, 2]]'`	Columns 1 and 2	`[[20 30], [50 60], [80 90]]`
Select custom subarray	`'arr[np.ix_([0,2], [1,2])]'`	Rows 0 & 2, Columns 1 & 2	`[[20 30], [80 90]]`

✅ **Use Case:** Select non-contiguous features or records — e.g., columns ['Age', 'Salary'] from selected customers.

In [62]:
```
print("original array:",arr)
rows = [0, 2]
cols = [1, 2]
print("Rows 0 & 2, Cols 1 & 2:\n", arr[np.ix_(rows, cols)])
```

```
original array: [[10 20 30]
 [40 50 60]
 [70 80 90]]
Rows 0 & 2, Cols 1 & 2:
 [[20 30]
 [80 90]]
```

A. Conditional (Boolean) Indexing

| Condition Type | Syntax | Meaning | Example Output | |----------------|---------|----------|----------------| | Simple condition | `'arr[arr > 50]'` | Elements > 50 | `[60 70 80 90]` | | Combined (AND) | `'arr[(arr >= 30) & (arr <= 80)]'` | 30 ≤ x ≤ 80 | `[30 40 50 60 70 80]` | | Combined (OR) | `'arr[(arr < 20) | (arr > 80)]'` | x < 20 or x > 80 | `[10 90]` | | Negation | `'arr[~(arr > 50)]'` | NOT condition | `[10 20 30 40 50]` |

✅ **Use Case:** Data filtering — extract elements, rows, or features based on logical conditions.

In [64]:
```
print("original array:",arr)
print("Elements > 50:\n", arr[arr > 50])
print("Elements between 30 and 80:\n", arr[(arr >= 30) & (arr <= 80)])
print("Elements < 20 or > 80:\n", arr[(arr < 20) | (arr > 80)])
print("Negation (NOT > 50):\n", arr[~(arr > 50)])
```

```
original array: [[999 999  30]
 [999 999  60]
 [ 70  80  90]]
Elements > 50:
 [999 999 999 999  60  70  80  90]
Elements between 30 and 80:
 [30 60 70 80]
Elements < 20 or > 80:
 [999 999 999 999  90]
Negation (NOT > 50):
 [30]
```

B. Extracting Rows and Columns

Action	Syntax	Output Example
Select single row	`'arr[1, :]'`	`[40 50 60]`
Select single column	`'arr[:, 2]'`	`[30 60 90]`
Select multiple rows/columns	`'arr[np.ix_([0, 2], [1, 2])]'`	`[[20 30], [80 90]]`

✅ **Use Case:** Retrieve specific rows/columns when preprocessing datasets for ML models.

In [65]:
```
print("original array:",arr)
print("2nd row:\n", arr[1, :])
print("3rd column:\n", arr[:, 2])
print("Rows 0 & 2, Columns 1 & 2:\n", arr[np.ix_([0, 2], [1, 2])])
```

```
original array: [[999 999  30]
 [999 999  60]
 [ 70  80  90]]
2nd row:
 [999 999  60]
3rd column:
 [30 60 90]
Rows 0 & 2, Columns 1 & 2:
 [[999  30]
 [ 80  90]]
```

C. Step and Reversed Slicing

Operation	Syntax	Description	Output
Every 2nd row	`'arr[::2, :]'`	Selects alternate rows	`[[10 20 30], [70 80 90]]`
Reverse rows	`'arr[::-1, :]'`	Reverses row order	`[[70 80 90], [40 50 60], [10 20 30]]`
Reverse columns	`'arr[:, ::-1]'`	Reverses column order	`[[30 20 10], [60 50 40], [90 80 70]]`

✅ **Use Case:** Time-series reversal, data reordering, sampling.

In [66]:
```
print("original array:",arr)
print("Every 2nd row:\n", arr[::2, :])
print("Reversed rows:\n", arr[::-1, :])
print("Reversed columns:\n", arr[:, ::-1])
```

```
original array: [[999 999  30]
 [999 999  60]
 [ 70  80  90]]
Every 2nd row:
 [[999 999  30]
 [ 70  80  90]]
Reversed rows:
 [[ 70  80  90]
 [999 999  60]
 [999 999  30]]
Reversed columns:
 [[ 30 999 999]
 [ 60 999 999]
 [ 90  80  70]]
```

D. Views vs Copies (Important!)

Type	Behavior	Example	Effect on Original
Slice (view)	Shares memory	`'subset = arr[0:2, 0:2]'`	✅ Changes affect original
Fancy/boolean (copy)	Independent memory	`'subset = arr[arr > 50]'`	❌ Changes don't affect original
Explicit copy	Force new memory	`'subset = arr[0:2, 0:2].copy()'`	Safe copy

✅ **Use Case:** Use `.copy()` when extracting subsets for isolated analysis or model testing,

In [68]:
```python
subset = arr[0:2, 0:2]
subset[:] = 999
print("Subset (View):\n", subset)
print("Original after view modification:\n", arr)

# Create a copy safely
copy_subset = arr[0:2, 0:2].copy()
copy_subset[:] = 555
print("Copied subset:\n", copy_subset)
print("Original unaffected:\n", arr)
```

```
Subset (View):
 [[999 999]
 [999 999]]
Original after view modification:
 [[999 999  30]
 [999 999  60]
 [ 70  80  90]]
Copied subset:
 [[555 555]
 [555 555]]
Original unaffected:
 [[999 999  30]
 [999 999  60]
 [ 70  80  90]]
```

--> Data Science Use Cases

Scenario	Indexing Type	Example
Extract first 10 records	Row slicing	`'arr[:10, :]'`
Select last 5 rows	Negative slicing	`'arr[-5:, :]'`
Select features (columns)	Column slicing	`'arr[:, [1, 3, 5]]'`
Filter by value	Boolean indexing	`'arr[arr[:, 2] > 50000]'`
Remove outliers	Boolean mask	`'arr[arr < threshold]'`
Sampling every 5th row	Step slicing	`'arr[::5, :]'`

✅ **Use Case:** Core step in EDA, feature selection, and model training pipelines.

In [86]:
```
data = np.arange(1, 41).reshape(10, 4)
print("Dataset (10x3):\n", data)

# Extract first 5 records
print("\nFirst 5 records:\n", data[:5, :])

# Last 3 records
print("\nLast 3 records:\n", data[-3:, :])

# Select feature columns (Age & Salary)
print("\nSelect columns 0 & 2:\n", data[:, [0, 2]])

# Filter elements > 20
print("\nElements > 20:\n", data[data > 20])

# Step slicing (every 2nd record)
print("\nEvery 2nd record:\n", data[::2, :])
```

```
Dataset (10x3):
 [[ 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]]

First 5 records:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]
 [17 18 19 20]]

Last 3 records:
 [[29 30 31 32]
 [33 34 35 36]
 [37 38 39 40]]

Select columns 0 & 2:
 [[ 1  3]
 [ 5  7]
 [ 9 11]
 [13 15]
 [17 19]
 [21 23]
 [25 27]
 [29 31]
 [33 35]
 [37 39]]

Elements > 20:
 [21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40]

Every 2nd record:
 [[ 1  2  3  4]
 [ 9 10 11 12]
 [17 18 19 20]
 [25 26 27 28]
 [33 34 35 36]]
```

--> Axis Concept Reference

Axis	Meaning	Example Operation	Description
`'axis=0'`	Column-wise	`'np.sum(arr, axis=0)'`	Operates vertically (down each column)
`'axis=1'`	Row-wise	`'np.sum(arr, axis=1)'`	Operates horizontally (across each row)

🧩 **Remember:**

- `'axis=0'` → Down the rows (columns are fixed)
- `'axis=1'` → Across the columns (rows are fixed)

🔷 Quick Summary Table

Type	Purpose	Returns	Typical Use
Basic Indexing	Select element(s)	Scalar	Direct access
Slicing	Select ranges	View	Subarray extraction
Fancy Indexing	Arbitrary picks	Copy	Non-contiguous selection
Boolean Indexing	Conditional filtering	Copy	Data cleaning/filtering
Step Slicing	Skipped selection	View	Sampling
Negative Indexing	From end	View	Last rows/cols
Axis Control	Direction of operation	-	Row/column aggregation

🧠 Key Takeaways

- `'arr[start:end, :] → row selection'`
- `'arr[:, start:end] → column selection'`
- **Views** share memory → efficient but risky for accidental edits.
- **Copies** are independent → safe for analysis.
- Master `'axis'` and conditional indexing — it's the foundation for **Pandas**, **machine learning preprocessing**, and **vectorized data filtering**.

✅ **Conclusion:**
Efficient **indexing and slicing** unlock NumPy's real power — enabling you to manipulate massive datasets in milliseconds.
It's one of the top 5 must-master skills for every **data scientist** working with Python.

9.Scalar Math — Element-wise Operations in NumPy

📘 Overview

NumPy allows you to perform **fast, vectorized arithmetic operations** on arrays — without writing loops.

These operations are **element-wise**, meaning each element of one array is combined with the corresponding element of another array.

Scalar math also supports operations between arrays and constants.

Common Arithmetic Functions

Function	Description
`'np.add(a, b)'`	Element-wise addition
`'np.subtract(a, b)'`	Element-wise subtraction
`'np.multiply(a, b)'`	Element-wise multiplication
`'np.divide(a, b)'`	Element-wise division
`'np.power(a, b)'`	Element-wise exponentiation

All operations can also be written using arithmetic symbols:
`'a + b'`, `'a - b'`, `'a * b'`, `'a / b'`, `'a ** b'`

In [70]:
```python
import numpy as np

# Create two sample arrays
a = np.array([10, 20, 30, 40])
b = np.array([1, 2, 3, 4])

print("Array a:", a)
print("Array b:", b)

# Element-wise operations
print("\nAddition (a + b):", np.add(a, b))
print("Subtraction (a - b):", np.subtract(a, b))
print("Multiplication (a * b):", np.multiply(a, b))
print("Division (a / b):", np.divide(a, b))
print("Power (a ** b):", np.power(a, b))
```

```
Array a: [10 20 30 40]
Array b: [1 2 3 4]

Addition (a + b): [11 22 33 44]
Subtraction (a - b): [ 9 18 27 36]
Multiplication (a * b): [ 10  40  90 160]
Division (a / b): [10. 10. 10. 10.]
Power (a ** b): [      10      400    27000 2560000]
```

10. Vector Math Operations

Description

Perform element-wise mathematical operations using **NumPy**.
Each operation is vectorized, meaning it applies to every element of the array efficiently without loops.

Example Arrays

Variable	Definition	Example Values
`a`	First NumPy array	`[1, 2, 3]`
`b`	Second NumPy array	`[4, 5, 6]`

Operations and Examples

Operation	NumPy Function	Description	Example Code	Output
Addition	`np.add(a, b)`	Adds corresponding elements	`np.add(a, b)`	`[5 7 9]`
Multiplication	`np.multiply(a, b)`	Multiplies each element	`np.multiply(a, b)`	`[ 4 10 18]`
Square Root	`np.sqrt(a)`	Finds square root of each element	`np.sqrt(a)`	`[1. 1.4142 1.7320]`
Logarithm	`np.log(b)`	Natural log of each element	`np.log(b)`	`[1.386 1.609 1.791]`
Absolute Value	`np.abs()`	Converts negatives to positives	`np.abs([-1, -2, 3])`	`[1 2 3]`
Ceil	`np.ceil()`	Rounds up to nearest integer	`np.ceil([1.2, 2.7])`	`[2. 3.]`
Floor	`np.floor()`	Rounds down to nearest integer	`np.floor([1.2, 2.7])`	`[1. 2.]`
Round	`np.round()`	Rounds to nearest integer	`np.round([1.49, 2.51])`	`[1. 3.]`

In [72]:
```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

print("Addition:", np.add(a, b))
print("Multiplication:", np.multiply(a, b))
print("Square Root:", np.sqrt(a))
```

```python
print("Logarithm:", np.log(b))
print("Absolute:", np.abs([-1, -2, 3]))
print("Ceil:", np.ceil([1.2, 2.7]))
print("Floor:", np.floor([1.2, 2.7]))
print("Round:", np.round([1.49, 2.51]))
```

```
Addition: [5 7 9]
Multiplication: [ 4 10 18]
Square Root: [1.         1.41421356 1.73205081]
Logarithm: [1.38629436 1.60943791 1.79175947]
Absolute: [1 2 3]
Ceil: [2. 3.]
Floor: [1. 2.]
Round: [1. 3.]
```

11. Statistics in NumPy

Description

NumPy provides built-in **statistical functions** to analyze data quickly and efficiently.
These functions can compute summary statistics on **1D** and **2D arrays** with ease.

Common Statistical Functions

Function	Description	Example Usage
`np.mean()`	Calculates the average of array elements	`np.mean(a)`
`np.sum()`	Computes the sum of all elements	`np.sum(a)`
`np.min()`	Returns the minimum element	`np.min(a)`
`np.max()`	Returns the maximum element	`np.max(a)`
`np.var()`	Computes variance of array elements	`np.var(a)`
`np.std()`	Computes standard deviation	`np.std(a)`
`np.corrcoef()`	Calculates correlation coefficients between arrays	`np.corrcoef(a, b)`

Example Arrays

Variable	Definition	Example Values
`a`	1D NumPy array	`[1, 2, 3, 4, 5]`
`b`	2D NumPy array	`[[1, 2, 3], [4, 5, 6]]`

Example Code

In [73]:

```python
import numpy as np

# 1D array
a = np.array([1, 2, 3, 4, 5])

# 2D array
b = np.array([[1, 2, 3],
              [4, 5, 6]])

print("1D Array Mean:", np.mean(a))
print("1D Array Sum:", np.sum(a))
print("1D Array Min:", np.min(a))
print("1D Array Max:", np.max(a))
print("1D Array Variance:", np.var(a))
print("1D Array Std Dev:", np.std(a))

# 2D Array Operations
print("\n2D Array Mean:", np.mean(b))
print("2D Array Sum:", np.sum(b))
print("2D Array Min:", np.min(b))
print("2D Array Max:", np.max(b))
print("2D Array Variance:", np.var(b))
print("2D Array Std Dev:", np.std(b))
```

```
1D Array Mean: 3.0
1D Array Sum: 15
1D Array Min: 1
1D Array Max: 5
1D Array Variance: 2.0
1D Array Std Dev: 1.4142135623730951

2D Array Mean: 3.5
2D Array Sum: 21
2D Array Min: 1
2D Array Max: 6
2D Array Variance: 2.9166666666666665
2D Array Std Dev: 1.707825127659933
```


Kudum Veerabhadraiah

Data Science and AI Enthusiast