**Numpy Tutorials**

NumPy (short for **Numerical Python**) is one of the most fundamental libraries in [Python](https://www.geeksforgeeks.org/python-programming-language-tutorial/)for scientific computing. It provides support for large, multi-dimensional arrays and matrices along with a collection of mathematical functions to operate on arrays.

At its core it introduces the [ndarray (n-dimensional array)](https://www.geeksforgeeks.org/numpy-ndarray/" \t "_blank) object which allows us to store and manipulate large datasets in a memory-efficient manner. Unlike Python’s built-in lists, NumPy arrays are homogeneous and enable faster operations.

***Important Facts to Know :***

* ***Vectorized Operations****: NumPy operations are faster than Python lists because they use optimized C-based functions.*
* ***Broadcasting Feature****: NumPy allows operations between arrays of different shapes without explicit looping known as* [***broadcasting***](https://www.geeksforgeeks.org/numpy-array-broadcasting/)*making it easier to handle large datasets.*

**What is NumPy Used for?**

With NumPy, you can perform a wide range of numerical operations, including:

* Creating and manipulating arrays.
* Performing element-wise and matrix operations.
* Generating random numbers and statistical calculations.
* Conducting linear algebra operations.
* Working with Fourier transformations.
* Handling missing values efficiently in datasets.

**Why Learn NumPy?**

* NumPy speeds up math operations like addition and multiplication on large groups of numbers compared to regular Python..
* It’s good for handling large lists of numbers (arrays), so you don’t have to write complicated loops.
* It gives ready-to-use functions for statistics, algebra and random numbers.
* Libraries like Pandas, SciPy, TensorFlow and many others are built on top of NumPy.
* NumPy uses less memory and stores data more efficiently, which matters when working with lots of data.

**Ways to Create Numpy Arrays**

Below are some of the ways by which we can create NumPy Arrays in [Python](https://www.geeksforgeeks.org/python-programming-language/):

**Create Numpy Arrays Using Lists or Tuples**

The simplest way to create a [NumPy array](https://www.geeksforgeeks.org/numpy-array-in-python/) is by passing a Python list or tuple to the numpy.array() function. This method creates a one-dimensional array.

**Initialize a Python NumPy Array Using Special Functions**

NumPy provides several built-in functions to generate arrays with specific properties.

* [np.zeros()](https://www.geeksforgeeks.org/numpy-zeros-python/): Creates an array filled with zeros.

numpy.zeros(shape, dtype=float, order='C', \*, like=None)

🔹 Parameters:

Parameter Description

shape (Required): Integer or tuple of integers. Defines the shape of the output array.

For example:

• 3 → 1D array with 3 elements.

• (2, 3) → 2D array with 2 rows and 3 columns.

dtype (Optional): Data type of the output array. Default is float.

Common values: int, float, complex, bool, object, str, etc.

order (Optional): Specifies how multi-dimensional data is stored in memory.

• 'C': C-style row-major (default).

• 'F': Fortran-style column-major.

like (Optional, added in NumPy 1.20): Reference object to allow creation of arrays that are not NumPy arrays. Used for compatibility with other array libraries (like CuPy).

If specified, the output array is created like the passed object.

* [np.ones()](https://www.geeksforgeeks.org/numpy-ones-python/): Creates an array filled with ones.

numpy.ones(shape, dtype=None, order='C', \*, like=None)

🔹 Parameters:

Parameter Description

shape (Required): Integer or tuple of integers that define the shape of the array.

Examples: 3 → 1D array with 3 elements; (2, 3) → 2D array with 2 rows and 3 columns.

dtype (Optional): The desired data type of the array.

Default is float (i.e., float64). You can specify types like int, bool, complex, etc.

order (Optional): Memory layout of the array:

• 'C': Row-major (C-style)

• 'F': Column-major (Fortran-style)

like (Optional, NumPy 1.20+): Allows the creation of arrays that are not necessarily NumPy arrays, useful for compatibility with other array libraries (like CuPy, Dask, etc.).

* [np.full():](https://www.geeksforgeeks.org/numpy-full-python/)Creates an array filled with a specified value.

numpy.full(shape, fill\_value, dtype=None, order='C', \*, like=None)

🔸 Parameters:

Parameter Description

shape (Required): Tuple or int. Specifies the shape of the output array. Examples: 3, (2, 3)

fill\_value (Required): The constant value to fill the array with (e.g., 7, True, 'a', etc.).

dtype (Optional): Desired data type. If None, NumPy infers it from fill\_value.

order (Optional): Memory layout. 'C' (row-major, default) or 'F' (column-major).

like (Optional, NumPy 1.20+): Reference object to allow output array to be created with the same type as the given array.

* [np.arange()](https://www.geeksforgeeks.org/numpy-arrange-in-python/): Creates an array with values that are evenly spaced within a given range.

numpy.arange([start,] stop[, step], dtype=None, \*, like=None)

🔸 Parameters:

Parameter Description

start (Optional) Starting value of the sequence. Default is 0.

stop (Required) End of interval. Not included in the output.

step (Optional) Spacing between values. Default is 1. Can be float.

* [np.linspace()](https://www.geeksforgeeks.org/numpy-linspace/): Creates an array with values that are evenly spaced over a specified interval.

numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)

🔸 Parameters:

Parameter Description

start The starting value of the interval.

stop The end value of the interval.

num (Optional) Number of samples to generate. Default is 50.

endpoint (Optional) If True (default), stop is included in the result. If False, stop is excluded.

retstep (Optional) If True, return a tuple: (array, step size) between values.

dtype (Optional) The desired data type of the result.

axis (Optional, NumPy 1.16+) The axis along which the values are stored. Default is 0.

**Create Python Numpy Arrays Using Random Number Generation**

NumPy provides functions to create arrays filled with random numbers.

* [np.random.rand()](https://www.geeksforgeeks.org/numpy-random-rand-python/): Creates an array of specified shape and fills it with random values sampled from a uniform distribution over [0, 1).

The np.random.rand() function is used to **generate random numbers** from a **uniform distribution** over the interval **[0, 1)** — i.e., numbers are greater than or equal to 0 and less than 1.

It returns values drawn from a **uniform distribution**, which means each value in the interval has equal probability.

**✅ Syntax:**

np.random.rand(d0, d1, ..., dn)

It is a shorthand for:

np.random.random\_sample(size=(d0, d1, ..., dn))

**🔸 Parameters:**

| **Parameter** | **Description** |
| --- | --- |
| d0, d1, ..., dn | *(Optional)* Dimensions of the returned array. If no argument is passed, returns a single float. |

**Key Points:**

* Values ∈ [0, 1)
* Uniform distribution
* All values are positive (good for image pixels, probabilities, etc.)
* [np.random.randn()](https://www.geeksforgeeks.org/numpy-random-randn-python/): Creates an array of specified shape and fills it with random values sampled from a standard normal distribution.

The np.random.randn() function is used to **generate random numbers from the standard normal distribution** (also known as the Gaussian distribution).

This means:

* **Mean (μ)** = 0
* **Standard Deviation (σ)** = 1

It generates samples that are distributed according to the **bell-shaped curve**.

**✅ Syntax:**

np.random.randn(d0, d1, ..., dn)

It's a shorthand for:

np.random.standard\_normal(size=(d0, d1, ..., dn))

**🔸 Parameters:**

| **Parameter** | **Description** |
| --- | --- |
| d0, d1, ..., dn | *(Optional)* The dimensions of the returned array. If no argument is given, a single float is returned. |

**Key Points:**

* Values can be **negative or positive**
* Follows the **bell curve** (standard normal distribution)
* Good for simulating noise, initializing weights, etc.
* [np.random.randint()](https://www.geeksforgeeks.org/random-sampling-in-numpy-randint-function/): Creates an array of specified shape and fills it with random integers within a given range.

The np.random.randint() function is used to generate **random integers** from a specified range.

It returns integers **randomly sampled** from a **uniform discrete distribution** over a specified interval — meaning each integer in the range has equal probability.

**✅ Syntax:**

np.random.randint(low, high=None, size=None, dtype=int, endpoint=False)

**🔸 Parameters:**

| **Parameter** | **Description** |
| --- | --- |
| **low** | **Required.** Lowest (inclusive) value of the range if high is specified. Otherwise, it's treated as the upper bound (exclusive), and 0 is used as the lower bound. |
| **high** | *(Optional)* Upper bound (exclusive by default, inclusive if endpoint=True). |
| **size** | *(Optional)* Output shape. If None, returns a single integer. Otherwise, returns an array of shape size. |
| **dtype** | *(Optional)* Desired integer type (e.g., int32, int64). Default is int. |
| **endpoint** | *(Optional, NumPy 1.20+)* If True, include high as a possible result. Default is False. |

**Create Python Numpy Arrays Using Matrix Creation Routines**

NumPy provides functions to create specific types of matrices.

* [np.eye()](https://www.geeksforgeeks.org/numpy-eye-python/): Creates an identity matrix of specified size.

The np.eye() function is used to create a **2D identity matrix** or a **matrix with ones on a specified diagonal** and zeros elsewhere.

An **identity matrix** is a square matrix with 1s on the main diagonal and 0s everywhere else. It's widely used in linear algebra and machine learning.

**✅ Syntax:**

np.eye(N, M=None, k=0, dtype=float, order='C', \*, like=None)

**🔸 Parameters:**

| **Parameter** | **Description** |
| --- | --- |
| N | **Required**. Number of rows in the output matrix. |
| M | *(Optional)* Number of columns. If None, it defaults to N (makes a square matrix). |
| k | *(Optional)* Index of the diagonal:  k=0 (main diagonal),  k>0 (above main),  k<0 (below main). |
| dtype | *(Optional)* Data type of the returned array. Default is float. |
| order | *(Optional)* Memory layout: 'C' for row-major, 'F' for column-major. |
| like | *(Optional)* Allows creating the output array using an object that implements the \_\_array\_function\_\_ protocol (advanced use). |

* np.identity():

The np.identity() function is used to create an **identity matrix** of a given size. The identity matrix is a **square matrix** with 1s along the main diagonal (from top left to bottom right) and 0s everywhere else.

It's a special kind of matrix that, when multiplied by another matrix, returns the other matrix unchanged (this is the **identity** property in matrix multiplication).

**✅ Syntax:**

np.identity(n, dtype=float)

**🔸 Parameters:**

| **Parameter** | **Description** |
| --- | --- |
| **n** | **Required.** The size of the identity matrix, i.e., it creates an n x n matrix. |
| **dtype** | *(Optional)* The desired data type of the returned array. By default, it is float. |

**🧠 Returns:**

* A square **2D array** of shape (n, n) where the diagonal elements are 1 and all other elements are 0.
* [np.diag()](https://www.geeksforgeeks.org/numpy-diag-python/): Constructs a diagonal array.

The np.diag() function in NumPy is used for two main purposes:

1. **Creating a diagonal matrix** from a 1D array.
2. **Extracting a diagonal** from a 2D matrix (i.e., the main diagonal or any other specified diagonal).

**✅ Syntax:**

python

CopyEdit

np.diag(v, k=0)

**🔸 Parameters:**

| **Parameter** | **Description** |
| --- | --- |
| **v** | **Required.** Can be a 1D array or a 2D array (matrix). If a 1D array is passed, a diagonal matrix is created with elements of the array on the diagonal. If a 2D array is passed, the diagonal(s) are extracted. |
| **k** | *(Optional)* The diagonal to extract or place values in:  - k=0: Main diagonal.  - k>0: Diagonal above the main diagonal.  - k<0: Diagonal below the main diagonal. |

**🧠 Returns:**

* **If v is a 1D array**: A 2D diagonal matrix with v placed on the main diagonal, and all other values are 0.
* **If v is a 2D array**: A 1D array with the elements from the diagonal specified by k.

**✅ Examples:**

**1. Create a diagonal matrix from a 1D array:**

import numpy as np

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

matrix = np.diag(v)

print(matrix)

**Output:**

[[1 0 0]

[0 2 0]

[0 0 3]]

In this case, the elements [1, 2, 3] are placed on the main diagonal of the resulting matrix.

**2. Extract the main diagonal from a 2D matrix:**

matrix = np.array([[1, 2, 3],

[4, 5, 6],

[7, 8, 9]])

diag = np.diag(matrix)

print(diag)

**Output:**

[1 5 9]

Here, the main diagonal elements [1, 5, 9] are extracted from the matrix.

**3. Extract the diagonal above the main diagonal (k=1):**

matrix = np.array([[1, 2, 3],

[4, 5, 6],

[7, 8, 9]])

diag\_above = np.diag(matrix, k=1)

print(diag\_above)

**Output:**

[2 6]

This extracts the diagonal elements immediately above the main diagonal.

* [np.zeros\_like()](https://www.geeksforgeeks.org/numpy-zeros_like-python/): Creates an array of zeros with the same shape and type as a given array.

**np.zeros\_like() in NumPy**

The np.zeros\_like() function is used to create an array of zeros that has the same shape and data type as a given input array. It is essentially a convenient way to initialize a zero-filled array that matches the shape and type of another array.

**✅ Syntax:**

python

CopyEdit

np.zeros\_like(a, dtype=None, order='K', subok=True, like=None)

**🔸 Parameters:**

| **Parameter** | **Description** |
| --- | --- |
| **a** | **Required.** The input array whose shape and data type are to be copied. |
| **dtype** | *(Optional)* The desired data type of the returned array. If None, it uses the data type of a. |
| **order** | *(Optional)* Memory layout order. 'C' means row-major (C-style), 'F' means column-major (Fortran-style). The default is 'K', which preserves the memory layout of the input array. |
| **subok** | *(Optional)* If True, the returned array will be a subclass of a (if a is a subclass). |
| **like** | *(Optional)* This parameter allows the creation of the output array using an object that implements the \_\_array\_function\_\_ protocol (advanced use). |

**🧠 Returns:**

A new array filled with zeros that has the same shape and data type as the input array a.

**✅ Examples:**

**1. Create an array of zeros with the same shape as a given array:**

import numpy as np

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

zeros\_array = np.zeros\_like(a)

print(zeros\_array)

**Output:**

[0 0 0 0]

Here, a new array of zeros is created with the same shape as a (1D array).

**2. Create a 2D array of zeros with the same shape and type as a given 2D array:**

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

zeros\_array = np.zeros\_like(a)

print(zeros\_array)

**Output:**

[[0 0]

[0 0]]

* [np.ones\_like()](https://www.geeksforgeeks.org/numpy-ones_like-python/): Creates an array of ones with the same shape and type as a given array.

The np.ones\_like() function creates a new array filled with ones, having the same shape and data type as a given input array. It's similar to np.zeros\_like() but instead of zeros, it initializes the array with ones.

**✅ Syntax:**

np.ones\_like(a, dtype=None, order='K', subok=True, like=None)

**🔸 Parameters:**

| **Parameter** | **Description** |
| --- | --- |
| **a** | **Required.** The input array whose shape and data type are to be copied. |
| **dtype** | *(Optional)* The desired data type of the returned array. If None, it uses the data type of a. |
| **order** | *(Optional)* Memory layout order. 'C' means row-major (C-style), 'F' means column-major (Fortran-style). The default is 'K', which preserves the memory layout of the input array. |
| **subok** | *(Optional)* If True, the returned array will be a subclass of a (if a is a subclass). |
| **like** | *(Optional)* This parameter allows the creation of the output array using an object that implements the \_\_array\_function\_\_ protocol (advanced use). |

**🧠 Returns:**

A new array filled with ones that has the same shape and data type as the input array a.

**✅ Examples:**

**1. Create an array of ones with the same shape as a given array:**

import numpy as np

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

ones\_array = np.ones\_like(a)

print(ones\_array)

**Output:**

[1 1 1 1]

Here, a new array of ones is created with the same shape as a (1D array).

**2. Create a 2D array of ones with the same shape and type as a given 2D array:**

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

ones\_array = np.ones\_like(a)

print(ones\_array)

**Output:**

[[1 1]

[1 1]]

In this case, a 2D array of ones is created, matching the shape of a (2x2 matrix).

Practicing **NumPy** is a must if you're diving into data science or machine learning in Python. Here’s a curated list of the **most important NumPy methods and classes**, grouped by topic, so you can structure your practice effectively.

**✅ 1. NumPy Basics**

**👉 Core Class**

* np.ndarray — main class for array objects

**👉 Key Functions**

* np.array() — create array from list/tuple
* np.arange() — range of values (like range())
* np.linspace() — linearly spaced values
* np.ones(), np.zeros(), np.full() — arrays filled with a value
* np.eye() — identity matrix
* np.empty() — uninitialized array (fast, but beware!)
* np.shape, np.dtype, np.ndim, np.size — array metadata

**✅ 2. Array Indexing, Slicing, Reshaping**

* array[i], array[i:j], array[:, 1] — basic slicing
* np.reshape() — change shape
* np.ravel() / np.flatten() — flatten array
* np.transpose() / array.T — transpose
* np.expand\_dims(), np.squeeze() — add/remove dimensions
* np.concatenate(), np.vstack(), np.hstack() — joining arrays
* np.split(), np.array\_split() — splitting arrays

**✅ 3. Math & Stats Functions**

* np.sum(), np.mean(), np.median(), np.std(), np.var()
* np.min(), np.max(), np.argmin(), np.argmax()
* np.cumsum(), np.cumprod()
* np.percentile(), np.quantile()
* np.corrcoef(), np.cov()

**✅ 4. Element-wise Operations**

* +, -, \*, /, \*\*, //, % — element-wise operators
* np.add(), np.subtract(), np.multiply(), np.divide()
* np.sqrt(), np.exp(), np.log(), np.power()
* np.sin(), np.cos(), np.tan(), np.pi
* np.clip() — limit values

**✅ 5. Logical & Comparison Operations**

* np.where(condition, x, y) — vectorized if-else
* np.logical\_and(), np.logical\_or(), np.logical\_not()
* array > 5, array == 0, etc.
* np.any(), np.all()

**✅ 6. Random Number Generation**

(Use numpy.random module)

* np.random.rand() — uniform [0,1)
* np.random.randn() — standard normal
* np.random.randint(low, high, size)
* np.random.choice(array)
* np.random.shuffle(array) — in-place
* np.random.seed() — reproducibility

**✅ 7. Linear Algebra**

(Use numpy.linalg module)

* np.linalg.inv() — matrix inverse
* np.linalg.det() — determinant
* np.linalg.eig() — eigenvalues/vectors
* np.linalg.norm() — vector norm
* np.dot(), np.matmul() — dot/matrix product
* np.trace() — sum of diagonal

**✅ 8. File I/O**

* np.loadtxt(), np.genfromtxt() — read from text/CSV
* np.save(), np.load() — binary .npy files
* np.savetxt() — save to text

**✅ 9. Broadcasting and Vectorization**

* Learn how NumPy applies operations to arrays of different shapes using **broadcasting rules**
* Vectorized operations = fast and clean code

**✅ 10. Advanced Topics** meshgrid, unique, sort, argsort, isin, setdiff1d, intersect1d, tile, repeat, fromfunction, vectorize

* np.meshgrid() — for 2D grid creation
* np.unique(), np.sort(), np.argsort()
* np.isin(), np.setdiff1d(), np.intersect1d()
* np.tile(), np.repeat()
* np.fromfunction(), np.vectorize() — functional programming