Importing and Installing NumPy

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

```
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages
(2.0.2)
```

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

Difference between NumPy Arrays and Python Lists

In [ ]:
```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] * 1000000

start = time.time()
for i in range(len(a)):
  a[i] = a[i]*2

print("\nTime taken for list:", time.time() - start)
```

```
Time taken for list: 2.8755266666412354
```

In [ ]:
```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] * 1000000
b = np.array(a)

start = time.time()
b = b*2
print("\nTime taken for array:", time.time() - start)
```

```
Time taken for array: 0.025333166122436523
```

In [ ]:
```
# Import the sys module
# 'sys' provides access to system-specific parameters and functions

import sys
```

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

print("List size:", sys.getsizeof(a))

print("Array size:", arr.nbytes)
```

```
List size: 104
Array size: 40
```

In [ ]:
```
a = [1, 'a', "apple", 2.1]
print(type(a))
```

```
<class 'list'>
```

In [ ]:
```
arr = np.array(a)
```

In [ ]:
```
arr
```

Out[ ]:
```
array(['1', 'a', 'apple', '2.1'], dtype='<U32')
```

Feature	NumPy Array (`np.array`)	Python List (`list`)
Data type	Homogeneous (all elements must be of the same type)	Heterogeneous (can store elements of different types)
Memory usage	More compact, stores data in a contiguous block in memory	Less efficient, stores references to objects
Speed	Much faster for numerical operations (uses C under the hood)	Slower for numerical operations (interpreted Python loops)
Functionality	Rich mathematical functions (`sum` , `mean` , `dot` , `sqrt` , etc.)	Limited built-in math operations (need `for` loops or `map`)
Vectorization	Supports vectorized operations (no explicit loops needed)	No native vectorization — you have to loop manually
Dimensionality	Can be multi-dimensional (`ndarray`)	Mostly 1D (lists of lists for 2D, which are slower and inconsistent)
Indexing	Supports advanced indexing, slicing, boolean masks	Only basic indexing and slicing
Broadcasting	Yes — can operate between different-shaped arrays automatically	No broadcasting — lengths must match for elementwise ops
Type safety	All elements automatically converted to the same type	No automatic type conversion
Dependencies	Requires NumPy library	Built into Python (no installation needed)

Dimensions and Shapes in NumPy

Dimensions in NumPy

A dimension (or axis) is basically the number of levels of indexing in the array.

Types of Dimensions

Dimension	Name	Example	`ndim` Output
0-D	Scalar	`np.array(42)` → just one value	`0`
1-D	Vector	`[1, 2, 3]`	`1`
2-D	Matrix	`[[1, 2, 3], [4, 5, 6]]`	`2`
3-D	Tensor (cube)	Shape like `(2, 2, 3)`	`3`
n-D	Higher dimensions	`(a, b, c, ...)`	`n`

Example:

In [ ]:
```
a = np.array(42)
b = np.array([1, 2, 3])
c = np.array([[1, 2], [3, 4]])
d = np.array([[[1, 2], [3, 4]],
              [[5, 6], [7, 8]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

```
0
1
2
3
```

In [ ]:
```
a = np.array([])
b = np.array([[]])
c = np.array([[[]]])
d = np.array([[[[]]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

```
1
2
3
4
```

Shape in NumPy

- Shape tells you the size of the array along each dimension.
- It's a tuple: (rows, columns, depth, ...)

Example:

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

print(arr.shape)
```

```
(2, 3)
```

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

print(arr.ndim)
print(arr.shape)
```

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

- 2 blocks (depth)
- 2 rows in each block
- 3 columns in each row

Changing Shape

You can reshape arrays without changing the data:

Example:

In [ ]:
```
# Create a 1D array with values from 0 to 5
arr = np.arange(6)

# Reshape the array into a 2D array with 2 rows and 3 columns
# reshape() does not change the original array unless reassigned
print(arr.reshape(2, 3))
```

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

- Shape → useful for checking matrix size in ML/DL.
- Dimensions → important for knowing if your array is 1D, 2D, 3D, etc.

Creating NumPy Arrays

From Python Lists & Tuples

In [ ]:
```
# From list
arr1 = np.array([1, 2, 3])
print(arr1)

# From tuple
arr2 = np.array((4, 5, 6))
```

```
print(arr2)

# 2D array from list of lists
arr3 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr3)
```

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

Using `np.zeros`

In [ ]:
```
# Create a 2D array with 3 rows and 4 columns, filled with zeros

np.zeros([3, 4])
```

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

Using `np.ones`

In [ ]:
```
# Create a 2D array with 2 rows and 3 columns, filled with ones

np.ones((2, 3))
```

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

Using `np.full`

In [ ]:
```
# Create a 1D NumPy array with 12 elements, all filled with the value 2

np.full(12, 2)
```

Out[ ]:
```
array([2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

In [ ]:
```
# Create a 2D NumPy array with shape (2, 2)
# All elements are filled with the value 7

np.full([2, 2], 7)
```

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

Using `np.arange`

Like Python's `range()`, but returns an array.

In [ ]:
```
# Create a 1D array starting from 0 up to 10, Step size is 2.

np.arange(0, 10, 2)
```

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

Using `np.linspace`

Generates evenly spaced numbers over a range.

In [ ]:
```
# Create a 1D array of 5 equally spaced numbers

np.linspace(0, 1, 5)
```

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

Using `np.logspace`

In [ ]:
```
# 3 numbers spaced evenly on a log scale from 10^1 to 10^3

np.logspace(1, 3, 3)
```

Out[ ]:
```
array([  10.,  100., 1000.])
```

Random Arrays

In [ ]:
```
# 1D array of 10 random numbers from uniform distribution [0, 1)

np.random.rand(10)
```

Out[ ]:
```
array([0.86894205, 0.59741745, 0.98357847, 0.07294397, 0.51285312,
       0.76932204, 0.06642499, 0.70856022, 0.21140528, 0.33137628])
```

In [ ]:
```
# 2x3 array of random numbers from uniform distribution [0, 1)

np.random.rand(2, 3)
```

Out[ ]:
```
array([[0.08799929, 0.10839326, 0.97525872],
       [0.68027429, 0.05231534, 0.6951883 ]])
```

In [ ]:
```
# Creates a 2x3 array with random numbers (mean=0, std=1)

np.random.randn(2, 3)
```

Out[ ]:
```
array([[ 1.82428783,  1.44718162,  0.7506156 ],
       [-0.1775636 ,  0.53058653, -0.88808626]])
```

In [ ]:
```
# Random integer between 10 and 100.

np.random.randint(10,100)
```

Out[ ]:
```
16
```

In [ ]:
```
# 3x3 array of random integers from 0 to 9

np.random.randint(0, 10, (3, 3))
```

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

Identity Matrix & Eye

In [ ]:
```
# 3x3 identity matrix (1s on diagonal, 0s elsewhere)
```

```python
np.eye(3)
```

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

In [3]:
```python
# 4x4 identity matrix (1s on diagonal, 0s elsewhere)

np.identity(3)
```

Out[3]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

difference between `np.eye()` and `np.identity()`:

Feature	`np.eye()`	`np.identity()`
Purpose	Creates a 2D array with 1s on the **main diagonal** and 0s elsewhere.	Creates a **square identity matrix** (1s on diagonal, 0s elsewhere).
Shape Control	Can create **rectangular matrices** by specifying `N` (rows) and `M` (columns).	Always creates a **square matrix** of size `n x n`.
Extra Options	Has a `k` parameter to shift the diagonal up or down.	No `k` parameter — always main diagonal only.
Syntax	`np.eye(N, M=None, k=0)`	`np.identity(n)`
Example	`np.eye(3, 4, k=1)` → 3×4 matrix with diagonal shifted up by 1.	`np.identity(4)` → always 4×4 matrix.

From Existing Data (Copy / View)

In [ ]:
```python
a = np.array([1, 2, 3])
b = np.array(a)          # Copy
c = np.asarray(a)        # View (shares memory)
```

NumPy Array Creation Methods

Method	Purpose
`np.array()`	From list/tuple
`np.zeros()`	All zeros
`np.ones()`	All ones
`np.full()`	Filled with constant
`np.arange()`	Range with step
`np.linspace()`	Even spacing
`np.eye()` / `np.identity()`	Identity matrix
`np.random.rand()`	Random uniform
`np.random.randn()`	Random normal

Method	Purpose
`np.random.randint()`	Random integers
`np.fromfunction()`	From function
`np.fromiter()`	From iterator

NumPy Data Types and Type Casting

NumPy Data Types (`dtype`)

NumPy arrays can store only one data type at a time (homogeneous). When you create an array, NumPy infers the data type automatically, but you can also specify it manually.

Common NumPy Data Types

Data type	Description	Example
`int8`, `int16`, `int32`, `int64`	Signed integers	`np.array([1, 2], dtype='int16')`
`uint8`, `uint16`, `uint32`, `uint64`	Unsigned integers	`np.array([1, 2], dtype='uint8')`
`float16`, `float32`, `float64`	Floating point	`np.array([1.5, 2.3], dtype='float32')`
`complex64`, `complex128`	Complex numbers	`np.array([1+2j], dtype='complex64')`
`bool`	Boolean values	`np.array([True, False])`
`str_`, `unicode_`	Strings	`np.array(['a', 'b'])`

Example:

In [ ]:
```
arr1 = np.array([1, 2, 3])
arr2 = np.array([1.5, 2.5, 3.5])
arr3 = np.array([1, 2, 3], dtype='float32')

print(arr1.dtype)
print(arr2.dtype)
print(arr3.dtype)
```

```
int64
float64
float32
```

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

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

In [ ]:
```
arr = np.array([1, 2, 3.1, 4, 5])
arr
```

Out[ ]:
```
array([1. , 2. , 3.1, 4. , 5. ])
```

In [ ]:
```
print(type(arr))
```

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

In [ ]:
```
lst = ["String", 1, 2, 5.6]
arr = np.array(lst)
arr
```

Out[ ]:
```
array(['String', '1', '2', '5.6'], dtype='<U32')
```

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

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

In [ ]:
```
arr.dtype
```

Out[ ]:
```
dtype('int64')
```

In [ ]:
```
arr = np.array([1, 2, 3.1, 4, 5])
arr
```

Out[ ]:
```
array([1. , 2. , 3.1, 4. , 5. ])
```

In [ ]:
```
arr.dtype
```

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

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

In [ ]:
```
arr = np.array([1, 2, 3], dtype=np.int8)
print(arr, arr.dtype)
```

```
[1 2 3] int8
```

In [ ]:
```
arr = np.array([1.1, 2.2, 3, 4], dtype=np.int8)
print(arr, arr.dtype)
```

```
[1 2 3 4] int8
```

Type Casting (`astype`)

You can convert an array from one dtype to another.

Example:

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

Out[ ]:
```
dtype('int64')
```

In [ ]:
```
new_arr = arr.astype(np.float64)
new_arr
```

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

In [ ]:
```
new_arr.dtype
```

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

In [ ]:
```
new_arr2 = arr.astype(np.int64)
new_arr2
```

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

In [ ]:
```
new_arr2.dtype
```

Out[ ]:
```
dtype('int64')
```

Type Casting Errors

Example:

In [ ]:
```
arr = np.array(["1", "2", "Imran"])
arr2 = arr.astype(np.int64)
arr2
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-2953318522.py in <cell line: 0>()
      1 arr = np.array(["1", "2", "Imran"])
----> 2 arr2 = arr.astype(np.int64)
      3 arr2

ValueError: invalid literal for int() with base 10: np.str_('Imran')
```

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-1637308676.py in <cell line: 0>()
----> 1 arr = np.array([1,2,3],[4,5,6])

TypeError: Field elements must be 2- or 3-tuples, got '4'
```

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

NumPy Array Attributes

In [ ]:
```
arr
```

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

In [ ]:
```
arr.ndim
```

Out[ ]:
```
2
```

In [ ]:
```
arr.shape
```
Out[ ]:
```
(2, 3)
```

In [ ]:
```
arr.size
```
Out[ ]:
```
6
```

In [ ]:
```
arr.itemsize
```
Out[ ]:
```
8
```

Array Reshaping - Reshape, Ravel, Flatten

Reshaping in NumPy

Reshaping means changing the shape of an array without changing its data.

Example:

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

In [ ]:
```
reshape2 = arr.reshape(3, 2)
print(reshape2)
```
```
[[1 2]
 [3 4]
 [5 6]]
```

Key Points

- Total elements must remain the same: (2, 3) has 6 elements, same as (6,).
- You can use -1 to let NumPy automatically calculate one dimension:

`ravel()`

- Returns a 1D view of the array (if possible, no copy is made).
- Changes in the raveled array may affect the original array.

Example:

In [ ]:
```
ravel = reshape2.ravel()
print(ravel)
```
```
[1 2 3 4 5 6]
```

In [ ]:
```
ravel[0] = 100
```

In [ ]:
```
print(ravel)
```

```
[100   2   3   4   5   6]
```

In [ ]:
```
print(reshape2)
```

```
[[100   2]
 [  3   4]
 [  5   6]]
```

`flatten()`

- Returns a 1D copy of the array.
- Always creates a new array (changes won't affect original).

Example:

In [ ]:
```
flat = reshape2.flatten()
print(flat)
```

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

In [ ]:
```
flat[0] = 1
```

In [ ]:
```
print(flat)
```

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

In [ ]:
```
print(reshape2)
```

```
[[100   2]
 [  3   4]
 [  5   6]]
```

Difference Between `reshape`, `ravel`, and `flatten`

Method	Purpose	Copy or View?	Shape Change?	Affects Original?
`reshape`	Change dimensions	Returns view (if possible)	✅	If view, changes affect original
`ravel`	1D view (flatten)	View (copy if needed)	✅	If view, changes affect original
`flatten`	1D copy	Always copy	✅	No

Arithmetic Operations on NumPy Arrays

1. Basic Arithmetic Operations

NumPy allows element-wise arithmetic between arrays (of the same shape) or between an array and a scalar.

Example:

In [ ]:
```
a = np.array([10, 20, 30, 40])
b = np.array([1, 2, 3, 4])

print(a + b)   # Addition
print(a - b)   # Subtraction
print(a * b)   # Multiplication
print(a / b)   # Division
print(a % b)   # Modulus
print(a ** b)  # Power
```

```
[11 22 33 44]
[ 9 18 27 36]
[ 10  40  90 160]
[10. 10. 10. 10.]
[0 0 0 0]
[      10      400    27000 2560000]
```

2. Operations with Scalars

If you operate an array with a single number, NumPy applies the operation to all elements.

Example:

In [ ]:
```
print(a + 5)   # Adds 5 to every element
print(a * 2)   # Multiplies every element by 2
```

```
[15 25 35 45]
[20 40 60 80]
```

3. Universal Functions (ufuncs)

NumPy provides built-in functions for common operations:

Example:

In [ ]:
```
print(np.add(a, b))
print(np.subtract(a, b))
print(np.multiply(a, b))
print(np.divide(a, b))
print(np.power(a, b))
```

```
[11 22 33 44]
[ 9 18 27 36]
[ 10  40  90 160]
[10. 10. 10. 10.]
[      10      400    27000 2560000]
```

In [ ]:
```
arr = np.array([1, 4, 9, 16])
```

In [ ]:
```
print(np.sqrt(arr))
```

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

In [ ]:
```
print(np.exp([1, 2]))
```

```
[2.71828183 7.3890561 ]
```

In [ ]:
```
angle = np.array([0, np.pi, np.pi/2, np.pi/4])
print(np.sin(angle))
```

```
[0.00000000e+00 1.22464680e-16 1.00000000e+00 7.07106781e-01]
```

4. Broadcasting

When arrays have different shapes, NumPy applies broadcasting rules to match shapes for element-wise operations.

Example:

In [ ]:
```
x = np.array([1, 2, 3])
y = np.array([[10], [20], [30]])

print(x + y)
```

```
[[11 12 13]
 [21 22 23]
 [31 32 33]]
```

5. Comparison Operations

Element-wise comparisons return boolean arrays:

Example:

In [ ]:
```
print(a > 20)
print(a == 30)
```

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

Indexing and Slicing

Indexing in NumPy

Indexing means accessing elements from a NumPy array. NumPy indexing works similarly to Python lists but supports multi-dimensional arrays.

1D Array

Example:

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

print(arr[0])
print(arr[-1])
print(arr[2])
```

```
10
50
30
```

2D Array

Example:

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

print(arr2d[0, 0])  # First row, first column → 1
print(arr2d[1, 2])  # Second row, third column → 6
print(arr2d[-1, -1]) # Last row, last column → 9
```

```
1
6
9
```

3D Array

Example:

In [ ]:
```
arr3d = np.array([
    [[1, 2], [3, 4]],
    [[5, 6], [7, 8]]
])

print(arr3d[0, 1, 1])  # First block, second row, second column → 4
print(arr3d[1, 0, 0])  # Second block, first row, first column → 5
```

```
4
5
```

Slicing in NumPy

Slicing means getting a subarray from the main array.

Syntax:

```
array[start:end:step]
```

1D Array Slicing

Example:

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

print(arr[1:4])    # Elements from index 1 to 3 → [20 30 40]
print(arr[:3])     # From start to index 2 → [10 20 30]
print(arr[2:])     # From index 2 to end → [30 40 50]
print(arr[::2])    # Step of 2 → [10 30 50]
```

```
[20 30 40]
[10 20 30]
[30 40 50]
[10 30 50]
```

2D Array Slicing

Example:

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

print(arr2d[0:2, 1:3])  # Rows 0-1, Columns 1-2 → [[2 3], [5 6]]
print(arr2d[:, 1])      # All rows, Column 1 → [2 5 8]
print(arr2d[1, :])      # Row 1, all columns → [4 5 6]
```

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

3D Array Slicing

it works just like 2D slicing, but you have an extra dimension to think about.

Understanding a 3D array

A 3D array has three axes:

- Axis 0 → depth (pages)
- Axis 1 → rows
- Axis 2 → columns

Example:

In [ ]:
```
# Create a 3D array with shape (2 blocks, 3 rows, 4 columns)

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

Format:

```
arr[depth_start:depth_end, row_start:row_end,
col_start:col_end]
```

In [ ]:
```
#Get first page only
arr[0, :, :]
```

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

In [ ]:
```
#Get first row of every page
arr[:, 0, :]
```

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

In [ ]:
```
#Get first two columns of first page
arr[0, :, 0:2]
```

Out[ ]:
```
array([[0, 1],
       [4, 5],
       [8, 9]])
```

In [ ]:
```
#Get a sub-block (page 0, rows 1 to 2, cols 2 to 3)
arr[0, 1:3, 2:4]
```

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

Negative Indexing

Example:

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

print(arr[-3:])   # Last 3 elements → [30 40 50]
print(arr[:-2])   # All except last 2 → [10 20 30]
```

```
[30 40 50]
[10 20 30]
```

In [ ]:
```
# np.take -> build in function to perform indexing and slicing

arr = np.array([10, 20, 30, 40, 50])
ind = [0, 2]
print(np.take(arr, ind))
```

```
[10 30]
```

In [ ]:
```
#iterating with nditer()

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

for x in np.nditer(arr):
  print(x, end =" ")
```

```
1 2 3 4
```

In [ ]:
```
#ndenumerate() -> both index + value

for idx, x in np.ndenumerate(arr):
  print(idx, x)  # idx = tuple of (row_index, col_index), x = element value
```

```
(0, 0) 1
(0, 1) 2
(1, 0) 3
(1, 1) 4
```

Key Difference from Python Lists:

When you slice a NumPy array, it returns a view (not a copy) of the original data, so changes affect the original array.

In [ ]:
```
a = np.array([1, 2, 3, 4])
b = a[1:3]
b[0] = 99
print(a)  # Original array changed
```

```
[ 1 99  3  4]
```

Views Vs Copy

View

- Shares the same data in memory.
- Changes in the view affect the original array, and vice versa.
- Created when you do slicing (most of the time).

Example:

In [ ]:
```
arr = np.array([10, 20, 30, 40, 50])
view = arr[1:4]  # This is a view, not a copy

view[0] = 999
print("View:", view)
print("Original:", arr)
```

```
View: [999  30  40]
Original: [ 10 999  30  40  50]
```

Copy

- Has its own memory (independent of the original).
- Changes in the copy do not affect the original.
- Created when you explicitly call `.copy()`.

Example:

In [ ]:
```
copy_arr = arr[1:4].copy()
copy_arr[0] = 555
```

```
print("Copy:", copy_arr)
print("Original:", arr)
```

```
Copy: [555  30  40]
Original: [ 10 999  30  40  50]
```

How to Check?

You can check if two arrays share memory:

In [ ]:
```
np.shares_memory(arr, view)
```

Out[ ]:
```
True
```

In [ ]:
```
np.shares_memory(arr, copy_arr)
```

Out[ ]:
```
False
```

Important Points

- Slicing → usually creates a view.
- Operations like `.reshape()` also often create views (if possible).
- Fancy indexing (like `arr[[0, 2, 4]]`) → creates a copy.
- `.copy()` → forces a copy.

Transpose of a Matrix

The transpose of a matrix is when you flip it over its diagonal — rows become columns, and columns become rows.

Example:

Using `.T` Attribute

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

print("Original:\n", arr)

# Print the transposed array (rows become columns)
print("Transpose:\n", arr.T)
```

```
Original:
 [[1 2 3]
 [4 5 6]]
Transpose:
 [[1 4]
 [2 5]
 [3 6]]
```

Using `np.transpose()` Function

In [ ]:
```
arr = np.array([[1, 2], [3, 4], [5, 6]])
transposed = np.transpose(arr) # Transpose the array (convert rows to columns)

print("Original:\n", arr)
print("Transpose:\n", transposed)
```

```
Original:
 [[1 2]
 [3 4]
 [5 6]]
Transpose:
 [[1 3 5]
 [2 4 6]]
```

For Higher-Dimensional Arrays

You can specify the axes order.

In [ ]:
```
arr3d = np.arange(8).reshape(2, 2, 2)
print("Original Shape:", arr3d.shape)

transposed = np.transpose(arr3d, (0, 2, 1))
print("Transposed Shape:", transposed.shape)
```

```
Original Shape: (2, 2, 2)
Transposed Shape: (2, 2, 2)
```

- `.T` is just a shorthand for `np.transpose(arr)` for 2D arrays.
- For 1D arrays, transpose has no effect because there's only one axis.
- Transpose does not make a copy — it returns a view (unless reordering axes requires a copy).

Concatenation and Stacking in NumPy Arrays

In NumPy, we can combine arrays in two main ways:

- Concatenation → join along an existing axis
- Stacking → join along a new axis

Concatenation

- Meaning: Joins arrays along an existing axis.
- Shape change: Only the size along the chosen axis changes; the number of dimensions stays the same.

Example:

Using `np.concatenate()`

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

```
In [ ]: res1 = np.concatenate((a, b))
        res1
```

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

```
In [ ]: # Axis 0 → vertical join (rows)
        res1 = np.concatenate((a, b), axis=0)
        print("Axis 0:\n", res1)
```

```
Axis 0:
 [[1 2]
 [3 4]
 [5 6]]
```

```
In [ ]: # Axis 1 → horizontal join (columns)
        c = np.array([[5], [6]])
        res2 = np.concatenate((a, c), axis=1)

        print("Axis 1:\n", res2)
```

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

Stacking

- Meaning: Joins arrays along a new axis.
- Shape change: The number of dimensions increases by 1.

Example:

`np.stack()`

```
In [ ]: x = np.array([1, 2])
        y = np.array([3, 4])

        stacked = np.stack((x, y), axis=0)  # vertical

        print(stacked)
```

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

```
In [ ]: stacked2 = np.stack((x, y), axis=1) # horizontal

        print(stacked2)
```

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

Vertical & Horizontal Stack Shortcuts

`np.vstack()` → vertical stack

In [ ]:
```
a = np.array([1, 2])
b = np.array([3, 4])
print(np.vstack((a, b)))
```

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

`np.hstack()` → horizontal stack

In [ ]:
```
a = np.array([1, 2])
b = np.array([3, 4])
print(np.hstack((a, b)))
```

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

`np.dstack()` → depth stack

Stacks along axis=2 (3rd dimension).

In [ ]:
```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print(np.dstack((a, b)))
```

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

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

Splitting NumPy Arrays

In NumPy, splitting means dividing an array into multiple sub-arrays. We can split horizontally, vertically, or depth-wise depending on the axis.

Example:

1. Using `np.split()`

- Splits an array into equal-sized sub-arrays.
- Syntax:

```
np.split(array, sections, axis)
```

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

# Split into 3 equal parts
```

```
parts = np.split(arr, 3)
print(parts)
```

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

2. Using `np.array_split()`

Same as `split()` but allows unequal sizes.

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

parts = np.array_split(arr, 3)
print(parts)
```

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

3. Vertical Split → `np.vsplit()`

Splits along rows `(axis=0)`.

In [ ]:
```
arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
v_parts = np.vsplit(arr, 2)
print(v_parts)
```

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

4. Horizontal Split → `np.hsplit()`

Splits along columns `(axis=1)`.

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

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

5. Depth Split → `np.dsplit()`

Splits along depth `(axis=2)`.

In [ ]:
```
# 3D array with shape (2, 2, 2)
arr = np.array([
    [[1, 2], [3, 4]],
    [[5, 6], [7, 8]]
])

# Split the array along the depth (3rd) axis into 2 parts
d_parts = np.dsplit(arr, 2)
print(d_parts)
```

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

       [[5],
        [7]]]), array([[[2],
        [4]],

       [[6],
        [8]]])]
```

Repeat Vs Tile in NumPy Arrays

`repeat` vs `tile` in NumPy because they look similar at first, but actually work quite differently.

`np.repeat()`

- Repeats each element a specific number of times.
- Works element-by-element.

Syntax:

```
np.repeat(array, repeats, axis=None)
```

- `repeats` → how many times to repeat each element.
- `axis` → repeat along rows (axis=0), columns (axis=1), or flatten if None.

Example 1: Without axis

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

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

Example 2: With axis

In [ ]:
```
arr = np.array([[1, 2], [3, 4]])
print(np.repeat(arr, 4, axis=0))  # repeat rows
```

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

`np.tile()`

- Repeats the whole array a number of times (like tiling a floor).

- Works block-by-block.

Syntax:

```
np.tile(array, reps)
```

Example 1: 1D array

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

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

Example 2: 2D array

In [ ]:
```
arr = np.array([[1, 2], [3, 4]])
print(np.tile(arr, (2, 3)))  # 2 times vertically, 3 times horizontally
```

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

Key Difference

Feature	`repeat`	`tile`
Works on	**Each element**	**Whole array**
Behavior	Duplicates every value individually	Duplicates entire array block
Shape change	Can increase size along specific axis	Expands in a grid pattern

Aggregate Functions in NumPy Arrays

What are Aggregate Functions?

Aggregate functions perform calculations over an entire array (or along a specific axis) and return a single value or a reduced array.

Example: sum, mean, min, max, standard deviation, etc.

Common Aggregate Functions in NumPy

Function	Description
`np.sum()`	Sum of elements
`np.mean()`	Mean (average) value
`np.min()` / `np.max()`	Minimum / Maximum value
`np.argmin()` / `np.argmax()`	Index of min / max value
`np.std()`	Standard deviation

Function	Description
`np.var()`	Variance
`np.median()`	Median value
`np.prod()`	Product of all elements
`np.cumsum()`	Cumulative sum
`np.cumprod()`	Cumulative product

Example:

1. Sum

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

print(np.sum(arr))
```

```
21
```

In [ ]:
```
# Column-wise sum
print(np.sum(arr, axis=0))
```

```
[5 7 9]
```

In [ ]:
```
# Row-wise sum
print(np.sum(arr, axis=1))
```

```
[ 6 15]
```

2. Mean (Average)

In [ ]:
```
print(np.mean(arr))
```

```
3.5
```

In [ ]:
```
# Column-wise mean
print(np.mean(arr, axis=0))
```

```
[2.5 3.5 4.5]
```

3. Min / Max

In [ ]:
```
print(np.min(arr))
print(np.max(arr))
print(np.min(arr, axis=0))
print(np.max(arr, axis=1))
```

```
1
6
[1 2 3]
[3 6]
```

4. Index of Min / Max

In [ ]:
```
print(np.argmin(arr))        #(index in flattened array)
print(np.argmax(arr))
```

```
0
5
```

5. Standard Deviation & Variance

In [ ]:
```
print(np.std(arr))
print(np.var(arr))
```

```
1.707825127659933
2.9166666666666665
```

6. Median

In [ ]:
```
print(np.median(arr))
```

```
3.5
```

7. Product of Elements

In [ ]:
```
print(np.prod(arr))        # (product of all elements)
print(np.prod(arr, axis=0)) # Column-wise product
```

```
720
[ 4 10 18]
```

8. Cumulative Sum & Product

In [ ]:
```
# Cumulative sum of all elements in a flattened way
print(np.cumsum(arr))


# Cumulative product of all elements in a flattened way
print(np.cumprod(arr))
```

```
[ 1  3  6 10 15 21]
[  1   2   6  24 120 720]
```

Most aggregate functions have an `axis` parameter:

- `axis=0` → operate column-wise
- `axis=1` → operate row-wise
- `axis=None` (default) → flatten array and operate

Conditional-Based Operations on NumPy Arrays

What Are Conditional-Based Operations?

These are operations where conditions are applied to array elements to filter, replace, or perform calculations.

1. Boolean Conditions

You can directly compare NumPy arrays with scalars or other arrays, and the result will be a Boolean array.

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

print(arr > 25)
print(arr % 2 == 0)
```

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

2. Boolean Indexing

You can use these Boolean arrays to filter values.

```
In [ ]: # Print elements greater than 25
print(arr[arr > 25])


# Print elements divisible by 4
print(arr[arr % 4 == 0])
```

```
[30 40 50]
[20 40]
```

3.Using `np.where()`

```
np.where(condition, value_if_true, value_if_false)
```

Returns indices where the condition is true, or allows conditional replacement.

```
In [ ]: # Get indices where condition is true
new_arr = np.where(arr > 25)
print(new_arr)
```

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

```
In [ ]: # Conditional replacement
new_arr = np.where(arr > 25, 100, arr)
print(new_arr)
```

```
[ 10  20 100 100 100]
```

4. `np.argwhere()`

`np.argwhere(condition)` returns the indices of elements where the condition is `True`.

- Unlike `np.where()`, it always returns indices as a 2D array (each row is the index of one matching element).
- Works for N-dimensional arrays.

Basic Example

```
In [ ]: # Get positions (indices) where elements are greater than 25
pos = np.argwhere(arr > 25)
```

```
# Prints 2D array of positions
print(pos)

# Flatten the positions array to 1D
print(pos.flatten())
```

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

In [ ]:
```
indices = np.argwhere(arr > 25)
print(indices)

# To get the actual values:
print(arr[indices])
```

```
[[2]
 [3]
 [4]]
[[30]
 [40]
 [50]]
```

2D Array

In [ ]:
```
arr2d = np.array([[5, 15, 25],
                  [35, 45, 55],
                  [65, 75, 85]])

# Find positions where elements are greater than 40
indices = np.argwhere(arr2d > 40)

# Prints row and column positions
print(indices)
```

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

3D Array

In [ ]:
```
arr3d = np.arange(1, 13).reshape(2, 2, 3)
print(arr3d)


# Find positions where elements are greater than 6
# np.argwhere returns indices for each dimension (layer, row, col)
indices = np.argwhere(arr3d > 6)

# Prints the (layer, row, column) positions
print(indices)
```

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

 [[ 7  8  9]
  [10 11 12]]]
[[1 0 0]
 [1 0 1]
 [1 0 2]
 [1 1 0]
 [1 1 1]
 [1 1 2]]
```

Checking Any or All

In [ ]:
```
print(np.any(arr > 45))  # (at least one element > 45)
print(np.all(arr > 5))   # (all elements > 5)
```

```
True
True
```

Extract Indices (`np.where` without replacement)

In [ ]:
```
indices = np.where(arr > 25)
print(indices)
print(arr[indices])
```

```
(array([2, 3, 4]),)
[30 40 50]
```

Difference Between `np.where()` and `np.argwhere()`

Feature	`np.where()`	`np.argwhere()`
Output	Tuple of arrays (one per dimension)	2D array (each row is full coordinates)
Shape	Separate index arrays	Single combined index array
Use case	Direct indexing or replacement	Iterating over coordinates

5. Multiple Conditions

i. Logical AND

Using `&` (bitwise AND)

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

mask = (arr > 15) & (arr < 45)  # both conditions true
print(mask)

print(arr[mask])  # filtered values
```

```
[False  True  True  True False]
[20 30 40]
```

Using `np.logical_and()`

In [ ]:
```python
mask = np.logical_and(arr > 15, arr < 45)
print(mask)
```

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

ii. Logical OR

Using `|` (bitwise OR)

In [ ]:
```python
mask = (arr < 15) | (arr > 45)
print(mask)
```

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

Using `np.logical_or()`

In [ ]:
```python
mask = np.logical_or(arr < 15, arr > 45)
print(mask)
```

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

iii. Logical NOT

In [ ]:
```python
# bitwise NOT
mask = ~(arr > 25)
print(mask)


# or using function
mask = np.logical_not(arr > 25)
print(mask)
```

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

Operation	Bitwise Operator	NumPy Function	
AND	`&`	`np.logical_and()`	
OR	`	`	`np.logical_or()`
NOT	`~`	`np.logical_not()`	

Masking in Arrays

Masking means creating a boolean array (mask) that specifies which elements of another array should be selected, modified, or replaced.

1. Basic Boolean Masking

A mask is a boolean array (`True` / `False`) that is the same shape as the array it's applied to.

In [ ]:
```python
arr = np.array([10, 20, 30, 40, 50])

# Create mask for values greater than 25
```

```
mask = arr > 25
print(mask)

# Use mask to select elements
print(arr[mask])
```

```
[False False  True  True  True]
[30 40 50]
```

2. Masking with Conditions

You can directly use conditions inside array indexing without creating a separate mask variable.

In [ ]:
```
print(arr[arr % 20 == 0])
```

```
[20 40]
```

3. Multiple Conditions in Masks

Use bitwise operators (`&` , `|` , `~`) with parentheses for combining conditions.

In [ ]:
```
# Values between 15 and 45
print(arr[(arr > 15) & (arr < 45)])

# Values less than 20 or greater than 40
print(arr[(arr < 20) | (arr > 40)])
```

```
[20 30 40]
[10 50]
```

4. Modifying Values Using Masks

Masks can also be used to change array elements.

In [ ]:
```
arr[arr > 25] = 999
print(arr)
```

```
[ 10  20 999 999 999]
```

5. Masking in 2D Arrays

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

mask = matrix > 5
print(mask)

print(matrix[mask])
```

```
[[False False False]
 [False False  True]
 [ True  True  True]]
[6 7 8 9]
```

6. Inverse Masking (NOT)

Use `~` to invert a mask.

In [ ]:
```
arr = np.array([10, 20, 30, 40, 50])
mask = arr > 25
print(arr[~mask])
```

```
[10 20]
```

7.Masking with `np.ma` (Masked Arrays)

NumPy has a masked array type to hide certain values completely.

In [ ]:
```
# Create a masked array with elements [1,2,3,4,5]
# mask = [0,1,0,0,1] → 1 means the element is masked (hidden), 0 means visible

masked_arr = np.ma.masked_array([1, 2, 3, 4, 5], mask=[0, 1, 0, 0, 1])
print(masked_arr)
```

```
[1 -- 3 4 --]
```

Broadcasting

Broadcasting is the method NumPy uses to perform operations on arrays of different shapes without explicitly copying data. It automatically expands smaller arrays to match the shape of larger arrays during arithmetic or other element-wise operations.

1. Same Shape — No Broadcasting

If two arrays have the same shape, operations happen element-wise without any expansion.

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

print(a + b)
```

```
[5 7 9]
```

2. Scalar with Array (Simple Broadcasting)

A scalar is stretched to match the array shape.

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

```
[6 7 8]
```

3. Broadcasting with Different Shapes

Rule:

- Compare shapes from right to left:
- If dimensions are equal → ✅ OK
- If one dimension is 1 → expand to match the other
- Otherwise → ❌ Error

Example:

```
In [ ]: A = np.array([[1, 2, 3],[4, 5, 6]])      # Shape (2, 3)

B = np.array([10, 20, 30])

# B is broadcasted to shape (2, 3)
print(A + B)
```

```
[[11 22 33]
 [14 25 36]]
```

4. Broadcasting with Extra Dimensions

```
In [ ]: A = np.array([[1], [2], [3]])  # Shape (3, 1)
B = np.array([10, 20, 30])     # Shape (3,)

# ❌ Direct addition fails because (3,1) vs (3,)
# ✅ Reshape B to (1, 3)
print(A + B)
```

```
[[11 21 31]
 [12 22 32]
 [13 23 33]]
```

5. Practical Example

```
In [ ]: prices = np.array([100, 200, 300])  # Prices for 3 items
discounts = np.array([[10], [20]])  # Discounts for 2 customers

# Broadcasting applies each discount to all prices
final_prices = prices - discounts
print(final_prices)
```

Vectorization in NumPy Arrays

Vectorization means performing operations on entire arrays at once without using explicit loops in Python. It's about replacing slow, element-by-element loops with fast, underlying C operations.

Why Vectorization?

- Python loops are slow → They run in the Python interpreter, which has overhead for each iteration.
- NumPy operations are fast → Implemented in optimized C, Fortran, or BLAS/LAPACK code.
- Makes code cleaner, shorter, and more readable.

Example — Without Vectorization

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

# Loop-based approach
for x in arr:
    result.append(x ** 2)

print(result)
```

```
[np.int64(1), np.int64(4), np.int64(9), np.int64(16), np.int64(25)]
```

Drawbacks:

- Manual loop → slow for large datasets
- More code → harder to read

Example — With Vectorization

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

# Vectorized operation
result = arr ** 2

print(result)
```

```
[ 1  4  9 16 25]
```

Vectorized Operations Examples

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

print(a + b)
print(a * b)
print(a ** 2)
print(np.sqrt(a))
```

```
[5 7 9]
[ 4 10 18]
[1 4 9]
[1.         1.41421356 1.73205081]
```

These operations happen element-wise automatically.

Vectorization with Functions

NumPy also provides vectorized universal functions (ufuncs) for mathematical operations:

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

```
[ 1  4  9 16 25]
```

In [ ]:
```
result = np.square(arr)
print(result)
```

```
[ 1  4  9 16 25]
```

In [ ]:
```
x = np.linspace(0, np.pi, 5)  # [0, π/4, π/2, 3π/4, π]
y = np.sin(x)
print(y)
```

```
[0.00000000e+00 7.07106781e-01 1.00000000e+00 7.07106781e-01
 1.22464680e-16]
```

Performance Example

In [ ]:
```
import time

arr = np.arange(1_000_000)

# Loop version
start = time.time()
result_loop = [x * 2 for x in arr]
end = time.time()
print("Loop time:", end - start)

# Vectorized version
start = time.time()
result_vec = arr * 2
end = time.time()
print("Vectorized time:", end - start)
```

```
Loop time: 0.2649991512298584
Vectorized time: 0.005300283432006836
```

Dealing with Missing Values

In NumPy, missing values are usually represented as np.nan (Not a Number).

1.Detect Missing Values

In [ ]:
```
arr = np.array([1, 2, np.nan, 4, np.nan, 6])

print(np.isnan(arr))
```

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

`np.isnan()` returns a boolean mask where `True` means missing.

2.Removing Missing Values

In [ ]:
```
clean_arr = arr[~np.isnan(arr)]
print(clean_arr)
```

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

`~` inverts the boolean mask (keeps only non-missing values).

3.Replacing Missing Values

In [ ]:
```
filled_arr = np.where(np.isnan(arr), 0, arr)
print(filled_arr)
```

```
[1. 2. 0. 4. 0. 6.]
```

Replace all NaNs with a fixed value (here `0`).

4.Using Aggregate Functions with Missing Values

- Problem: Normal functions treat NaN as infectious:

In [ ]:
```
print(np.mean(arr))
```

```
nan
```

Solution: Use `nan`-safe versions:

In [ ]:
```
print(np.nanmean(arr))
print(np.nanmax(arr))
print(np.nansum(arr))
```

```
3.25
6.0
13.0
```

Example: Replace NaN with Mean

In [ ]:
```
mean_value = np.nanmean(arr)
arr[np.isnan(arr)] = mean_value
print(arr)
```

```
[1.   2.   3.25 4.   3.25 6.  ]
```

Summary Table

Task	Function
Detect NaN	`np.isnan()`
Remove NaN	`arr[~np.isnan(arr)]`
Replace NaN	`np.where()` or masking
Mean ignoring NaN	`np.nanmean()`
Sum ignoring NaN	`np.nansum()`
Max ignoring NaN	`np.nanmax()`