

## What is NumPy?

**NumPy** stands for Numerical Python and is used to operate efficient computations of arrays and matrices behind the scenes of machine learning models.

The building block of **Numpy** is the array, which is a data structure very similar to the list, with the difference that it provides a huge amount of mathematical functions.

In other words, the **Numpy** array is a multidimensional array object. It provides:

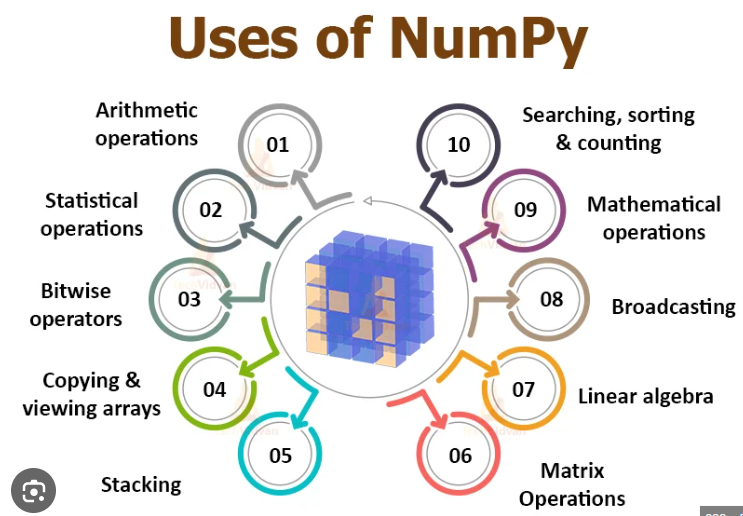
* A fast and efficient array object: ndarray
* Tools for performing **mathematical, logical, statistical, and algebraic operations** on arrays

## Why Use NumPy in Data Science?

Data Science involves a lot of **data manipulation**, **analysis**, and **mathematical operations** — NumPy makes all of this fast and efficient.

### Key Features of NumPy:

* Multidimensional arrays
* Broadcasting (automatic reshaping during operations)
* Linear algebra and statistical operations
* Data manipulation and transformation
* Fast performance (uses C under the hood)



## Example: Simple NumPy Use

**import numpy as np**

**data = np.array([1, 2, 3, 4, 5])**

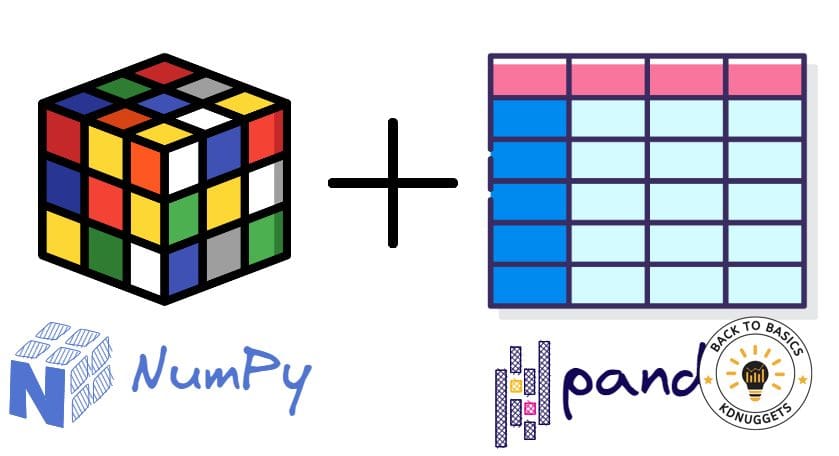
**print(data.mean()) # Average value: 3.0**

**print(data.std()) # Standard deviation**

**print(data \* 2) # Element-wise multiplication**

## NumPy With Other Data Science Libraries

|  |  |  |
| --- | --- | --- |
| **Library** | **Purpose** | **How NumPy Helps** |
| **Pandas** | Data manipulation (tables: DataFrame) | Pandas is built on top of NumPy |
| **Matplotlib** | Data visualization (charts and plots) | Uses NumPy arrays to plot graphs |
| **Scikit-learn** | Machine learning models | Uses NumPy arrays for model input/output |
| **TensorFlow / PyTorch** | Deep learning | Uses NumPy-like tensors for computations |
| **SciPy** | Advanced scientific computing | Extends NumPy for more complex operations |



## Example Workflow in Data Science

1. **Load Data**: Use pandas.read\_csv()
2. **Convert to NumPy**: df.values → NumPy array
3. **Clean/Transform Data**: Use NumPy operations
4. **Visualize**: Use matplotlib or seaborn
5. **Model Training**: Feed NumPy arrays into scikit-learn or other ML libraries

## In Short

**NumPy is the foundation of scientific computing in Python**, especially for data science and machine learning.

Without NumPy, most of the popular Python data science libraries wouldn’t exist or perform as efficiently.

1. NumPy array creation
2. NumPy array from existing data
3. NumPy array from numerical ranges
4. NumPy array manipulation
5. NumPy indexing and slicing
6. NumPy Broadcasting

=====================================================

## 1. ARRAY CREATION

### np.array()

**a = np.array([2, 3, 4])**

**print(a)**

* **What it does:** Creates a NumPy array from a list [2, 3, 4].
* **Why it's useful:** Converts normal Python lists to more powerful NumPy arrays for mathematical operations.

### np.empty()

**c = np.empty((2, 3), dtype=int)**

**print(c)**

* **What it does:** Creates a 2x3 array of integers with **random values (uninitialized)**.
* **Why it's useful:** Faster when you plan to fill it right after, since it skips setting initial values.

### np.zeros()

**d = np.zeros(5)**

**print(d)**

* **What it does:** Creates a 1D array with **five zeroes**.
* **Why it's useful:** You often need an array filled with 0s (e.g., initializing counters, masks, etc.).

### np.ones()

**f = np.ones(5)**

**print(f)**

* **What it does:** Creates a 1D array with **five ones**.
* **Why it's useful:** Handy for multiplication, default values, and initializations.

### np.arange()

**h = np.arange(10, 20, 2)**

**print(h)**

* **What it does:** Like Python's range(), it creates numbers from 10 to 18, skipping by 2.
* **Why it's useful:** Great for generating ranges of numbers without writing loops.

### np.linspace()

**j = np.linspace(2.0, 4.0, num=5)**

**print(j)**

* **What it does:** Creates 5 evenly spaced numbers from 2.0 to 4.0.
* **Why it's useful:** Best when you want smooth intervals (e.g., in plotting graphs or interpolation).

## 2. FROM EXISTING DATA

### np.asarray()

**arr1 = np.asarray([1, 2, 3])**

**print(arr1)**

* **What it does:** Converts a list to an array. If already an array, it returns the original (no copy).
* **Why it's useful:** Useful when writing functions that accept either lists or arrays.

### np.frombuffer()

**arr3 = np.frombuffer(b'\x01\x02\x03\x04', dtype=np.uint8)**

**print(arr3)**

* **What it does:** Turns raw bytes into an array.
* **Why it's useful:** Advanced usage—efficient when reading binary files or network data.

### np.fromiter()

**gen = (x\*x for x in range(5))**

**arr4 = np.fromiter(gen, dtype=int)**

**print(arr4)**

* **What it does:** Converts an iterator (here: squares of numbers) to a NumPy array.
* **Why it's useful:** Memory efficient when creating large arrays from a generator.

## 3. NUMERICAL RANGES

### np.logspace()

**lsp = np.logspace(1, 2, num=5)**

**print(lsp)**

* **What it does:** Returns numbers between 10^1 and 10^2 (log scale), evenly spaced.
* **Why it's useful:** Common in plotting where you want values like 10, 100, etc.

## 4. ARRAY MANIPULATION

### reshape(), ravel(), flatten()

**x = np.arange(12)**

**import numpy as np**

**x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])**

**X = x.reshape((3, 4)) # 3 rows, 4 columns**

**print(X)**

**r = X.ravel()**

**print(r)**

** Converts the 2D array back into a 1D array.**

** ravel() returns a view of the original array. If you change r, the original X might also change.**

**f = X.flatten()**

**print(f)**

** Also flattens the 2D array into a 1D array.**

** But this time it returns a copy, not a view — changes to f will not affect the original X.**

* **reshape:** Changes the shape without changing the data.
* **ravel():** Flattens the array (returns a view).
* **flatten():** Flattens the array (returns a copy).

### Broadcasting

**A = np.array([[0, 0, 0], [10, 10, 10]])**

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

**print(A + B)**

* **What it does:** B is “stretched” to match A’s shape row-wise.
* **Why it's useful:** You avoid writing loops. Very efficient.

### broadcast\_to()

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

**print(np.broadcast\_to(y, (3, 3)))**

* **What it does:** Copies y into 3 rows to match the shape.
* **Why it's useful:** Helps in vectorized calculations where shapes must match.

## 5. JOINING AND SPLITTING ARRAYS

### concatenate(), stack(), hstack(), vstack()

**np.concatenate([a1, a2], axis=0) # joins vertically**

**np.hstack([a1, a2]) # joins side-by-side**

**np.vstack([a1, a2]) # joins one below another**

**import numpy as np**

**a1 = np.array([[1, 2], [3, 4]])**

**a2 = np.array([[5, 6], [7, 8]])**

**result = np.concatenate([a1, a2], axis=0)**

**print(result)**

**result = np.hstack([a1, a2])**

**print(result)**

**result = np.vstack([a1, a2])**

**print(result)**

**Columns from a2 are added beside the columns of a1.**

* **Why it's useful:** You often need to combine datasets or matrix pieces together.

### split(), hsplit(), vsplit()

**arr = np.arange(8)**

**np.split(arr, 4) # splits into 4 parts: [0,1], [2,3], ...**

* **Why it's useful:** Lets you break large arrays into smaller pieces (e.g., mini-batches).

### append(), insert(), delete(), unique()

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

**np.append(arr, [4,5]) # [1 2 3 4 5]**

**np.insert(arr, 1, 9) # [1 9 2 3]**

**np.delete(arr, 2) # [1 2]**

**np.unique([1,2,2,3,3,3]) # [1 2 3]**

**import numpy as np**

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

**new\_arr = np.append(arr, [4, 5])**

**print(new\_arr)**

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

**new\_arr = np.insert(arr, 1, 9)**

**print(new\_arr)**

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

**new\_arr = np.delete(arr, 2)**

**print(new\_arr)**

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

**unique\_arr = np.unique(arr)**

**print(unique\_arr)**

**------------------------------------------------------------------------------------------------**

* **Why it's useful:** For modifying arrays like lists (add, remove, clean up duplicates).

## 6. INDEXING AND SLICING

**import numpy as np**

**a = np.array([10, 20, 30, 40, 50])**

**print(a[0]) # 10**

**print(a[2]) # 30**

**print(a[-1]) # 50 (last element)**

**b = np.array([[1, 2, 3],**

**[4, 5, 6],**

**[7, 8, 9]])**

**print(b[0, 0]) # 1 (1st row, 1st column)**

**print(b[1, 2]) # 6 (2nd row, 3rd column)**

**print(b[-1, -2]) # 8 (last row, second last column)**

**slicing**

**a = np.array([10, 20, 30, 40, 50])**

**print(a[1:4]) # [20 30 40]**

**print(a[:3]) # [10 20 30]**

**print(a[3:]) # [40 50]**

**print(a[::2]) # [10 30 50] (every 2nd element)**

**b = np.array([[1, 2, 3],**

**[4, 5, 6],**

**[7, 8, 9]])**

**print(b[0:2, 1:3]) # Rows 0-1, Columns 1-2**

**# Output:**

**# [[2 3]**

**# [5 6]]**

**print(b[:, 0]) # First column from all rows: [1 4 7]**

**print(b[1, :]) # Second row: [4 5 6]**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Meaning** | **Example** | **Output** |
| a[i] | Element at index i | a[2] | 30 |
| a[start:stop] | Elements from start to stop-1 | a[1:4] | [20 30 40] |
| a[:n] | First n elements | a[:3] | [10 20 30] |
| a[m:] | From index m to end | a[2:] | [30 40 50] |
| a[::step] | Every step-th element | a[::2] | [10 30 50] |
| a[-1] | Last element | a[-1] | 50 |

* **Slicing:** Like lists, use [start:stop:step].
* **Why it's useful:** Lets you extract rows, columns, or specific parts easily.