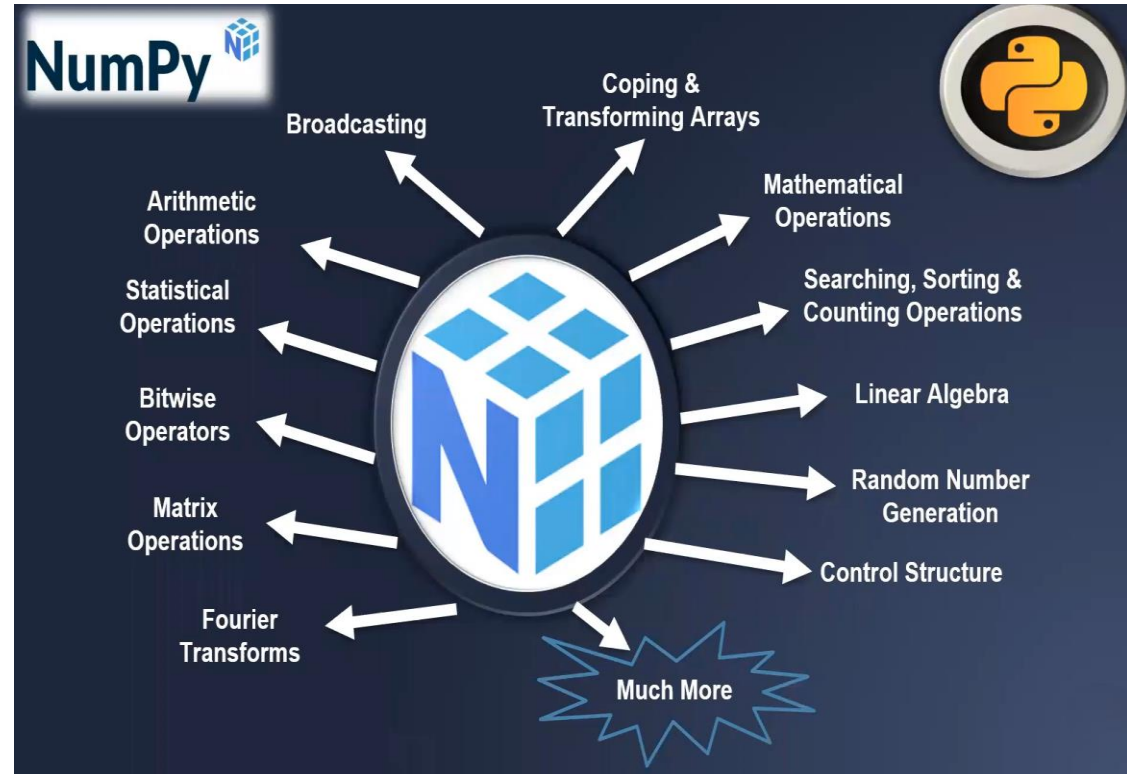


Introduction To NumPy

What is NumPy?

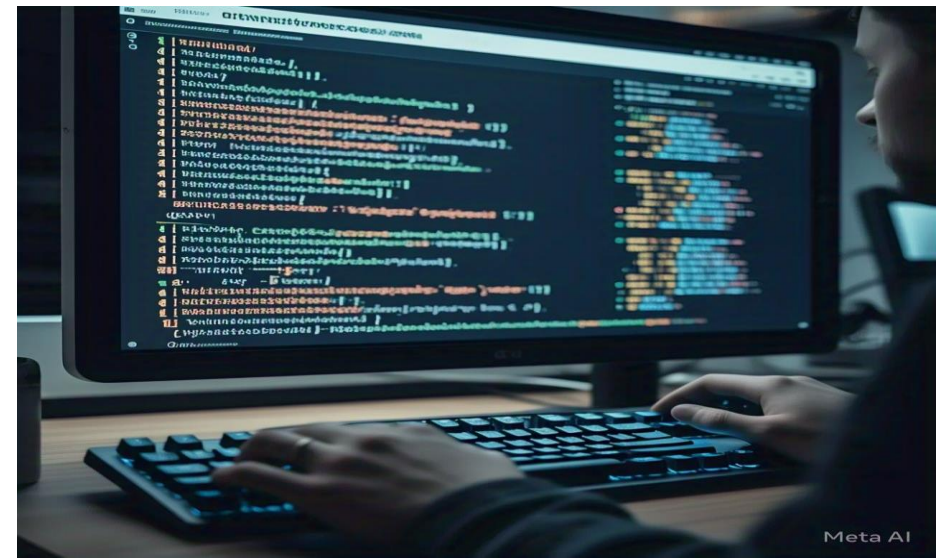
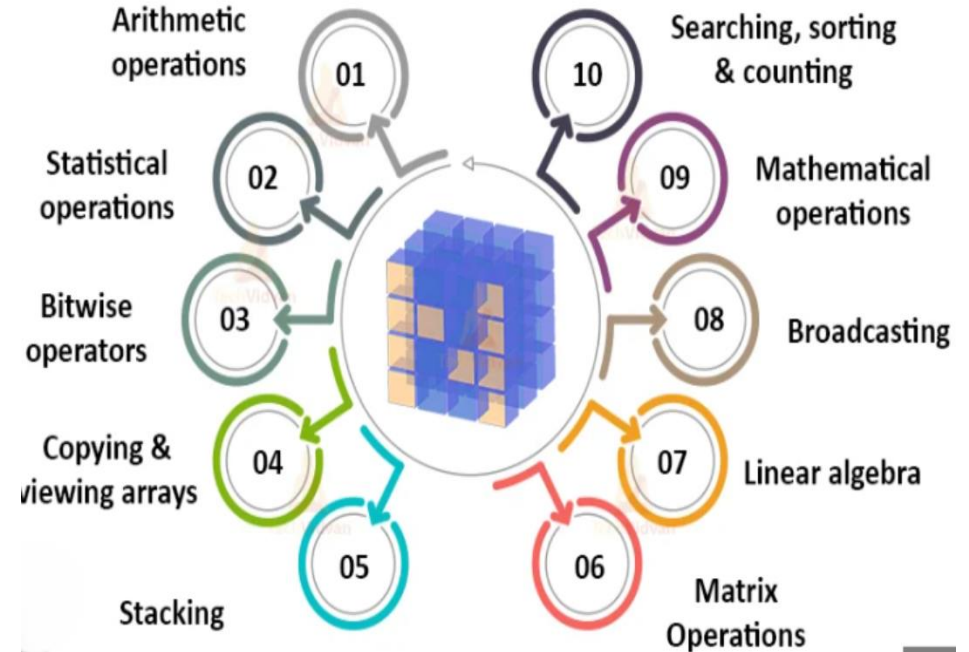
- NumPy Stand for Numerical Python.
- It is a library consisting of multidimensional array objects and a collection of routines for processing those arrays..
- It is a powerful library for numerical computations.
- Used heavily in **Data Science, Machine Learning, and Scientific Computing**
- Provides multi-dimensional arrays and tools to operate on them efficiently.
- Easy-to-use tool for numerical operations in Python



Why Use NumPy

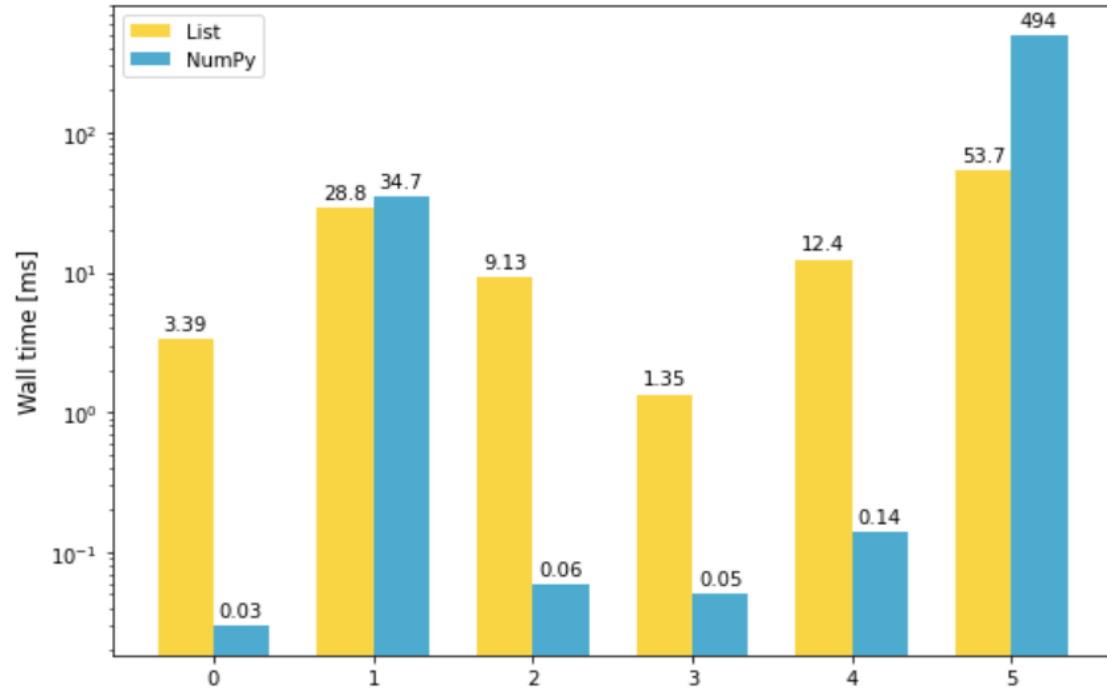
- Mathematical and logical operations on arrays can be performed. Also provide high performance.
- In Python we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- The Array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
- Arrays are very frequently used in data science, where speed and resources are very important.

Uses of NumPy

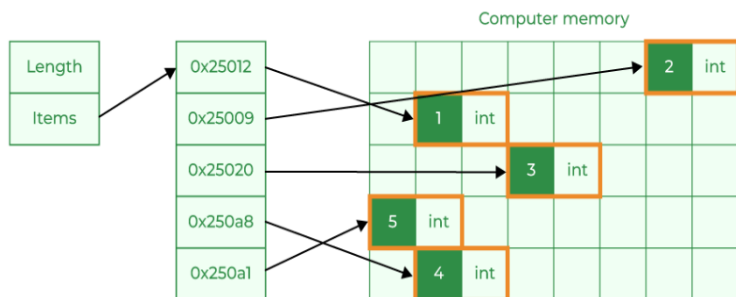


NumPy Array is faster than List

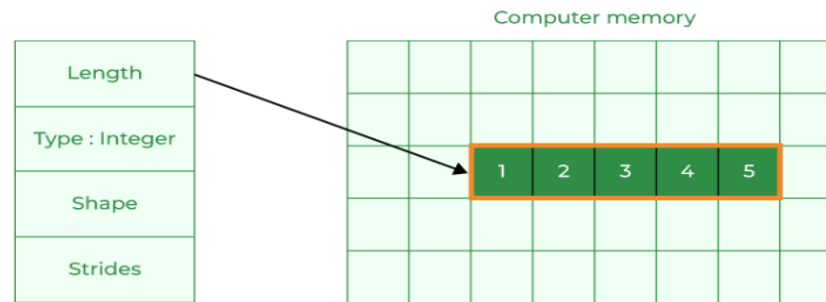
- NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
- This behavior is called locality of reference in computer science.
- This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.



Python List



NumPy Array



Installing Numpy

- Numpy is so popular that chances are you already have it installed, but otherwise is as easy as installing it with pip like this:
- `pip install numpy`
- Numpy installed in your environment automatically
- If You're Using Jupyter Notebook or Colab
- `!pip install numpy`
- The Exclamation ! allows running shell commands in notebook environments.
- If You're Using Anaconda
- `conda install numpy`



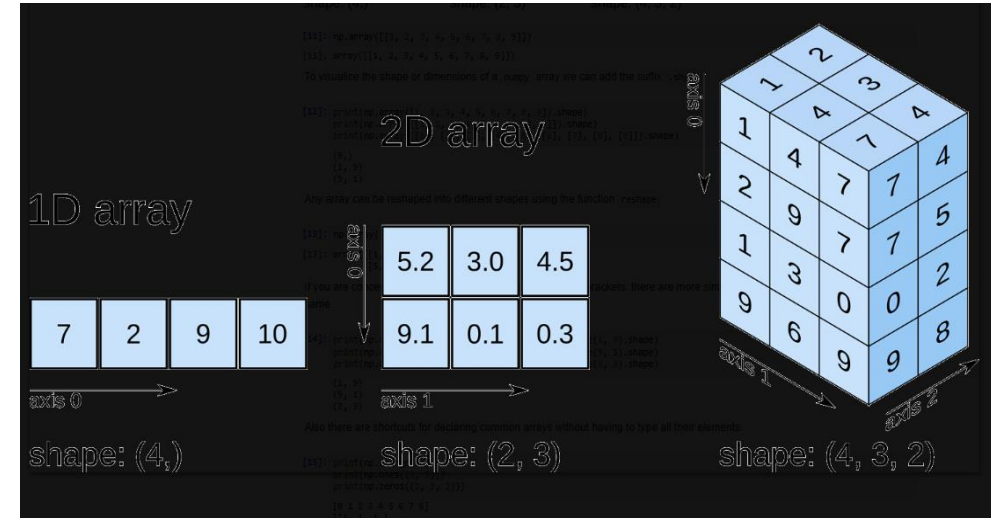
Importing NumPy

- Standard Import Convention
- `import numpy as np`
- **Why as np?**
 - Short and convenient alias.
 - Makes code cleaner and easier to read.
 - Used widely in tutorials, libraries, and documentation.
- Example
 - `import numpy as np`
 - `arr = np.array([1, 2, 3])`
 - `print(np.mean(arr))`



Arrays in NumPy

- An Array is a collection of values, organized in a specific order. NumPy's main object is the homogeneous multidimensional array.
- It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.
- In NumPy dimensions are called axis. The number of axis is rank.
- NumPy's array class is called ndarray. It is also known by the alias array..



1-D Array

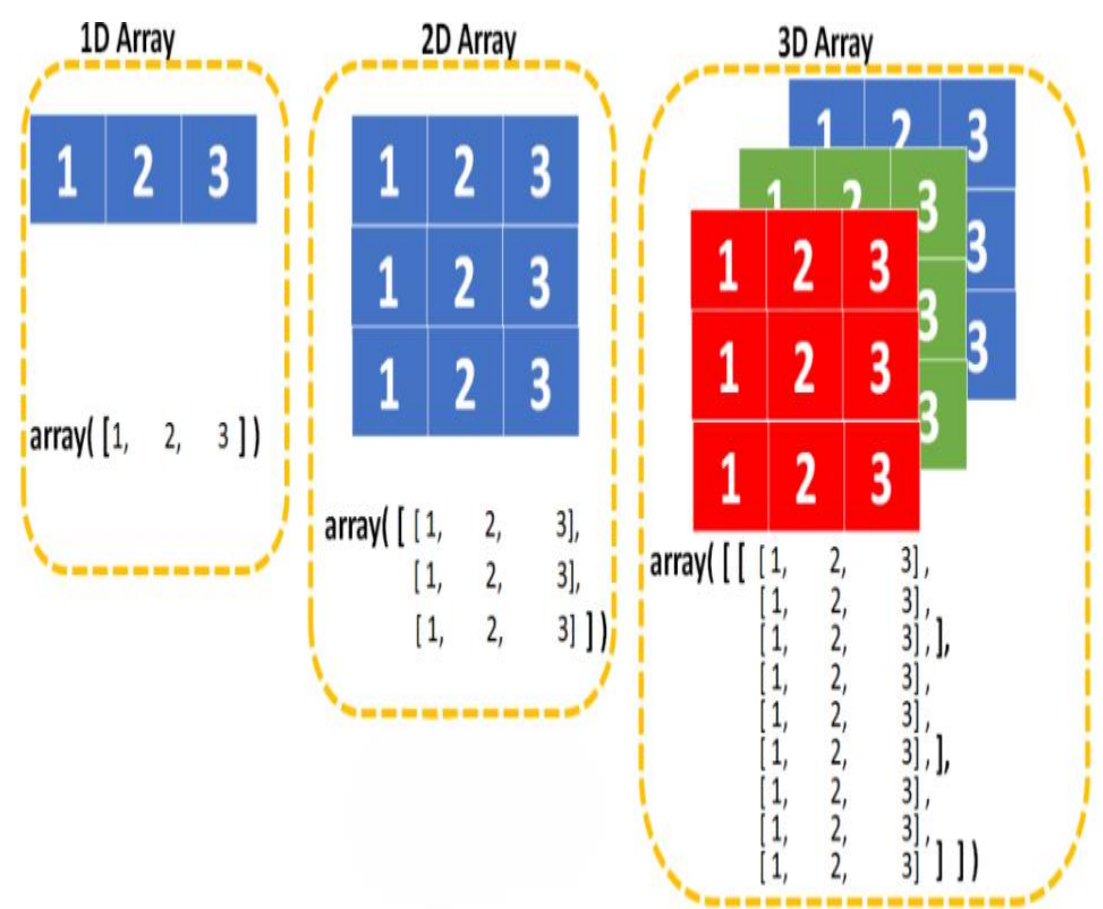
- An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array. These are the most common and basic arrays.
- **Import numpy as np**
- `arr = np.array([1, 2, 3, 4, 5])`
- `print(arr)`

2-D Array

- An array that has 2-D arrays as its elements is called 2-D-dimensional array. These are often used to represent matrix or 2nd order tensors.
- **Import numpy as np**
- `arr = np.array([[1, 2, 3],[4, 5,6]])`
- `print(arr)`

3-D Arrays

- An array that has 2-D arrays(matrices) as its elements is called 3-D-dimensional array. These are often used to represent matrix or 3rd order tensors.
- **Import numpy as np**
- `arr = np.array([[[1, 2, 3],[4, 5,6]], [[1,2,3],[4,5,6]]])`
- `print(arr)`



NumPy Array Indexing

- Access Array Elements
- Array indexing is the same as accessing an array element.
- You can access an array element by referring to its index number.
- The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.
- **import numpy as np**
`arr = np.array([1, 2, 3, 4])`
`print(arr[0])`

Element of array	2	3	11	9	6	4	10	12
Index	0	1	2	3	4	5	6	7

Access 2-D Arrays

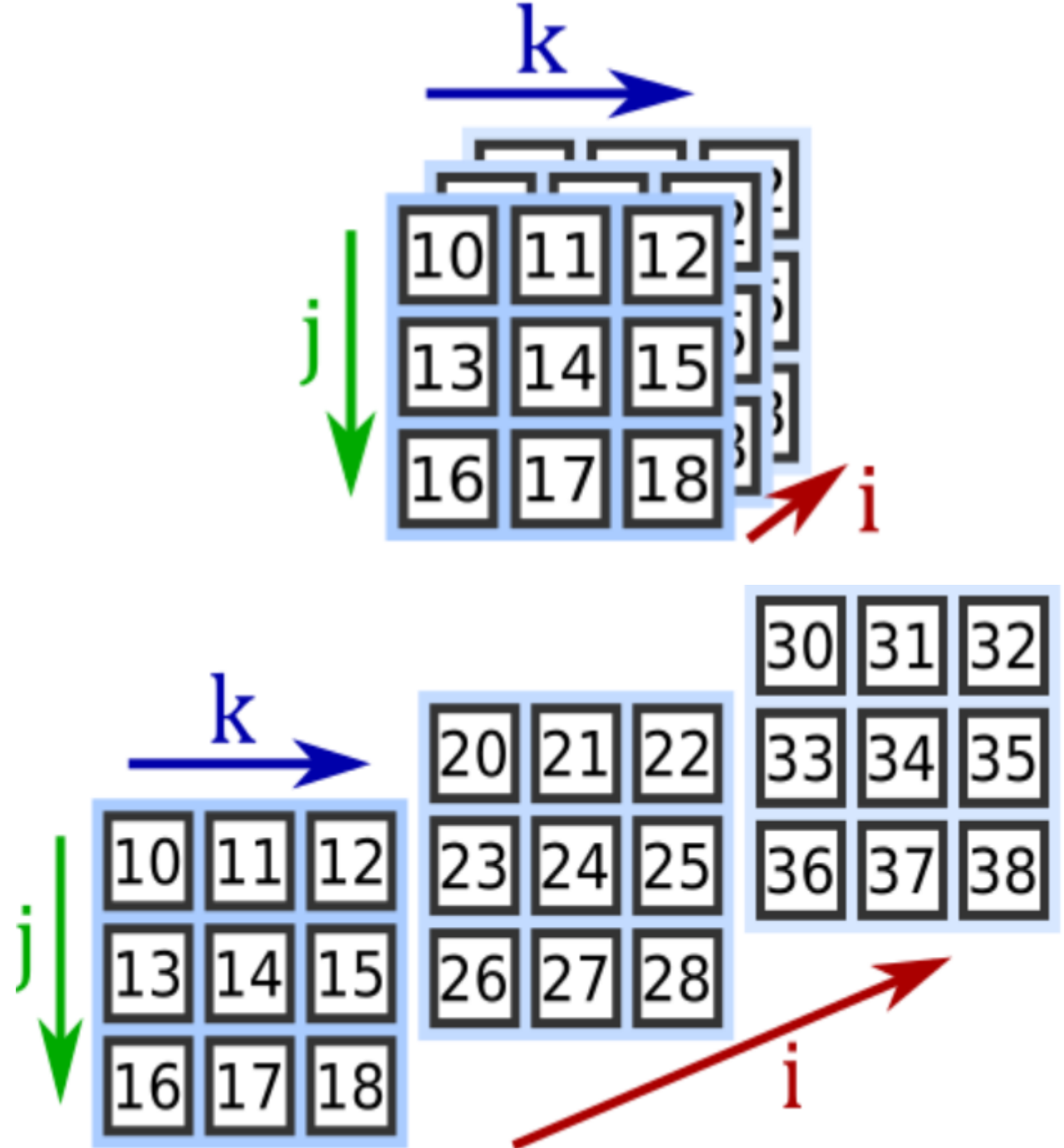
- To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.
- Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.
- **import numpy as np**
- **#Access the element on the first row, second column:**

```
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])  
print('2nd element on 1st row: ', arr[0, 1])
```

12(0,0)	11(0,1)	9 (0,2)
8(1,0)	0(1,1)	3(1,2)
10(2,0)	2(2,1)	5(2,2)

Access 3-D Arrays

- To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.
- The first index, **i**, selects the matrix
- The second index, **j**, selects the row
- The third index, **k**, selects the column
- **import numpy as np**
#Access the third element of the second array of the first array:
`arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])`
`print(arr[0, 1, 2])`



Negative Indexing

- Use negative indexing to access an array from the end.
- Example:
- Print the last element from the 2nd dim:
- **import numpy as np**

```
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])  
print('Last element from 2nd dim: ', arr[1, -1])
```

	1	3	5	7	9
index →	0	1	2	3	4
negative index →	-5	-4	-3	-2	-1

Copying and Sorting in NumPy

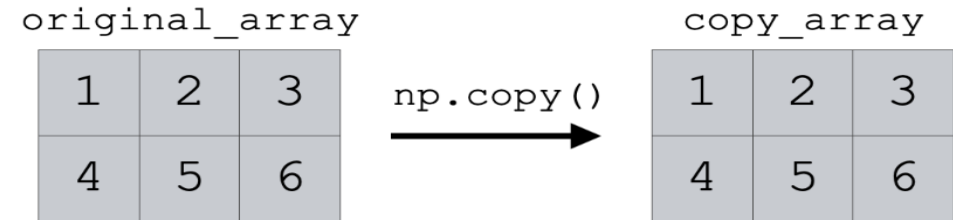
• Copying Array:

- Use `np.copy()` to create a **deep copy** of an array.
- A deep copy means the new array does **not share memory** with the original.
- Changes in the copied array **won't affect** the original one.

• Sorting Arrays:

- Use `np.sort()` to sort array **elements in ascending order**.
- It returns a **sorted copy** of the array (original remains unchanged).
- Works on 1D and 2D arrays — can specify axis.

NUMPY COPY COPIES NUMPY ARRAYS



numpy.sort

Sorted array along the first axis

15	01	94	19
----	----	----	----

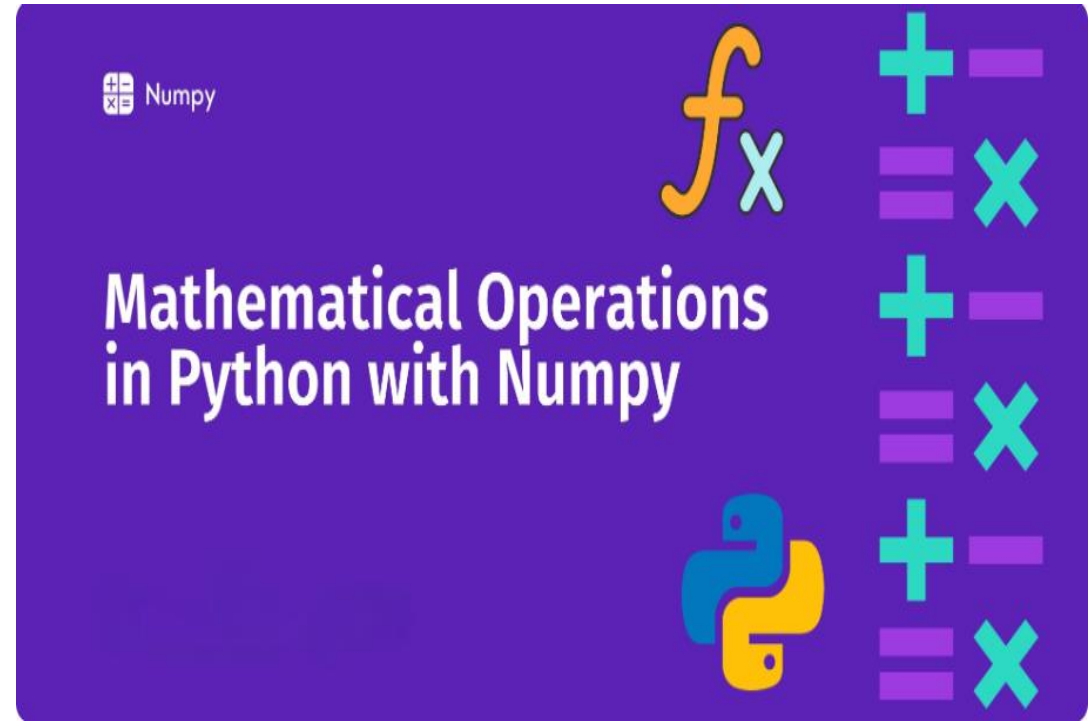


Sorted array along the last axis

01	15	19	94
----	----	----	----

Array Mathematics in NumPy

- **Arithmetic Operations (Element-wise)**
- NumPy allows **element-wise operations** between arrays of the same shape..
- That means:
- Each element in one array is operated on with the **corresponding element** in the other array.
- **Rule:** Arrays must be the **same shape** or **broadcastable** to the same shape.



Array Addition

- Adds each pair of corresponding elements
- Arrays must match in shape or support broadcasting
- Result has the **same shape** as input arrays
- You can use the numpy `np.add()` function to get the elementwise sum of two numpy arrays. The `+` operator can also be used as a shorthand for applying `np.add()` on numpy arrays

Example:

`[1,2,3]+[4,5,6] = [5,7,9]`

The diagram shows the addition of two 2x2 arrays. The first array has elements 1 (blue), 2 (yellow), 0 (green), and 3 (orange). The second array has elements 4 (blue), 1 (yellow), 2 (green), and 2 (orange). The result array has elements 5 (blue), 3 (yellow), 2 (green), and 5 (orange). The elements are summed pairwise: 1+4=5, 2+1=3, 0+2=2, and 3+2=5.

1	2
0	3

 +

4	1
2	2

 =

5	3
2	5

Array Subtraction

- Subtracts one array from another, element by element
- Like addition, supports same shape or broadcasting

NUMPY SUBTRACT PERFORMS
ELEMENT-WISE SUBTRACTION OF THE INPUT VALUES

a_1	a_2	a_3
a_4	a_5	a_6
a_7	a_8	a_9

 $-$

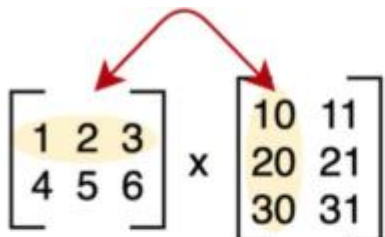
b_1	b_2	b_3
b_4	b_5	b_6
b_7	b_8	b_9

 $=$

a_1-b_1	a_2-b_2	a_3-b_3
a_4-b_4	a_5-b_5	a_6-b_6
a_7-b_7	a_8-b_8	a_9-b_9

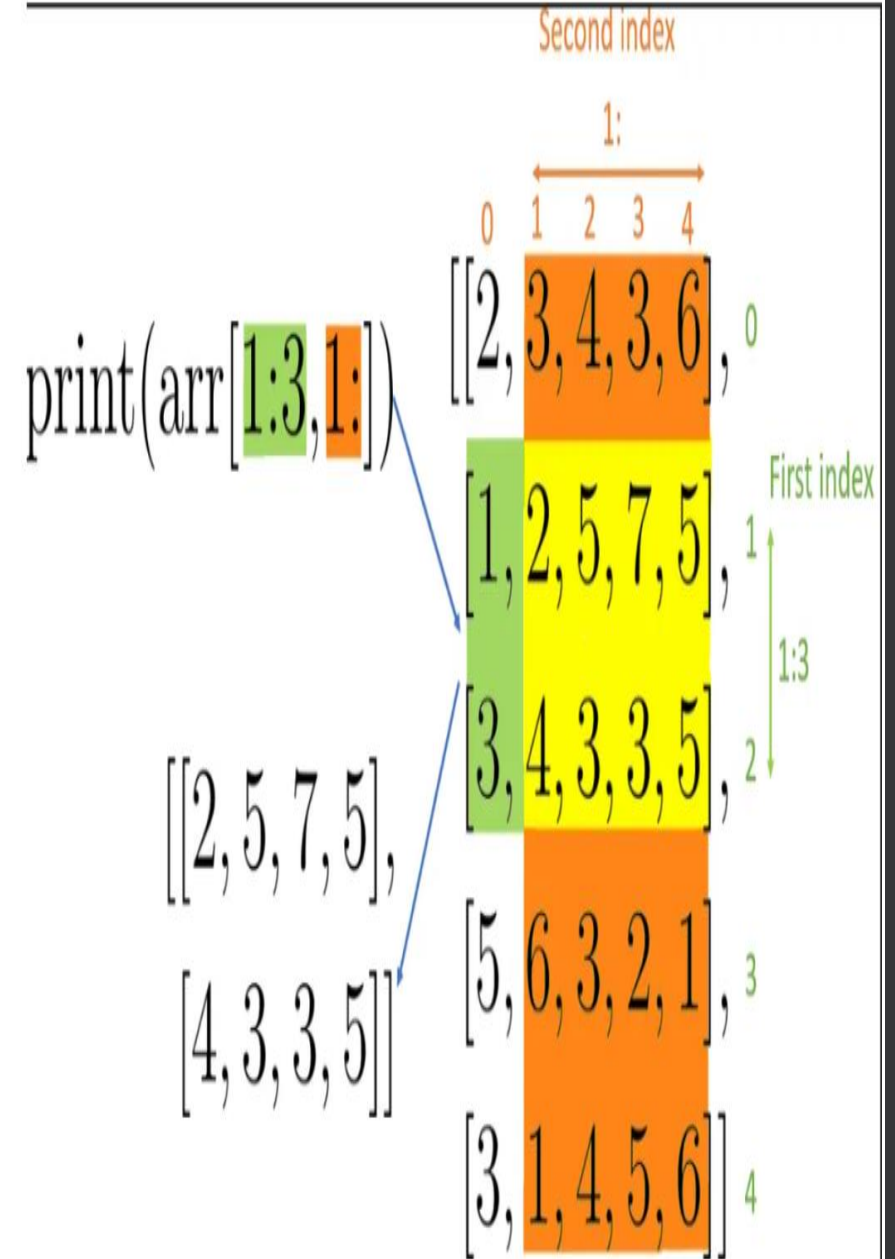
• Array Multiplication

- You can use `np.multiply` to multiply two same-sized arrays together.
- One way to use `np.multiply`, is to have the two input arrays be the exact same shape (i.e., they have the same number of rows and columns). If the input arrays have the same shape, then the Numpy multiply function will multiply the values of the inputs pairwise.
- Alternatively, if the two input arrays are *not* the same size, then one of the arrays must have a shape that can be broadcasted across the other array.
- Broadcasting concept explained the next slide.


$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \end{bmatrix}$$
$$= \begin{bmatrix} 1 \times 10 + 2 \times 20 + 3 \times 30 & 1 \times 11 + 2 \times 21 + 3 \times 31 \\ 4 \times 10 + 5 \times 20 + 6 \times 30 & 4 \times 11 + 5 \times 21 + 6 \times 31 \end{bmatrix}$$
$$= \begin{bmatrix} 10+40+90 & 11+42+93 \\ 40+100+180 & 44+105+186 \end{bmatrix} = \begin{bmatrix} 140 & 146 \\ 320 & 335 \end{bmatrix}$$

Broadcasting in NumPy?

- **Broadcasting** allows NumPy to perform element-wise operations on arrays of different shapes — *without explicitly copying data*.
- Broadcasting “stretches” smaller arrays so they match the shape of larger arrays for mathematical operations.
- **Why is Broadcasting Useful?**
 - Lets you write clean, vectorized code.
 - Avoids using loops → much faster and more memory-efficient.
 - Makes operations between arrays of different shapes possible.



Division of Arrays

- Divides each element by the corresponding one in the other array.
- Result is a **float array**, even if inputs are integers.
- The most important way to use this function is to divide two same-sized arrays. When you divide two same sized arrays, np.divide will divide values of the arrays, element-wise.
- Element-wise Modulo (Remainder) in NumPy
- **What Is Modulo?**
- The **modulo** operation finds the **remainder** after division of one number by another.
- In math: $a \bmod b = \text{remainder when } a \text{ is divided by } b$
- Rules:
- Arrays must be the same shape, or compatible for broadcasting.

NUMPY DIVIDE COMPUTES
THE ELEMENT-WISE DIVISION OF THE INPUT VALUES

a_1	a_2	a_3
a_4	a_5	a_6
a_7	a_8	a_9

 /

b_1	b_2	b_3
b_4	b_5	b_6
b_7	b_8	b_9

 =

a_1/b_1	a_2/b_2	a_3/b_3
a_4/b_4	a_5/b_5	a_6/b_6
a_7/b_7	a_8/b_8	a_9/b_9

Python Modulo Operator



Example

$$15 \% 7 = 1$$

Reshape in NumPy

- The `reshape()` function in NumPy allows you to change the shape (i.e., the number of rows and columns) of an array without changing its data.
- **`import numpy as np`**
- `arr = np.array([1, 2, 3, 4, 5, 6])`
- `reshaped = arr.reshape(2, 3)`
- `print(reshaped)`
- Output
- `[[1 2 3]`
- `[4 5 6]]`

data

1
2
3
4
5
6

data.reshape(2,3)

1	2	3
4	5	6

data.reshape(3,2)

1	2
3	4
5	6

NumPy Array Iterating

- Iterating Arrays
- Iterating means going through elements one by one.
- As we deal with multi-dimensional arrays in numpy, we can do this using basic **for** loop of python.
- If we iterate on a 1-D array it will go through each element one by one.
- Example:
 - Iterate on the elements of the following 1-D array:
 - **import numpy as np**

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

for x in arr:
 print(x)

NumPy for loop

```
import numpy as np
arr1 = np.array([[2, 1, 4],[2, 4, 6]])
arr2 = np.array([[8, 16, 44],[22, 40, 16]])
arr3 = np.array([[7, 14, 21],[0, 4, 7]])
for x in arr1, arr2, arr3:
    print(x)
```

```
[[2 1 4]
 [2 4 6]]
[[ 8 16 44]
 [22 40 16]]
[[ 7 14 21]
 [ 0  4  7]]
```



**Iterating Numpy
Array**

